LLNL-TR-815869

LAWRENCE LIVERMORE NATIONAL LABORATORY

The Multiphysics on Advanced Platforms Project

R. Rieben, K. Weiss

October 21, 2020

This work performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344.

The Multiphysics on Advanced Platforms Project [1]

R. Anderson, A. Black, L. Busby, B. Blakeley, R. Bleile, J.-S. Camier, J. Ciurej, A. Cook, V. Dobrev, N. Elliott, J. Grondalski, C. Harrison, R. Hornung, Tz. Kolev, M. Legendre, W. Liu, W. Nissen, B. Olson, M. Osawe, G. Papadimitriou, O. Pearce, R. Pember, A. Skinner, D. Stevens, T. Stitt, L. Taylor, V. Tomov, **R. Rieben**[2], A. Vargas, K. Weiss[3], D. White

November 25, 2020

[1]This work was performed 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-TR-815869

[2]Project Lead, corresponding author, Lawrence Livermore National Laboratory, `rieben1@llnl.gov`

[3]Computer Science Lead, Lawrence Livermore National Laboratory, `kweiss@llnl.gov`

Contents

List of Figures

List of Tables

Executive Summary

0.1 The Multiphysics on Advanced Platforms Project (MAPP)

In 2015, the Lawrence Livermore National Laboratory started development of next-generation multiphysics simulation capabilities for the National Nuclear Security Administration under the Advanced Technologies Development and Mitigation (ATDM) element of the Advanced Simulation and Computing program in collaboration with the Exascale Computing Project (ECP). A key driver for this effort across the NNSA tri-lab was the emergence of advanced high performance computing (HPC) architectures based on heterogeneous compute capabilities, including GPU based systems, as part of the national drive toward exascale computing platforms at multiple Department of Energy (DOE) facilities.

Developing a multiphysics code capable of meeting the various simulation needs of the NNSA as defined by the current generation of integrated codes (or ICs), initially developed as part of the Accelerated Strategic Computing Initiative (ASCI) program beginning in 1996, and able to scale to the current 100 petaflop class pre-exascale systems, as well the forthcoming exaflop class computers, is a daunting challenge. To accomplish this ambitious goal, LLNL has embraced two key themes: use of high-order numerical methods and a modular approach to code development. The LLNL next generation effort is organized under the Multi-Physics on Advanced Platforms Project (MAPP). A foundational component of MAPP is the Axom computer science (CS) toolkit which provides infrastructure for the development of modular, performance portable, multi-physics application codes. MARBL is a next-generation application code built on the Axom base to address the modeling needs of the high energy density physics (HEDP) community for simulating high-explosive, magnetic or laser driven experiments such as inertial confinement fusion (ICF), pulsed-power magneto-hydrodynamics (MHD), equation of state (EOS) and material strength studies as part of the NNSA's stockpile stewardship program (SSP).

MARBL is designed from inception to support multiple diverse algorithms, including Arbitrary Lagrangian-Eulerian (ALE) and direct Eulerian methods for solving the conservation laws associated with its various physics packages. A distinguishing feature of MARBL is the use of advanced, high-order numerical discretizations such as high-order finite element ALE and high-order finite difference Eulerian methods. This algorithmic diversity encompasses the ECP simulation motifs of unstructured and structured adaptive mesh refinement (AMR). High-order numerical methods were chosen because they have higher resolution/accuracy per unknown compared to standard low-order finite volume schemes and because they have computational characteristics which play to the strengths of current and emerging high-performance computing (HPC) architectures. Specifically, they have higher FLOP/byte ratios meaning that more floating-point operations are performed for each piece of data retrieved from memory. This leads to improved strong parallel scalability, better throughput on GPU platforms and increased computational efficiency.

A key goal for MARBL is enhanced end-user productivity including improved workflow for problem setup and meshing, simulation robustness, support for UQ and optimization driven ensembles, and in-situ data visualization and analysis. High-order ALE and Eulerian schemes have proven to be more robust and should significantly improve the overall analysis workflow for users. The advanced simulation capabilities provided by MARBL will improve user throughput along two axes: faster turnaround for multi-physics simulations on advanced architectures and less manual user intervention.

Success of MAPP will ultimately be determined by the degree of adoption of its simulation tools by the LLNL user community and beyond. To this end, emphasis at this relatively early stage of development has been placed on adding physics and capabilities to meet the current state of the art that users demand from today's petascale production simulation codes. In the case of MARBL, this includes coupled multi-material radiation-magneto-hydrodynamics, thermonuclear burn for ICF fusion calculations, general equations of state, material opacities and electrical conductivities, simulation diagnostics and queries, in-situ analytics and rendering, and parallel computational and file IO performance at a massive scale. In addition, performance of the new codes on advanced architectures like the GPU based Sierra and El Capitan systems at LLNL is critical. Portability of the software stack and long-term maintainability are critical as well, placing stringent demands on the integration and interoperability of high-quality production level software libraries and tools. Finally, MARBL will be the first demonstration of the viability of advanced high-order numerical approaches for production multi-physics simulation at scale in the NNSA and has already produced first-of-a-kind simulation results using such methods.

0.2 Accomplishments to Date

From its beginning, the project has had yearly milestones which are structured to develop core capabilities to enable targeted early engagements with the user community while steadily building out the multiphysics capabilities needed to meet long term mission needs and achieve our simulation challenge problems for the tri-lab 2020 L1 milestone. The milestone accomplishments for the past five years are summarized below along with a set of additional achievements in support of the FY20 goal and our ultimate goal of broader user adoption beyond this. While the official v1.0 user release is not scheduled until CY20, we believe user engagement as soon as possible is essential to provide the necessary feedback to generate a useful product. As such, we have been periodically releasing public “alpha” versions of the code to an ever increasing set of early adopters since the beginning of FY17 and incorporating their feedback into future development.

FY16 milestone accomplishments include:

- Initial deployment of Axom modular infrastructure toolkit with in-memory datastore (Sidre) and mesh aware data schema (Conduit Mesh Blueprint)
- Development of physics agnostic “main” and simulation/package abstractions for coupling physics packages with Axom infrastructure
- Successful integration of Axom CS toolkit with two modular, high order hydrodynamics packages including high order ALE and high-order direct Eulerian
- Mesh agnostic, scriptable and interactive user input based on Lua
- Successful integration of Axom datastore component and ability for parallel binary (HDF5) checkpoint/restart at scale

FY17 milestone accomplishments include:

- Initial public release of MARBL (first alpha version 0.1 for early adopters / user feedback)
- First deployment of Conduit-Carter in-situ data transfer application which utilizes mesh and field data schemas provided by Conduit Mesh Blueprint to enable simulation data transfer between physics modules
- Demonstrated ability to transfer data from high-order unstructured ALE mesh to low-order refined Eulerian mesh
- Successful demonstration of in-situ data-transfer on problems of interest at scale, including first-of-a-kind high-order ALE simulations with favorable comparisons to existing production codes

FY18 milestone accomplishments include:

- Development and release of a modular library for calculating thermonuclear (TN) reaction rates, electron-ion coupling coefficients and other commonly used plasma physics properties

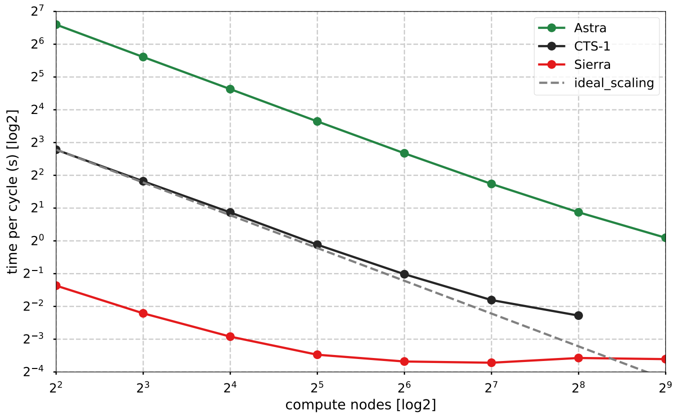
- Successful integration of a modular TN burn library in both the high-order ALE and high-order Eulerian hydro packages of MARBL
- Extended initial 2T high-order ALE rad-hydro capability to general 3T (ion+electron+radiation temperature) formulation with TN mass/energy sources
- Verified new capability with eight 3T radiation + TN burn analytic benchmarks and 1D/2D/3D simulations of ICF capsules with TN yield calculations

FY19 milestone accomplishments include:

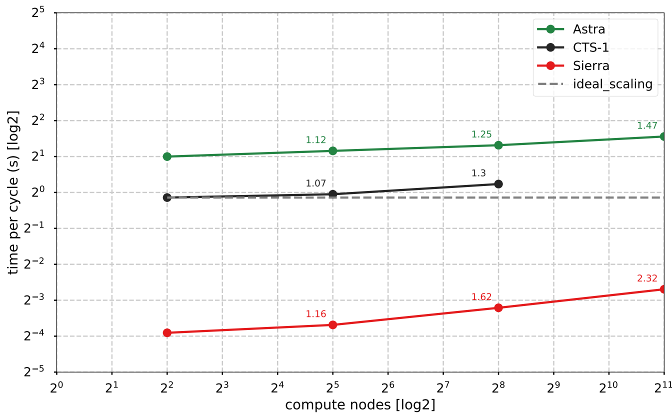
- Development of a numerical technique for conservatively remapping field and mesh data to and from a general high-order (HO) mesh and a low-order refined (LOR) mesh
- Successful integration of a forward and backward HO to LOR field transfer capability in modular software and demonstrated in an integrated calculation
- Development of a high-order, multi-rate implicit/explicit (IMEX) time stepping framework for solving coupled multi-physics problems with fast (explicit) and slow (implicit) time scales
- Fully integrated multirate IMEX approach in MARBL code and applied to radiation-magneto-hydrodynamics problems with stiff source terms, like TN burn, demonstrating ability to achieve high-order temporal convergence while keeping different time steps for fast and slow physics
- Performed dynamic two-way coupling in problems of interest
- Performed code-to-code comparisons on problems of interest with good agreement

FY20 milestone accomplishments include:

- Successful integration of multiple GPU capable libraries to enable performance portability including MFEM v4.0, RAJA, Umpire and Caliper
- Full code stack ports to Sierra and Astra platforms
- Successful conversion of high-order ALE hydro from full matrix assembly algorithms to high-performance partially assembled "matrix-free" algorithms
- Full GPU ports of high-order ALE and high-order Eulerian hydrodynamics
- Achieved $> 15X$ GPU node vs CPU node speedup for high-order ALE hydro
- Achieved $> 10X$ GPU node vs CPU node speedup for high-order direct Eulerian hydro
- Established detailed performance monitoring and tracking ability using Caliper and SPOT and automated performance testing
- Development of a new scaling study capability for generating strong, weak and throughput scaling data for CTS and ATS platforms
- Successfully scaled high-order ALE hydro to $\frac{1}{2}$ of Sierra and all of Astra (see Figure 1).
- Performed first of a kind multiphysics simulations at large scale on Sierra taking full advantage of GPU acceleration
- Performed most of FY20 work remotely during pandemic
- Meeting original project deliverables despite work on Sierra/Astra being impacted by pandemic



(a) Node-to-node strong scaling study



(b) Node-to-node weak scaling study

Figure 1: Node-to-node strong scaling (a) and weak scaling (b) studies for Lagrangian Triple-Pt-3D problem on an unstructured NURBS mesh. Our weak scaling study scales out to 2048 compute nodes, comprising half of Sierra and more than 80% of Astra. The annotations by each data point indicate the weak scaling efficiency with respect to the first data point in the series. The rest of our scaling study, including runs on all of Astra can be found in Section 3.8.

In addition to this work, the team has been engaged in many other aspects of code development to meet the many needs not explicitly captured by the milestones. Other major accomplishments include:

- As part of CEED co-design interaction, participated in the development of Laghos high-order Lagrange hydro and Remhos high-order ALE remap mini-apps (proxies for MARBL ALE hydro) and used these to inform refactor of MARBL for matrix-free partial assembly algorithms
- Development of a novel high-order finite element ALE MHD capability
- Development of high-performance point in cell lookup for arbitrary order finite element tracer particle queries
- Integration of Ascent library for in-line volume rendering / radiography
- Initial Jupyter notebook interface to MARBL code
- Further public releases of code including v0.2, v0.3, v0.4, v0.5 and v0.6
- Integration of novel high-order mesh optimization algorithms (TMOP) from MFEM library
- Non-conforming mesh refinement on arbitrary order meshes for all ALE physics using MFEM and development of initial high-order ALE AMR capability
- Integration of SAMRAI patch based structured adaptive mesh refinement for high-order direct Eulerian capability
- First-of-a-kind high-order ALE simulation results using novel non-linear mesh optimization plus high-order discontinuous Galerkin (DG) for ALE remesh/remap in large scale 3D rad-hydro simulations

0.3 Integration Points with ECP Software Technology projects

The LLNL ATDM code project represents a massive software development effort, incorporating multiple physics, mathematics and computer science packages into the overall integrated code. We collaborate with multiple software technology (ST) projects

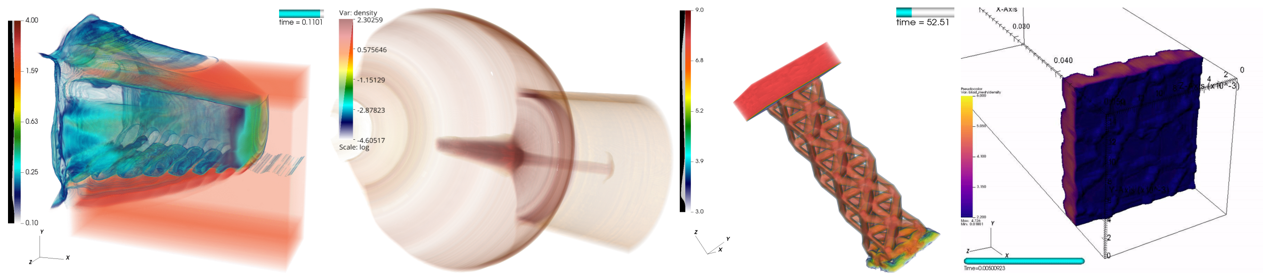


Figure 2: Example MARBL high-order 3D simulations: (*left-to-right*) radiation driven Kevin-Helmholtz instability experiment, BRL81a shaped charge, octet-truss high-velocity impact, radiation driven high-density Carbon micro-structure shock experiment.

to integrate these production quality capabilities, including software developed internally at LLNL, externally from the ECP and the broader open source community. To facilitate this, much time and effort has been invested in developing a robust build, test and documentation system for the MARBL project, including preliminary use of the Spack package manager. In addition, we employ extensive use of continuous integration and nightly regression testing across multiple platforms using multiple compilers to foster a culture of rigorously tested software developer contributions from a multi-disciplinary, geographically distributed team. As a result of this initial investment in software infrastructure and our focus on maintaining a collaborative culture, we have been able to successfully integrate multiple packages / capabilities into the code including:

- RAJA performance portability abstraction and Umpire host/device memory management
  - RAJA and Umpire also integrated with MFEM v4.0 as well for high-performance HO discretization kernels
- Ascent library integrated via Conduit and Axom datastore
  - In-situ visualization, volume rendering and radiography
- HDF binary parallel checkpoint/restart enabled via Conduit and Axom datastore
  - Data compression, burst buffer support
- Jupyter notebook interface prototype enabled using Conduit and Axom datastore
  - Simulation data access using lua/python
- Performance monitoring with Caliper and SPOT tools

The project relies heavily on the MFEM and Hypre libraries for high-order finite element capabilities as well as inter/intra node parallel performance, including:

- Parallel mesh abstraction
  - Non-conforming mesh refinement, adaptive mesh refinement
  - NURBS / high-order mesh geometry
- Parallel "GridFunction" field abstraction for scalar, vector, and tensor fields
- Local finite element, dense matrix operations
- Parallel matrix assembly and parallel sparse linear solvers
  - Bilinear forms for various physics
  - Continuous Galerkin (CG) and discontinuous Galerkin (DG) advection

  - Scalar / vector diffusion operators
  - Custom integrators (e.g. hydrodynamic force)
- Mesh optimization methods
  - Non-linear mesh optimization using the Target Matrix Optimization Paradigm (TMOP)

We have focused heavily on achieving high-performance on GPU platforms using a combination of software abstractions and efficient "matrix-free" algorithms in close collaboration with the Center for Efficient Exascale Discretization (CEED) Co-design center including:

- CEED bake-off problems targeting performance improvements on key kernels
- Full assembly being replaced with partial assembly where possible
- Optimal, partial assembly methods being developed by CEED
- Small, dense linear solves for DG Lagrange/Remap
- Linear algebra operations handled by MFEM
- Optimized kernels being developed by MFEM/CEED teams

The multiple C++ based packages in MARBL use RAJA for parallel loop execution, while the high-order Eulerian Fortran based package uses OpenMP. MARBL uses Umpire for host/device memory management, including memory pools shared between packages. The new MFEM 4.0 GPU abstraction developed under CEED was co-designed with our project to enable interoperability with RAJA and Umpire for host/device execution and memory management. This co-design process was essential for achieving GPU performance in the integrated MARBL code.

0.4 ATDM 2020 L1 Milestone Metrics

The ATDM program has requested from each lab a set of metrics for measuring progress at the five year mark according to four focus areas:

1. Mission Impact: *Define metrics that will express the status of both new and existing codes in terms of high interest to the end users. Areas to consider in the assessment include: problem setup times, code robustness, turn-around time, storage requirements, post-processing capabilities, visualization capabilities, degree of verification and validation, solution accuracy, workflow, etc ...*
2. Portability: *Evaluate code portability by running the same application's source code on different platforms and assessing the results. Identify the relative amount of source code that is platform- agnostic versus the source code that is unique to a specific platform.*
3. Developer Productivity: *Document and describe the ease of improving, modifying, or extending the codes, and training of staff. Some aspects to consider include: relative ease of implementing new models/methods/algorithms into the codes; improvements in software quality; maintenance and reuse of code in existing production capabilities; knowledge transfer, education, and training of developers.*
4. Code Performance: *Define and describe code capability in terms of computational performance. Measures to consider include: time-to-solution, percentage of peak FLOPS, strong and weak scaling, efficiency using the computational platform, etc. Measures should compare realized performance gains to reasonably expected gains, using CTS1 Broadwell processors as the baseline.*

This document contains details which we believe address all of the requested metrics described above. Below we summarize our overall response to these high level metric definitions, the details of which are contained in this document and will be presented at the final L1 review scheduled for December of 2020.

Mission Impact

For MAPP, mission impact has been the most important metric for success, with user adoption being our main goal. Along with the other NNSA labs, we have been targeting an FY20 demonstration of a 3D multi-physics problem of interest to the stockpile stewardship program at LLNL on ¼ of the Sierra machine as well as a performance portability demonstration by running a similar problem on the Sandia National Lab (SNL) Astra (ARM based) platform. The calculation will be multi-physics in nature, high-resolution and will utilize multiple new algorithms and novel capabilities developed as part of this work. Beyond this, we will demonstrate examples of early adoption where users have successfully applied our unique code capabilities to challenging problems.

- We will compare and contrast current user capabilities with next-gen capabilities by incorporating a user story based on successful, early adoption of the new code
- Our large-scale challenge problem will be directly relevant to the mission and will utilize several unique features which bring a simulation advantage for code users

Portability

Multiphysics simulation codes are complex, take several years to reach a point of maturity where they can begin to have an impact, and typically have lives measured in decades. Because of this, it is imperative that the code is flexible and extensible enough to be sustained over multiple generations of hardware. To achieve this, we have invested in performance portability abstractions and embraced the tools and libraries developed as part of the broader HPC / ECP software ecosystem. On and off node parallelism is sufficiently abstracted to enable physics algorithm developers to write single loop bodies which can be targeted to different hardware back-ends (at run time) including GPUs from multiple vendors and threaded multi-core CPUs.

- We will document our code portability strategy / process and demonstrate a single code base on three platforms (CTS1, Sierra and Astra)
- We will highlight platform specific optimizations in low level kernels in our 3rd party libraries
- We will provide a rough percentage of MAPP's source code base that is platform agnostic

Developer Productivity

Developer productivity has been a principle focus from the outset of this effort. We have devoted substantial project resources to our build, test and user/developer documentation system. To improve developer productivity, we have placed a high priority on rapid code builds, including the ability to link-in pre-built second and third party libraries which are generated automatically via continuous integration. This project has benefited from external developers of second and third party software libraries. In many cases, the developers of these libraries have directly participated in MAPP code development. To achieve this, we seek to constantly refine our on-boarding process to make it easier for new developers to join and begin contributing.

- We will provide a case study where a new developer has documented their experience working with the new code and two existing codes
- We will document our software development and integration strategy, on-boarding process and testing process
- We will highlight several successful software technology integrations with the new code
- We will document our successful leveraging of IC developments in 3rd party physics libraries on GPUs

Code Performance

We have focused primarily on mission impact in the early years of the project. Code performance was not a targeted effort until this past year. However, during this time, we have managed to achieve considerable performance improvements on both CPU and GPU architectures which we have steadily tracked over time. We are now seeing greater than 15X GPU node vs CPU node speedups and we are able to scale out to many thousands of MPI ranks.

- We will provide standard node-to-node performance comparisons on the same problem (e.g. 1 node CTS vs 1 node Astra vs 1 node Sierra) in both absolute performance and performance weighted by energy efficiency
- We will present CPU/GPU throughput curves
- We will present strong and weak scaling curves on multiple platforms

0.5 Technical Risks and Future Prospects

This project embraced several new technologies that presented risks which needed to be carefully managed. The biggest technical risk associated with the project is the substantial adoption of advanced numerical algorithms including high-order finite elements for ALE radiation-magneto-hydrodynamics and high order implicit-explicit (IMEX) methods for multi-physics coupling. While such methods have been successfully used in other industries, this project represents the first time they have been fully embraced for the NNSA simulation mission. We already have evidence that these methods are effective in this regard; however, further study will be required as new physics capabilities are added. In addition, the broad adoption of modular capabilities, including computer science infrastructure, is new to LLNL and requires careful coordination across multi-disciplinary teams and multiple modes of communication. The technical risks associated with MAPP, the mitigation strategies we put in place to address these and the expected retirement date for these risks is outlined in Table 1.

After the FY20 tri-lab milestone is complete, the team will begin transitioning to a user focused production code. In addition to adding more physics capability which is required to achieve parity with today's ASC integrated codes, additional efforts will focus on porting more physics capabilities to GPUs in preparation for the FY23 arrival of the exascale class El Capitan system. In parallel with this work, we will continue to push adoption of the code by the user community by applying it to new and more challenging simulation scenarios. Additional work will include:

- Initial v1.0 user release in CY20
- Continue to broaden application space, engaging with user community and working closely with early adopters
- Enable optimization / learning framework for ensemble runs
- Continued improvements to physics fidelity including deterministic and implicit Monte Carlo thermal radiation transport, Lagrangian auto-contact and RANS based mix models
- Advanced matrix-free solvers for diffusion and other implicit physics
- Complete GPU porting of all physics packages and El Capitan readiness by FY23

Major Technical Risk	**Mitigation Strategy**	**Anticipated Retirement Date**
• Radiation transport on high-order meshes is a research area • High-order coupling of multi-physics is a research area	• Blending of high-risk methods with more traditional low-order schemes using HO/LOR coupling • Arbitrary order capability in code enables running in low-order mode as an ultimate fall back position	• Evidence already available that such methods are viable • Full risk retirement only after more physics is complete and tested by FY21
• Complete separation of core CS components into a reusable toolkit is a new model for LLNL ASC code development	• Clearly defined APIs, design discussions between parties • Maintain communication and collaboration via wiki, email, project meetings, etc ...	• Evidence that this risk has largely been retired as of FY19
• Immaturity of vendor-supported programming models and compiler technology can slow progress	• Use of RAJA and Umpire abstractions to isolate parallel loop execution and host/device memory /pool management behind single API	• FY20 demonstration of performance on multiple platforms • Evidence suggests this risk is largely retired
• Arbitrary order selection at runtime can be expensive; important trade-offs in flexibility for user vs. compile-time optimizations and higher performance	• Focus on key orders for optimization (3D quadratic) using templates • Explore just-in-time compilation to exploit compiler optimizations given known loop bounds	• FY20 for commonly used discretization orders (quadratic elements in 3D) • FY22 for general, arbitrary orders
• Algebraic multigrid solvers are presently not GPU performant	• Consider vendor specific options while engaging with tri-lab in improving general linear solver libraries • Continue matrix-free preconditioner research and development	• FY22 for integration of GPU enabled matrix-free multigrid solvers

Table 1: MAPP technical risks and mitigation strategies

Chapter 1

Introduction

1.1 The LLNL ATDM "Next Generation" Code Project

LLNL is developing next-generation multi-physics simulation capabilities for national security applications under the Multi-physics on Advanced Platforms Project (MAPP). MAPP is part of the overall tri-lab ATDM NextGen code development effort taking place at LLNL, LANL and SNL to meet the challenge posed by exascale computing. Prior to the official start of this work in 2015, LLNL performed an extensive, year-long requirements gathering process resulting in multiple reports outlining the physics requirements, user workflow desires and computer science recommendations for a Next Generation code [1]. In addition, the LLNL effort was heavily influenced by the lessons learned from the ASCI program, launched in 1996, which brought forth the massively parallel simulation era and today's petascale class production codes [2]. Notable lessons learned include:

- Build on successful code development history and prototypes
- Risk identification, management and mitigation
- Determine the schedule and resources from the requirements
- Customer focus
- Use of modern but proven computer science techniques
- Better physics and computational mathematics is more important than better computer science

Developing a multiphysics code capable of meeting the various simulation needs of the NNSA as defined by the current generation of integrated codes (ICs) and able to scale to the current 100 petaflop class pre-exascale systems, as well the forthcoming exaflop class computers (see Figure 1.1), is a daunting challenge. To accomplish this ambitious and technically challenging goal, we have embraced two key themes: use of high-order numerical methods and a modular approach to code development. A foundational component of MAPP is the Axom computer science (CS) toolkit which provides infrastructure for the development of modular, performance portable, multi-physics application codes. The core capabilities provided by Axom are proven pieces of infrastructure that have been found to be common to all of our previous production codes; and in some cases can comprise up to 40% of a project's source code. In addition, the performance portability abstractions we have employed were also developed for the current generation of LLNL production codes which have undergone substantial refactoring to prepare for GPU based platforms like Sierra. Specifically, RAJA and Umpire enable performance portability and heterogeneous memory management. RAJA is an open source set of C++ abstractions (requiring only standard C++11 language features) for writing single-source, portable loop kernels which supports simple loops, complex kernels, and kernel transformations while providing portable reductions, scans, atomics, data views and thread-local shared memory. RAJA supports multiple back-ends including sequential, SIMD, OpenMP

(CPU, target), CUDA and AMD HIP. Umpire is an open source API for managing heterogeneous memory resources including memory ops for integrated applications (allocate, deallocate, copy, move, query etc . . .). Both of these libraries have been fully integrated and battle hardened in the current generation of LLNL production codes. In addition, novel capabilities of the NextGen code have motivated additional features in RAJA which have been upstreamed into the production version through a co-design process.

SIERRA
NNSA
ASTRA
U.S. DEPARTMENT OF ENERGY
NNSA
Lawrence Livermore National Laboratory
CRAY
EL CAPITAN

Figure 1.1: The NVidia GPU based 120 PF Sierra system (*left*), the ARM based Astra system at SNL (*middle*) and the forthcoming AMD GPU based El Capitan exascale system (*right*).

MARBL is a next-generation application code built on the Axom base to address the modeling needs of the high energy density physics (HEDP) community for simulating high-explosive, magnetic or laser driven experiments such as inertial confinement fusion (ICF), pulsed-power magneto-hydrodynamics (MHD), equation of state (EOS) and material strength studies as part of the NNSA's stockpile stewardship program (SSP). MARBL is designed from inception to support multiple diverse algorithms, including Arbitrary Lagrangian-Eulerian (ALE) and direct Eulerian methods for solving the conservation laws associated with its various physics packages. A distinguishing feature of MARBL is the use of advanced, high-order numerical discretizations such as high-order finite element ALE and high-order finite difference Eulerian methods. This algorithmic diversity encompasses the ECP simulation motifs of unstructured and structured adaptive mesh refinement (AMR). High-order numerical methods were chosen because they have higher resolution/accuracy per unknown compared to standard low-order finite volume schemes and because they have computational characteristics which play to the strengths of current and emerging high-performance computing (HPC) architectures. Specifically, they have higher FLOP/byte ratios meaning that more floating-point operations are performed for each piece of data retrieved from memory. This leads to improved strong parallel scalability, better throughput on GPU platforms and increased computational efficiency. To achieve the necessary meshing flexibility, high-order discretization capabilities and high performance including both on and off node parallel scalability, we make extensive use of the modular, open source finite element library MFEM as well as the scalable linear solvers library, Hypre.

A key goal for MARBL is enhanced end-user productivity including improved workflow for problem setup and meshing, simulation robustness, support for UQ and optimization driven ensembles, and in-situ data visualization and analysis. High-order ALE and Eulerian schemes have proven to be more robust and should significantly improve the overall analysis workflow for users. The advanced simulation capabilities provided by MARBL will improve user throughput along two axes: faster turnaround for multi-physics simulations on advanced architectures and less manual user intervention. MARBL's unique high-order ALE hydrodynamics capability on unstructured NURBS meshes, enabled by the MFEM discretization library and our mesh generation tool PMesh, leads to improved user workflow. An example is shown in Figure 1.2 where users supply geometry definitions to the PMesh tool which performs automatic "paved" mesh generation. In collaboration with this project, PMesh has been enhanced to write out unstructured meshes in a non-uniform rational B-spline (NURBS) format compatible with the MFEM library. This enables high-order, curved geometry at initial times. Using MFEM capabilities, MARBL is able to read in this mesh in serial, perform local conforming and non-conforming geometry preserving refinements in both serial and parallel, and then robustly run multi-material physics calculations on these high-order, unstructured meshes. This capability is unique to MARBL and represents a powerful capability for enabling rapid problem setup for users.

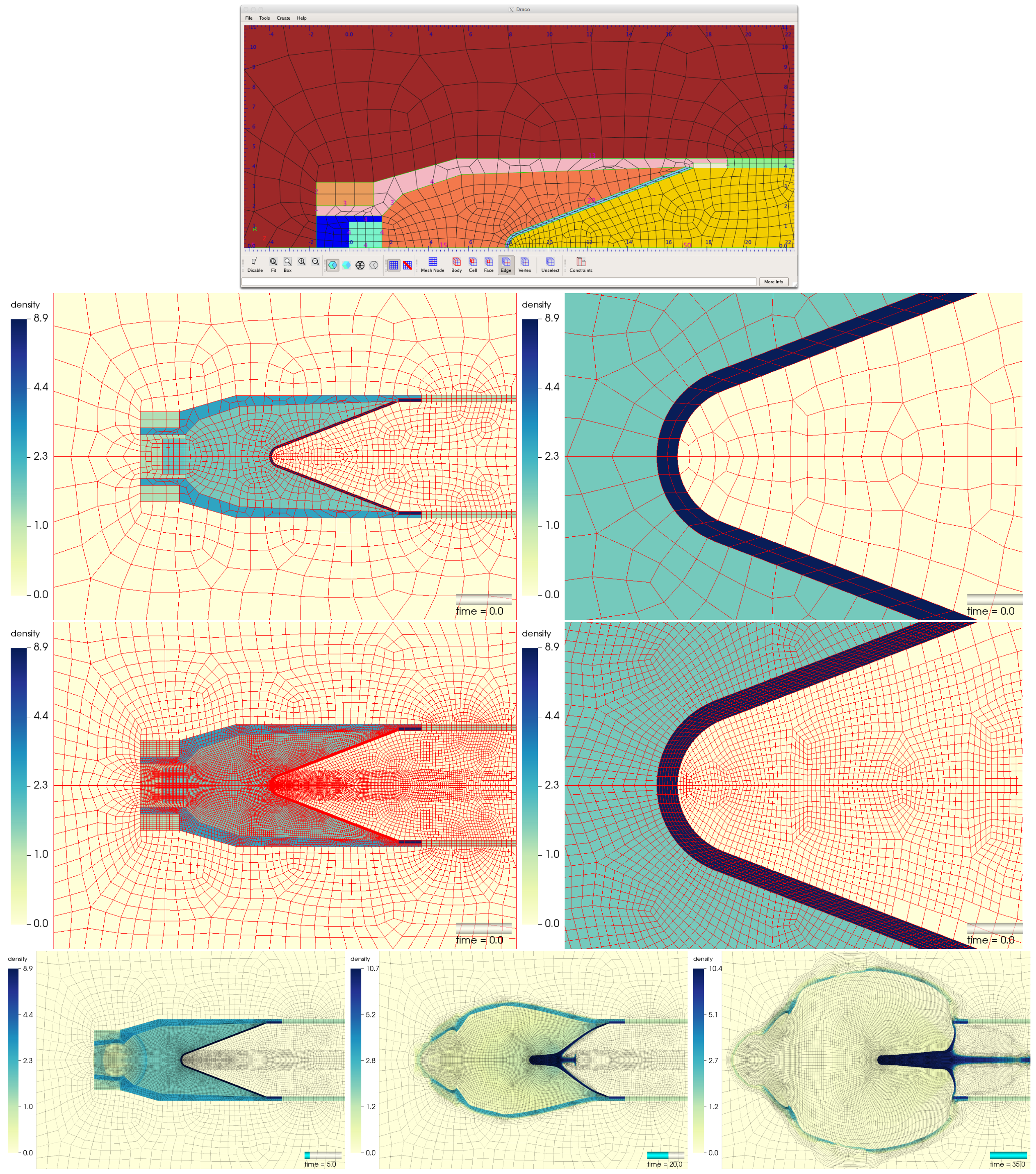


Figure 1.2: Example of MARBL high-order, unstructured mesh workflow. Users supply geometry, PMesh performs automatic “paved” mesh generation (*top-row*). PMesh saves output in MFEM NURBS format (*second-row*). From MARBL, users can optionally apply both uniform or non-conforming, geometry preserving refinement in serial or parallel via MFEM (*third-row*). MARBL can run robustly on high-order, non-conforming meshes using high-order numerical algorithms (*bottom-row*).

1.2 Key Project Themes

To meet the daunting challenge of multiphysics code development at the exascale, we have embraced key themes to help organize our project which build upon the lessons learned from other successful code development efforts throughout our history.

As already discussed, the adoption of high-order numerical techniques posed a risk for this project. While there is extensive evidence to support the numerical and computational benefits of such methods, a key obstacle for deploying these in an integrated application is the general immaturity of the supporting software ecosystem for dealing with high order. In addition to physics and material model capabilities, this includes the many ancillary libraries and supporting infrastructure for field and mesh descriptions, mesh generation, post processing, visualization and data extraction. In essence, we are a high order code in a generally low order HPC landscape. To account for this, we have taken a multi-tiered approach to the adoption and integration of high-order technologies throughout our integrated software stack. Our taxonomy for acquiring new high-order capabilities into our code consists of four categories, each with different advantages and risk profiles:

- **“0D” Operations**: For the class of computational libraries whose outputs are independent of spatial gradients of the simulation mesh (i.e. do not require knowledge of neighboring points in a mesh) and depend only on local position values or phase space input, we can readily incorporate these directly into HO applications by invoking them at quadrature points (instead of zones as is typically done). Examples of this include tabular equations of state (EOS), material strength, multi-group opacities, sample based geometric shaping, inside/outside queries, and TN burn.
- **Low-Order Refined (LOR) Compatibility**: For mesh based operations where the underlying library has no concept of a high-order mesh, we can project the HO solution onto a low-order refined (LOR) mesh (a subdivision of each high-order element into a nested set of standard, straight edged, low-order elements) and call libraries/utilities as normal. This process can lead to a geometric error if the parent high-order element is substantially curved and the low-order refined counterpart does not sufficiently capture this curvature; but this can be mitigated by increasing the number of subdivisions of the LOR mesh. This capability is provided through the MFEM discretization library (see Section 4.2.4 for details). Operations like this are typically used in post-processing data like visualization or a one time transfer of a HO simulation state to another physics library/package (e.g. transferring from a high-order Lagrange hydro mesh to a direct Eulerian hydro mesh). This is how the MAPP project leverages all of the visualization capabilities of the Visit library for high order fields and meshes, even though Visit has no internal concept of high-order.
- **LOR to High Order (HO) Reconstruction**: For mesh to mesh coupling with a low-order library, where we need to incorporate results back into the HO simulation, we utilize a novel high-order reconstruction technique that is provided by the MFEM discretization library and described in detail in Section 4.2.4. Example applications of this include coupling contact mechanics or thermal radiative transfer libraries with a high-order hydrodynamics mesh.
- **Native High-Order**: The most desirable solution in the long term is to have libraries/utilities capable of operating natively on high-order meshes/fields; however, in many cases this requires on-going research and development into new techniques. Some of these new techniques are production ready now, including point location queries for a high order field on a general high order mesh (for tracer particles) and native high-order visualization (provided by the Ascent library). Other capabilities are still in the R+D phase such as deterministic thermal radiative transfer and contact mechanics.

Another key theme for MAPP has been adoption of modularity throughout the project. This is true for our physics capabilities, where we have designed the code to treat all physics (including hydrodynamics) as modular from inception by developing an abstract “main” program (see Section 3.3 for details). This also includes modularity in terms of the mathematical libraries we employ and our computer science infrastructure. Unlike other simulation codes which tend to own all aspects of the computational mesh and spatial discretization, we have adopted software libraries (MFEM and Hypre) for handling the bulk of our mesh/field operations. In addition, for computer science needs, we have taken our centralized in-memory data store, computational geometry operations and performance portability abstractions and placed these in external libraries (Axom and RAJA).

This modular approach to software has many strengths. For example, complex mesh based operations like non-conforming refinement (e.g. adaptive mesh refinement) are readily incorporated as updates to libraries become available. Likewise, as new HPC hardware becomes available, modular performance portability abstractions like RAJA enable the use of these new devices by our integrated code as well as several others with library updates. However, these benefits are not without their costs. One trade-off is that updates are not a simple process which can be done on short notice (which is often needed in a production environment); but instead require careful communication, coordination and testing across the multiple projects involved. This increased complexity and potentially reduced flexibility is one of the biggest risks of modularity. However, we ultimately believe that modularity of this sort is the right approach for achieving production quality HPC code in the exascale computing era. Maintaining constant and multiple paths of communication and coordination is essential to realize the many benefits of modularity. Most importantly, it is imperative that the individual libraries view their success as dependent on the success of the integrated applications.

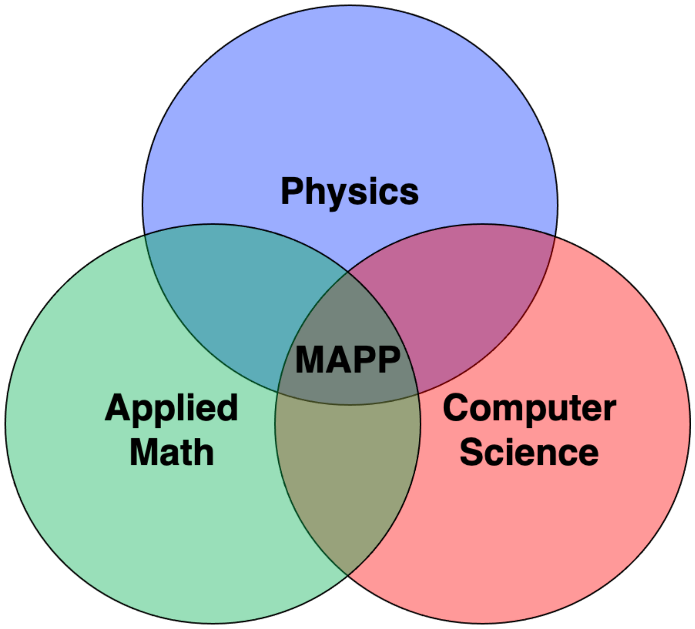


Figure 1.3: MAPP is a careful balance between the scientific disciplines of physics, applied mathematics and computer science; requiring team coordination across different technical backgrounds, software languages/abstractions and geographic locations.

Chapter 2

Physics

2.1 Introduction

MARBL is a multi-physics code for simulating high energy density physics (HEDP) and focused experiments driven by high-explosive, magnetic or laser based energy sources. From a software perspective, MARBL is designed to facilitate modular physics packages which are integrated in a single code for performing a simulation. For HEDP, this includes multi-material, compressible flow for all phases of matter (from solids to plasmas), three temperature radiation hydrodynamics (separate temperatures for ions, electrons and radiation), magneto-hydrodynamics using a resistive approximation for magnetic diffusion, thermonuclear burn for high-temperature plasmas in the fusion regime and various ”0D” models which define material properties like equations of sate (EOS), opacities, thermal and electrical conductivities, thermonuclear reaction rates, etc . . . .

In addition to modularity for the physics capabilities, we are also interested in modularity in terms of numerical algorithms. A principle goal of this project has been to support multiple, diverse algorithms for solving conservation laws / physics models. This is needed to provide a so called “internal second vote” capability; a mechanism for ensuring that we always have multiple ways of solving a problem to provide a crucial check on applications. MARBL is fundamentally a high-order finite element, unstructured mesh based code, but we want the ability to incorporate alternative approaches which can leverage the modular physics and CS infrastructure of the project. This includes direct Eulerian hydrodynamics which operates on a purely Cartesian mesh and eventually, the incorporation of particle based hydrodynamics (SPH) methods. In addition, we have invested heavily in CS infrastructure for managing the transfer of data between different mesh types to enable physics packages to operate on their own internal mesh data structures while being able to exchange field data to and from the high-order unstructured mesh.

In this section, we review the current set of physics capabilities available in MARBL. At present, this includes:

- Multi-material hydrodynamics for all-speed flow
  - Arbitrary-Lagrangian-Eulerian (ALE) unstructured mesh capability based on high-order finite elements
  - Alternative direct Eulerian option based on high-order finite difference methods
- 3T radiation-hydrodynamics
- Thermal conduction for electrons and ions
- Thermonuclear burn for fusion plasmas
- Magneto-hydrodynamics
- Constitutive “0D” material models
  - Material strength models

2.2 High-order finite element multi-material ALE hydrodynamics

For modeling high-speed compressible, multi-material flows over complex geometries, the national labs have historically relied on arbitrary Lagrangian-Eulerian (ALE) methods. The benefits of ALE methods can be found in [3, 4, 5]. The unstructured mesh ALE capability of MARBL is based on a general framework for high-order Lagrangian discretizations of the Euler equations of compressible shock hydrodynamics on general 1D, 2D and 3D spatial domains described in a series of papers [6, 7, 8, 9, 10, 11, 12, 13, 14, 15] and implemented in the code Blast [16]. The equations are discretized and solved on a generally unstructured computational mesh that moves with the fluid velocity, v, during the so called Lagrange phase of ALE. To simplify the discussion, we consider the single material form of the Euler equations in a moving frame (but emphasize that the method we describe is valid for the general multi-material case as discussed later):

$$\begin{aligned}
\text{Momentum Conservation: } \rho\frac{dv}{dt} &= \nabla\cdot\sigma, && (2.1)\\
\text{Mass Conservation: } \frac{1}{\rho}\frac{d\rho}{dt} &= -\nabla\cdot v, && (2.2)\\
\text{Energy Conservation: } \rho\frac{de}{dt} &= \sigma:\nabla v, && (2.3)\\
\text{Equation of Motion: } \frac{dx}{dt} &= v, && (2.4)
\end{aligned}$$

which involves the "mesh" derivative $\frac{d}{dt}$:

$$\frac{d\alpha}{dt} \equiv \frac{\partial\alpha}{\partial t} + v\cdot\nabla\alpha,$$

the kinematic variables for the fluid velocity vector v and position vector x, and the thermodynamic variables for the density ρ, pressure $p = EOS(\rho,\, e)$ and specific internal energy e of the fluid [17, 3]. The equation of state (EOS) is a material constitutive relation which is typically provided by a third party library which uses tabulated experimental data to interpolate values based on phase space input or some form of analytic model, the simplest case of which is the ideal gas model with a constant adiabatic index $\gamma > 1$ which has the form $p = (\gamma-1)\rho\, e$.

The total stress tensor, σ, is the sum of both physical stresses and the stress due to artificial viscosity. For the simplest case of gas dynamics, the total stress is given by

$$\sigma = -pI + \sigma_a + \dots, \tag{2.5}$$

where p is the scalar pressure, I is the identity tensor and σ_a is the artificial stress tensor. In general, we assume the artificial stress σ_a is a full rank 2 tensor. For more complex physics (e.g. elastic-plastic-flow, magnetic stress, radiation pressure, etc ...), we simply add additional stress terms to (2.5) without the need to modify σ_a. This is an attractive feature of artificial viscosity techniques in contrast with Riemann solver based methods which often require modifications based on the underlying types of stresses being modeled.

MARBL uses arbitrary order finite elements to solve the three distinct phases of ALE hydrodynamics: the Lagrange phase where the conservation laws are solved in a moving frame, the remesh phase where the computational mesh is optimized according to some metric and finally the remap phase where fields are conservatively, accurately and monotonically mapped from the Lagrangian mesh to the newly optimized mesh. These phases are depicted in Figure 2.1. While all ALE methods of this type can be run in an Eulerian like mode where the remesh/remap phases simply transfer the Lagrangian solution back to the initial mesh, it is important to emphasize that ALE methods are fundamentally solved in the Lagrangian frame and this is algorithmically distinct from direct Eulerian methods which solve the conservation laws in a fixed frame of reference (See Section 2.3)

The high order ALE method in MARBL is valid for any finite dimensional approximation of the kinematic and thermodynamic fields, including generic finite elements on two- and three-dimensional meshes with triangular, quadrilateral, tetrahedral, or hexahedral zones. Our method can be viewed as the high-order generalization of staggered-grid hydrodynamics (SGH) which has proven to be the numerical method of choice for the various ALE codes at our national labs. We discretize the kinematic variables of position and velocity using a continuous high-order basis function expansion of arbitrary polynomial degree which is obtained via a corresponding high-order parametric mapping from a standard reference element. This enables the use of curvilinear zone

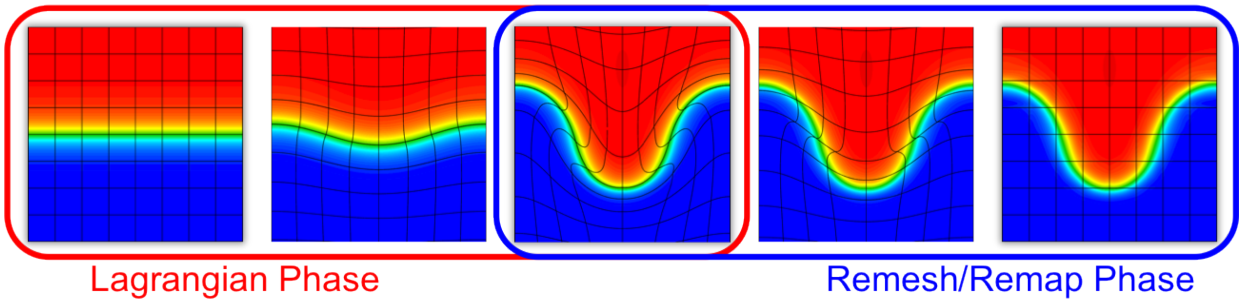


Figure 2.1: Schematic depiction of the three phases of the high-order ALE process: the Lagrange phase where the physics conservation laws are evolved in physical time on a high-order, curvilinear mesh which moves with the fluid, and the remesh/remap phases where the Lagrangian mesh is optimized and the field variables are conservatively and monotonically remapped to the new mesh in pseudo-time. (*right*)
.

geometry, higher-order approximations for fields within a zone, and a pointwise definition of mass conservation which we refer to as strong mass conservation. Element (or zone) curvature can arise naturally as a result of the Lagrangian fluid motion or due to initial geometric configurations; in other words, MARBL can support high-order curved element meshes for initial conditions, including NURBS based meshes [18].

We discretize the internal energy using a piecewise discontinuous high-order basis function expansion which is also of arbitrary polynomial degree. This facilitates multimaterial hydrodynamics by treating material properties, such as equations of state and constitutive models, as piecewise discontinuous functions which vary within a zone. To satisfy the Rankine–Hugoniot jump conditions at a shock boundary and generate the appropriate entropy, we introduce a general tensor artificial viscosity which takes advantage of the high-order kinematic and thermodynamic information available in each zone. To achieve high-order accuracy away from shocks, we employ a hyperviscosity technique which blends a first order mesh based dissipative term for robust shock capturing with a high-order term in smooth regions. Finally, we apply a generic high-order time discretization process to the semidiscrete equations to develop the fully discrete numerical algorithm. An illustrative example of the features of high-order Lagrangian hydrodynamics is shown in Figure 2.2.

2.2.1 High-order Lagrange phase

In this section we review the approach originally described in [9] which is based on a generic, finite dimensional, weak variational formulation of the Euler equations in a Lagrangian frame. The approach is valid for a wide variety of discrete functions and mesh types; however for simplicity, in this section, we limit the discussion to high-order finite elements defined on 2D quadrilateral meshes.

Kinematic and Thermodynamic Basis Functions

The high-order finite element Lagrangian formalism of MARBL begins with two discrete *function spaces* on the initial spatial domain $\Omega(0)$:

- A continuous, vector valued, *kinematic* space $\mathcal{V} \subset [H^1(\Omega(0))]^d$, with a basis $\{w_i\}_{i=1}^{N_\mathcal{V}}$,
- A discontinuous, scalar valued, *thermodynamic* space $\mathcal{E} \subset L_2(\Omega(0))$, with a basis $\{\phi_i\}_{j=1}^{N_\mathcal{E}}$.

This generic combination of both continuous Galerkin (CG) and discontinuous Galerkin (DG) function spaces is the high-order generalization of the "staggered grid" spatial discretization technique commonly employed in ALE codes. Indeed, the continuous

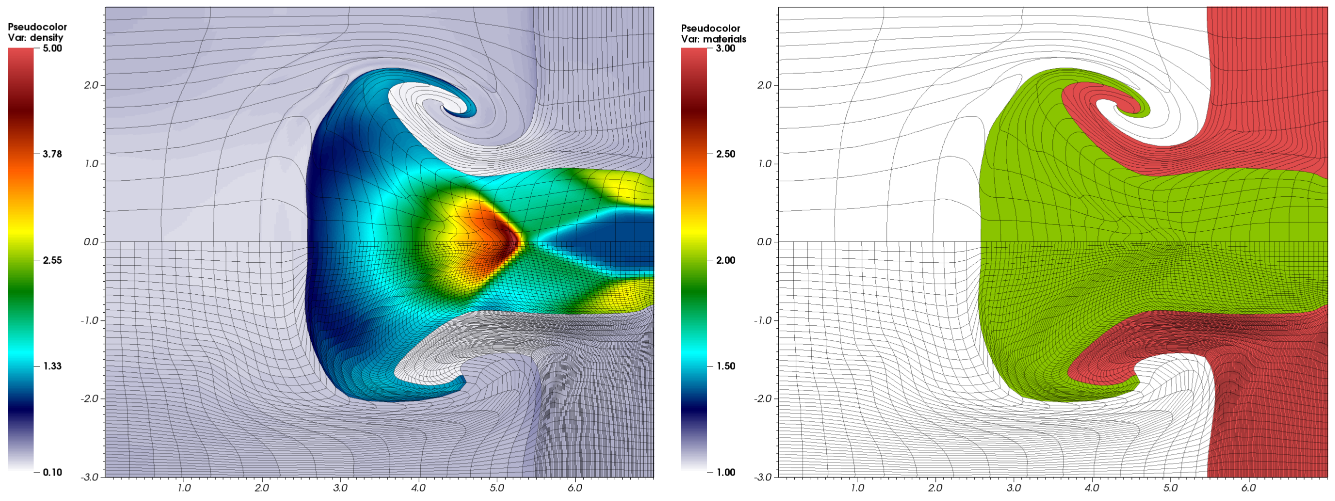


Figure 2.2: In contrast to traditional low order staggered grid discretizations of Lagrangian hydrodynamics (*bottom-half*) , high-order finite element Lagrangian hydrodynamics (*top-half*) has several unique features including extreme mesh curvature, sub-cell resolution and better resolved vortical motion of the materials for the same number of degrees of freedom
.

kinematic space is the generalization of so called "nodal" fields while the discontinuous thermodynamic space is the generalization of "zonal" fields. For the specific case of 2D finite elements defined on quadrilateral meshes, the basis functions for the kinematic and thermodynamic spaces can be constructed by tensor products of 1D interpolating polynomials defined over the unit interval

$$\hat{\eta}_{ij}(\hat{x},\hat{y}) = \hat{\eta}_i(\hat{x})\eta_j(\hat{y}),$$

which is illustrated in Figure 2.3. The interpolation points for the 1D basis functions $\hat{\eta}_i(\hat{x})$ are arbitrary, but in practice we use both Gauss-Lobatto and Gauss-Legendre points.

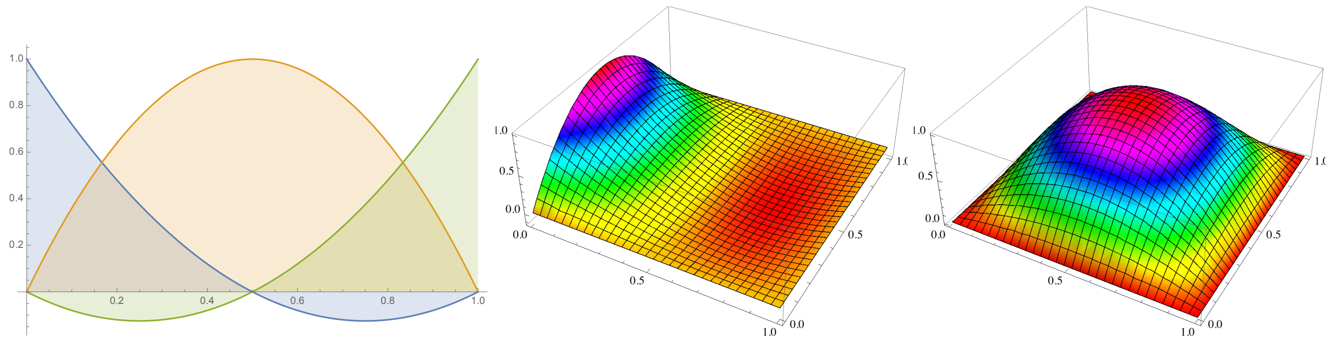


Figure 2.3: 1D Q2 Gauss-Lobatto basis functions, $\hat{\eta}_i(\hat{x})$, defined over the unit interval (*left*) and 2D bi-quadratic versions, $\hat{\eta}_i(\hat{x},\hat{y})$ (*right*)

Position and Deformation

Finite element "zones" are defined through a parametric mapping defined with respect to a reference space configuration as

$$\Omega_z(t) \equiv \{\mathbf{x} = \hat{\Phi}_z(\hat{\mathbf{x}},t) \, : \, \hat{\mathbf{x}} \in \hat{\Omega}_z\} \tag{2.6}$$

where $\hat{\Omega}_z$ is the reference zone (or element), e.g. the unit square for quadrilateral meshes. All quantities defined with respect to this reference space configuration are accented with a "hat" symbol.

We discretize the Lagrangian position variable, $\mathbf{x}$, using Gauss-Lobatto (nodal) basis functions defined on the reference element $\hat{\eta}_i$ as in Figure 2.3. The reference space mapping for a given zone $\Omega_z(t)$ is then given by

$$\hat{\Phi}_z(\hat{\mathbf{x}},t) = \sum_i \mathbf{x}_{z,i}(t)\hat{\eta}_i(\hat{\mathbf{x}}) \tag{2.7}$$

where $\mathbf{x}_z(t)$ is vector of nodal position values associated with zone z, a subset of the vector $\mathbf{x}(t)$ of size $N_{\mathcal{V}}$ which contains all of the discrete nodal position values. An illustration of this mapping is shown in Figure 2.4.

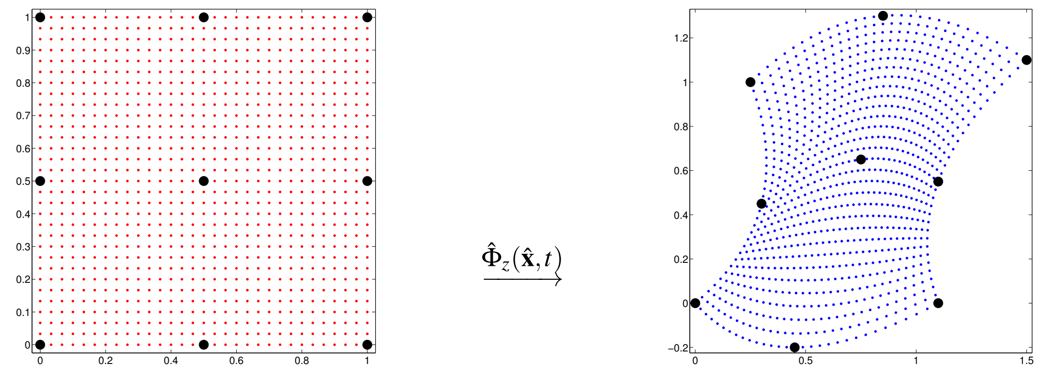


Figure 2.4: Example of reference space mapping for the case of a bi-quadratic basis. The black points represent the nodal position values, $\mathbf{x}_c(t)$. It is important to note that the mapping defines a function which varies over the entire zone, which is sampled at quadrature points

The kinematic space $\mathcal{V}(t)$ is used to discretize the Lagrangian position variable $\mathbf{x}$ as well as kinematic variables like velocity $\mathbf{v}$. With any discrete Lagrangian position $\mathbf{x}_p$ we associate d kinematic vector basis functions of the form

$$w_{p_1} = (\eta_p, 0), \qquad w_{p_2} = (0, \eta_p), \qquad \text{in 2D}.$$

The scalar function η_p is non-zero only in zones Ω_z that contain the point p. It is by definition continuous, which means it is single valued for any Lagrangian position value, including points that share the same physical location at shared cell boundaries. In contrast, the thermodynamic space is discontinuous to account for material interfaces and is therefore not necessarily single valued for any given Lagrangian position value. This is illustrated in Figure 2.5.

Since the kinematic and thermodynamic basis functions on the reference element are independent of time, the global basis functions are defined to move with the mesh and are therefore constant along all particle trajectories implying that

$$\frac{dw_i}{dt} = 0 \qquad \text{and} \qquad \frac{d\phi_j}{dt} = 0. \tag{2.8}$$

This means that the basis functions maintain there interpolation properties independent of the mesh configuration. This is illustrated in Figure 2.6.

Strong Mass Conservation Principle

The Jacobian matrix corresponding to the reference space mapping for a given zone z is

$$\hat{\mathbf{J}}_z = \frac{\partial \hat{\Phi}_z}{\partial \hat{\mathbf{x}}}(\hat{\mathbf{x}},t). \tag{2.9}$$

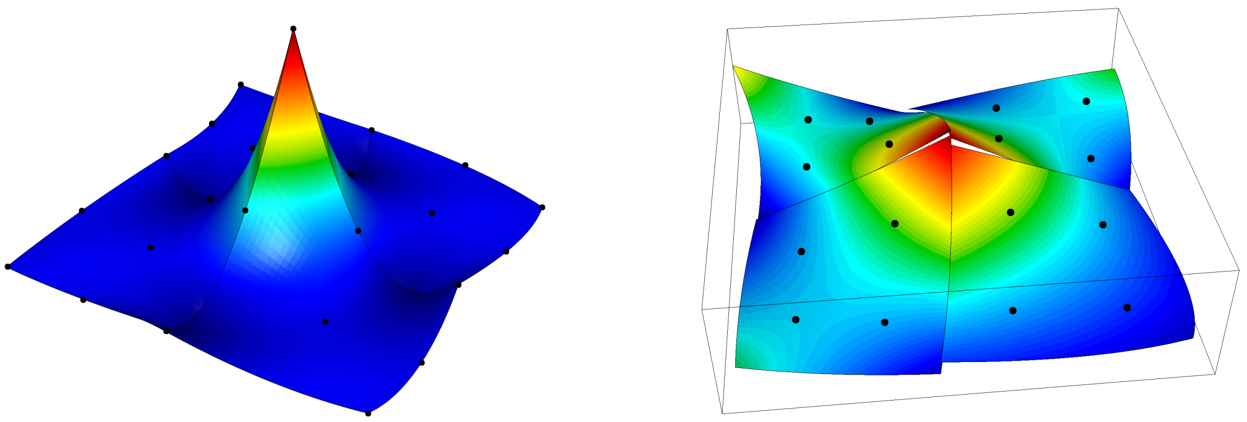


Figure 2.5: Examples of bi-quadratic (Q_2) continuous and bi-linear (Q_1) discontinuous finite element basis functions defined on a 4 element bi-quadratic (Q_2) quadrilateral mesh. Note that the continuous Q_2 function is single valued at all points (its derivative is not), including zone boundaries while the discontinuous function is multi-valued at zone boundaries. In both cases, the basis functions are obtained by mapping the reference space values via a Q_2 element transformation.

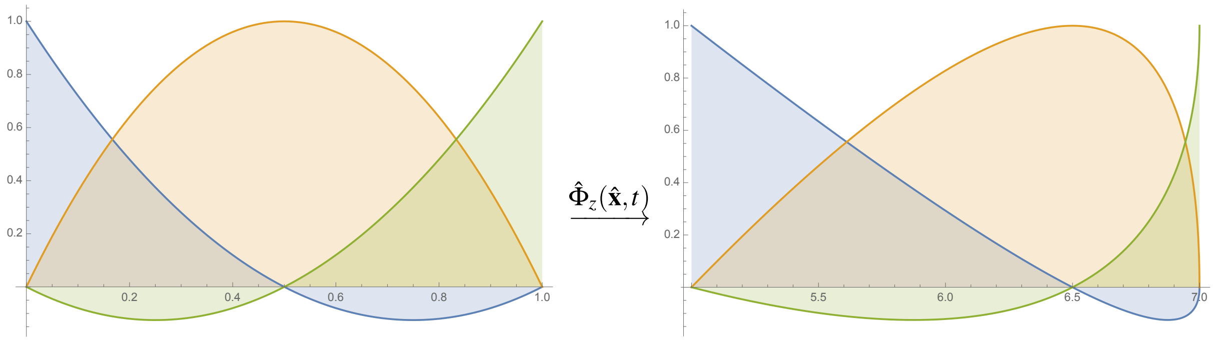


Figure 2.6: Illustration of the Lagrangian nature of the basis functions. A quadratic basis $\{\hat{w}\}$ defined on the 1D reference element $\hat{\mathbf{x}} = \{0, 0.5, 1\}$ (*left*) and the transformed basis $\{w\}$ on a deformed 1D element given by $\mathbf{x} = \{5, 5.7, 6\}$.

For bijective mappings (i.e. non twisted elements), it is a non singular function defined over each zone and its determinant, $|\mathbf{J}_z|$, can be viewed as a local (or point-wise) volume function since

$$|\Omega_z(t)| = \int_{\hat{\Omega}_z(t)} |\mathbf{J}_z|\,.$$

Furthermore, the total mass in a zone is

$$m_z \equiv \int_{\Omega_z(t)} \rho |\mathbf{J}_z|\,.$$

The Lagrangian perspective requires that

$$\frac{dm_z}{dt} = 0\,.$$

However, when dealing with high-order finite elements, both density and volume are now discrete *functions* which vary inside of a zone; so this is not a strong enough statement. We therefore utilize the "strong" (or point-wise) mass conservation principle which is derived by considering the limiting case of

$$\int_{\Omega'(t)} \rho(t) = \int_{\Omega'(0)} \rho(0) \quad \longrightarrow \quad \int_{\hat{\Omega}'} \hat{\rho}(t)|\mathbf{J}_z(t)| = \int_{\hat{\Omega}'} \hat{\rho}(0)|\mathbf{J}_z(0)|$$

for any $\Omega'(0) \subset \Omega(0)$. This leads to the point-wise equality:

$$\rho(t) = \frac{\rho(0)|\mathbf{J}_z(0)|}{|\mathbf{J}_z(t)|}. \tag{2.10}$$

We do not explicitly discretize (2.2); instead we compute the density via (2.10) which is the high-order generalization of the SGH concept of "mass conservation by fiat" [4].

Semi-discrete Motion

The discrete velocity field corresponding to the mesh motion is given by

$$\mathbf{v}(\mathbf{x},t) = \sum_i \frac{d\mathbf{x_i}}{dt}(t) w_i(\mathbf{x}) = \mathbf{v}(t)^{\mathrm{T}}\mathbf{w}(\mathbf{x})$$

Specifically,

$$\mathbf{v} = \frac{d\mathbf{x}}{dt}.$$

The high order mesh is therefore evolved in time by the velocity field $\mathbf{v}$, which can give rise to (additional) curvature of the mesh as defined by the high-order position vector $\mathbf{x}$.

Semi-discrete Momentum Conservation

We formulate a discrete momentum conservation equation by applying a variational formulation to the continuous equation (2.1). Using a Galerkin approach (at a given time t) we multiply (2.1) by a moving basis test function $w_j \in \mathscr{V}(t)$ and integrate over $w(t)$:

$$\int_{\Omega(t)} \rho \frac{dv}{dt} \cdot w_j = \int_{\Omega(t)} (\nabla \cdot \sigma) \cdot w_j. \tag{2.11}$$

Performing integration by parts on the right hand side, we obtain

$$\int_{\Omega(t)} \rho \frac{dv}{dt} \cdot w_j = -\int_{\Omega(t)} \sigma : \nabla w_j + \int_{\partial\Omega(t)} n \cdot \sigma \cdot w_j, \tag{2.12}$$

where n is the outward pointing unit normal vector of the surface $\partial\Omega(t)$. Assuming the boundary integral term vanishes (which is the case e.g. for boundary conditions $v \cdot n = 0$ and $\sigma = -pI$) and expanding the velocity in the moving basis gives us

$$\sum_i \frac{d\mathbf{v}_i}{dt} \int_{\Omega(t)} \rho\, w_i \cdot w_j = -\int_{\Omega(t)} \sigma : \nabla w_j. \tag{2.13}$$

In other words,

$$\mathbf{M}_{\mathscr{V}} \frac{d\mathbf{v}}{dt} = -\int_{\Omega(t)} \sigma : \nabla \mathbf{w}. \tag{2.14}$$

where $\mathbf{M}_{\mathscr{V}}$ is the *kinematic mass matrix* which is defined by the integral

$$\mathbf{M}_{\mathscr{V}} \equiv \int_{\Omega(t)} \rho \mathbf{w}\mathbf{w}^{\mathrm{T}}. \tag{2.15}$$

The kinematic mass matrix is sparse and computed over the entire mesh $\Omega(t)$ from the contributions of the degrees of freedom associated with each individual zone z as $\mathbf{M}_{\mathscr{V}} = \mathit{Assemble}(\mathbf{M}_{\mathscr{V},z})$. The process of global assembly is analogous to the concept of "nodal accumulation" that is used in a traditional SGH method where a quantity at a node is defined to be the sum of contributions from all of the zones which share this node. The difference here is that shared degrees of freedom may occur not only at nodes,

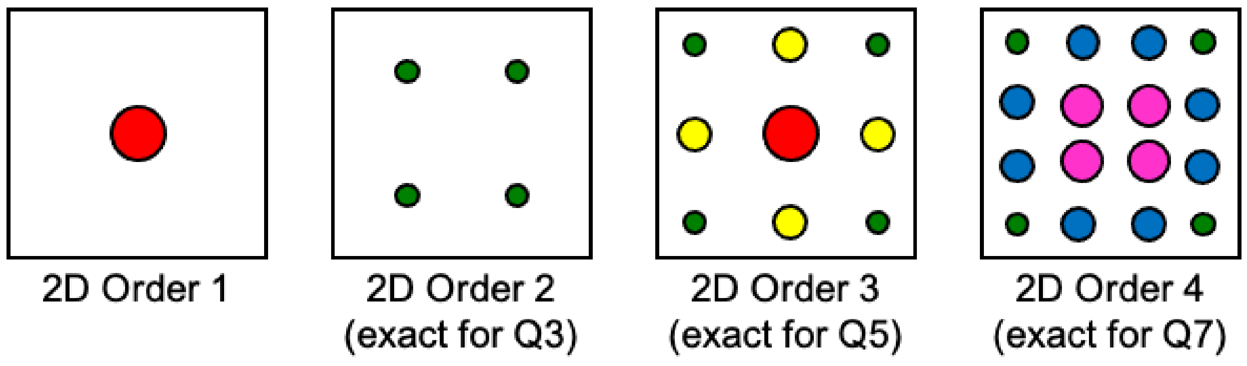

2D Order 1
2D Order 2 (exact for Q3)
2D Order 3 (exact for Q5)
2D Order 4 (exact for Q7)

Figure 2.7: Examples of 2D Gauss-Legendre quadrature rules on a reference element. The size of the points indicate their relative weight. In MARBL, the quadrature point is the "atomic" unit of spatial discretization where all physics based calculations are performed and is therefore the unit of computational work.

but also at shared edges(2D)/faces(3D) (see Figure 2.5 for a visual example of shared, high-order degrees of freedom). In this section, we consider the case where the kinematic mass matrix is fully assembled; however, it is more computationally efficient to avoid assembly of the full matrix and instead compute its *action* on a vector as needed (for example, in a conjugate gradient linear solve operation). We refer to this matrix-free, action only approach as *partial assembly* and we discuss this in great detail in Section 4.1 – the partial assembly approach is crucial to high computational performance of our method, especially on GPU architectures.

An important feature of our approach is that this mass matrix is independent of time due to (2.8):

$$\frac{d\mathbf{M}_{\mathcal{V}}}{dt} = \frac{d}{dt}\int_{\Omega(t)} \rho \mathbf{w}\mathbf{w}^{\mathrm{T}} = \int_{\Omega(t)} \rho \frac{d}{dt}(\mathbf{w}\mathbf{w}^{\mathrm{T}}) = 0\,. \tag{2.16}$$

Approximation of Integrals

In practice, the integrals for computing the kinematic mass matrix (as well as other matrices and right-hand-side vectors) are calculated by transforming the integrals over each Lagrangian zone $\Omega_z(t)$ to the standard reference zone $\hat{\Omega}_z$ by using the parametric mapping of (2.7). Applying this transformation to a general integral over a given Lagrangian zone gives

$$\int_{\Omega_z(t)} f = \int_{\hat{\Omega}_c} (f \circ \hat{\Phi})\ |\mathbf{J}_z|,$$

for some integrand f, where "$\circ$" denotes composition. We approximate these integrals using a quadrature rule of a specified order. A general integral over a Lagrangian mesh zone is therefore replaced with a weighted sum of the form

$$\int_{\hat{\Omega}_z} (f \circ \hat{\Phi})\ |\mathbf{J}_z| \approx \sum_{n=1}^{N_q} \alpha_n\ \left\{ (f \circ \hat{\Phi})\ |\mathbf{J}_z| \right\}_{\hat{\vec{x}}=\hat{\vec{q}}_n}\ , \tag{2.17}$$

where α_n are the N_q quadrature weights and $\hat{\vec{q}}_n$ are quadrature points inside of the reference zone where the integrand is sampled at. It is important to emphasize that the integrand, f, is a function which is sampled at each quadrature point inside of a zone. The quadrature point is therefore the "atomic" unit of spatial discretization and is where all physics based calculations are performed; we furthermore use quadrature points as our spatial unit of computational work (in contrast with traditional low order SGH codes which use the zone as the unit of work). In practice, we use Gauss-Legendre quadrature on quadrilaterals. A visual depiction of 2D Gauss-Legendre quadrature points and weights of various orders is shown in Figure 2.7.

Semi-discrete Energy Conservation

The thermodynamic discretization starts with the expansion of the internal energy in the basis $\{\phi_j\}$:

$$e(x,t) \approx \sum_j \mathbf{e}_j(t)\phi_j(\mathbf{x},t).$$

Consider a weak formulation of the internal energy conservation equation (2.3) obtained by multiplying it by ϕ_i and integrating over the domain $\Omega(t)$:

$$\int_{\Omega(t)} \left(\rho \frac{de}{dt}\right)\phi_i = \int_{\Omega(t)} (\sigma : \nabla v)\,\phi_i. \tag{2.18}$$

Expressing the energy in the moving thermodynamic basis gives:

$$\sum_j \frac{de_j}{dt}\int_{\Omega(t)} \rho\phi_j\phi_i = \int_{\Omega(t)} (\sigma : \nabla v)\,\phi_i.$$

In other words,

$$\mathbf{M}_{\mathscr{E}}\frac{de}{dt} = \int_{\Omega(t)} (\sigma : \nabla v)\,\phi. \tag{2.19}$$

where $\mathbf{M}_{\mathscr{E}}$ is the *thermodynamic mass matrix* which is defined by the integral

$$\mathbf{M}_{\mathscr{E}} \equiv \int_{\Omega(t)} \rho\phi\phi^{\mathrm{T}}. \tag{2.20}$$

As with the kinematic mass matrix, we can consider the thermodynamic mass matrix as being assembled from the degrees of freedom associated with each individual zone z as $\mathbf{M}_{\mathscr{E}} = Assemble(\mathbf{M}_{\mathscr{E},z})$. However, due to the discontinuous nature of the thermodynamic basis, there is no sharing of degrees of freedom across zone boundaries and so the ”assembled” thermodynamic mass matrix is *block diagonal* where each block a is symmetric positive definite matrix of dimension N_e by N_e, where N_e denotes the number of coefficients used in the basis function expansion for the internal energy.

Analogous to the kinematic case, we can use the fact that the thermodynamic basis functions have zero material derivatives to conclude that $\mathbf{M}_{\mathscr{E}}$ is independent of time.

The Force Matrix

We now introduce an $N_{\mathscr{V}} \times N_{\mathscr{E}}$ rectangular matrix $\mathbf{F}$, which we call the *force matrix* that connects the kinematic and thermodynamic spaces:

$$\mathbf{F}_{ij} = \int_{\Omega(t)} (\sigma : \nabla w_i)\,\phi_j\,. \tag{2.21}$$

As with the previously defined matrices, the matrix $\mathbf{F}$ from (2.21) is assembled from individual zone contributions:

$$\mathbf{F} = Assemble(\mathbf{F}_z)$$

This local rectangular matrix is the high-order generalization of the “corner force” concept described in [4]. It represents the hydrodynamic force contributions from a given zone to a given shared kinematic degree of freedom as well as the work done by the velocity gradient in the energy equation. As already mentioned, it is computationally advantageous to avoid full assembly of the matrix $\mathbf{F}$ and instead, compute only its action on a vector using the partial assembly technique described in Section 4.1.

Evaluating $\mathbf{F}_z$ is a locally FLOP-intensive calculation that forms the computational kernel of our finite element discretization method. Specifically, transforming each zone back to the reference element and applying quadrature yields:

$$(\mathbf{F}_z)_{ij} \approx \sum_n \alpha_n \hat{\sigma}(\hat{q}_k) : \mathbf{J}_z^{-1}(\hat{q}_k)\hat{\nabla}\hat{w}_i(\hat{q}_k)\,\hat{\phi}_j(\hat{q}_k)|\mathbf{J}(\hat{q}_k)_z|\,. \tag{2.22}$$

Note that in general, the total stress σ, including the pressure via an EOS call and any artificial viscous stress for shock capturing, is *evaluated at each quadrature point* in (2.22). Furthermore, the density (in an EOS call for example) is *evaluated at each quadrature point* using the strong mass conservation principle of (2.10). The notion of sampling the density and pressure as functions evaluated at zone quadrature points is a key component of our high-order discretization approach and is essential for robust behavior. Indeed, it is the reason we do not require any special hourglass filters and is similar in nature to the sub-zonal pressure method of [19].

The Euler Equations in Semi-discrete Form

Given the previous definitions, we can summarize the general semi-discrete Lagrangian conservation laws in the following simple form:

$$\text{Momentum Conservation:} \quad \mathbf{M}_{\mathcal{V}} \frac{d\mathbf{v}}{dt} = -\mathbf{F} \cdot \mathbf{1}, \tag{2.23}$$

$$\text{Energy Conservation:} \quad \mathbf{M}_{\mathcal{E}} \frac{de}{dt} = \mathbf{F}^{\mathrm{T}} \cdot \mathbf{u}, \tag{2.24}$$

$$\text{Equation of Motion:} \quad \frac{d\mathbf{x}}{dt} = \mathbf{u}. \tag{2.25}$$

The vector $\mathbf{1}$ above is the representation of the constant one in the thermodynamic basis $\{\phi_i\}$ (we assume that $1 \in \mathcal{E}$). Boundary conditions can be implemented by eliminating the corresponding rows and columns in $\mathbf{M}_{\mathcal{V}}$ (which will in general introduce a difference between its blocks).

Explicit Time Integration and the Fully-Discrete Approximation

Since MARBL is a multiphysics code, we must consider multiple time scales over which various physical processes take place. For compressible hydrodynamics, the time scales are typically *fast* relative to other physics like diffusion/conduction. In addition, since the semi-discrete equations of the previous section are non-linear, we typically employ explicit time discretization techniques to evolve the conservation laws of (2.23)-(2.25).

Specifically, let $Y = (\mathbf{v};\mathbf{e};\mathbf{x})$ be the hydrodynamic state vector. Then the semi-discrete conservation equations of (2.23)-(2.25) can be written in the form:

$$\frac{\mathrm{d}Y}{\mathrm{d}t} = \mathscr{F}(Y,t), \qquad \text{where} \qquad \mathscr{F}(Y,t) = \begin{pmatrix} -\mathbf{M}_{\mathcal{V}}^{-1}\mathbf{F}\cdot\mathbf{1} \\ \mathbf{M}_{\mathcal{E}}^{-1}\mathbf{F}^{\mathrm{T}}\cdot\mathbf{v} \\ \mathbf{v} \end{pmatrix}.$$

Standard high-order time integration techniques (e.g. explicit Runge-Kutta methods) can be applied to this system of nonlinear ODEs. However, these standard methods may need modifications to ensure numerical stability of the scheme and to ensure exact energy conservation. Note that in general, a time integration method of order N will require N evaluations of the function $\mathscr{F}(Y,t)$.

For so called *fast* physics we utilize an explicit, midpoint Runge-Kutta second order, energy conserving method. This explicit method is part of a more general multi-rate implicit/explicit (IMEX) time integration approach for multiphysics which we describe in Section 2.7. In MARBL, the default method for evolving the fully discrete hydrodynamics equations is given by:

$$Y^{n+\frac{1}{2}} = Y^n + \frac{\Delta t}{2}\mathscr{F}(Y^n,t^n), \qquad Y^{n+1} = Y^n + \Delta t\,\mathscr{F}(Y^{n+\frac{1}{2}},t^{n+\frac{1}{2}}).$$

In practice, we have observed that the above scheme may be unstable even for simple test problems. Therefore, we developed a modification of the scheme to improve its stability and to ensure total energy conservation. Its two stages are given by

$$\begin{aligned} \mathbf{v}^{n+\frac{1}{2}} &= \mathbf{v}^n - (\Delta t/2)\,\mathbf{M}_{\mathcal{V}}^{-1}\mathbf{F}^n\cdot\mathbf{1} & \qquad \mathbf{v}^{n+1} &= \mathbf{v}^n - \Delta t\,\mathbf{M}_{\mathcal{V}}^{-1}\mathbf{F}^{n+\frac{1}{2}}\cdot\mathbf{1} \\ \mathbf{e}^{n+\frac{1}{2}} &= \mathbf{e}^n + (\Delta t/2)\,\mathbf{M}_{\mathcal{E}}^{-1}(\mathbf{F}^n)^{\mathrm{T}}\cdot\mathbf{v}^{n+\frac{1}{2}} & \qquad \mathbf{e}^{n+1} &= \mathbf{e}^n + \Delta t\,\mathbf{M}_{\mathcal{E}}^{-1}(\mathbf{F}^{n+\frac{1}{2}})^{\mathrm{T}}\cdot\bar{\mathbf{v}}^{n+\frac{1}{2}} \\ \mathbf{x}^{n+\frac{1}{2}} &= \mathbf{x}^n + (\Delta t/2)\,\mathbf{v}^{n+\frac{1}{2}} & \qquad \mathbf{x}^{n+1} &= \mathbf{x}^n + \Delta t\,\bar{\mathbf{v}}^{n+\frac{1}{2}}, \end{aligned}$$

where $\mathbf{F}^k = \mathbf{F}(Y^k)$ and $\bar{\mathbf{v}}^{n+\frac{1}{2}} = (\mathbf{v}^n + \mathbf{v}^{n+1})/2$.

The RK2-Average scheme described above conserves the discrete total energy exactly: The change in kinetic and internal energy can be expressed as

$$\mathscr{K}^{n+1} - \mathscr{K}^n = (\mathbf{v}^{n+1} - \mathbf{v}^n) \cdot \mathbf{M}_{\mathscr{V}} \cdot \bar{\mathbf{v}}^{n+\frac{1}{2}} = -\Delta t \, (\mathbf{F}^{n+\frac{1}{2}} \cdot \mathbf{1}) \cdot \bar{\mathbf{v}}^{n+\frac{1}{2}}$$
$$\mathscr{E}^{n+1} - \mathscr{E}^n = \mathbf{1} \cdot \mathbf{M}_{\mathscr{E}} \cdot (\mathbf{e}^{n+1} - \mathbf{e}^n) = \Delta t \, \mathbf{1} \cdot (\mathbf{F}^{n+\frac{1}{2}})^{\mathrm{T}} \cdot \bar{\mathbf{v}}^{n+\frac{1}{2}}) \, .$$

Therefore the discrete total energy is preserved: $\mathscr{K}^{n+1} + \mathscr{E}^{n+1} = \mathscr{K}^n + \mathscr{E}^n$.

Hyperviscosity for Shock Capturing

Hyperviscosity enables shock capturing while preserving the high-order properties of the underlying discretization away from the shock region. Specifically, we compute a high-order term based on a product of the mesh length scale to a high power scaled by a hyper-Laplacian operator applied to a scalar field. We then form the total artificial viscosity by taking a non-linear blend of this term and a traditional artificial viscosity term. The details of the approach used in MARBL can be found in [20, 15].

We consider an artificial stress tensor, σ_a, which is a product of a strain rate tensor with a mesh dependent, non-linear coefficient

$$\sigma_a = \mu_\ell \varepsilon \, , \tag{2.26}$$

where

$$\varepsilon \equiv \frac{1}{2} \left(\nabla v + v \nabla \right) . \tag{2.27}$$

The artificial viscosity coefficient is defined as

$$\mu_\ell \equiv \min(\mu_{hyp}, \mu_{std}) \, . \tag{2.28}$$

The *standard* artificial viscosity coefficient, μ_{std}, given by

$$\mu_{std} \equiv \rho \left(q_2 \ell^2 \left| \delta_s v \right| + q_1 \ell c_s \right) , \tag{2.29}$$

where ρ is the density, c_s is the speed of sound, q_2 and q_1 are the so called “quadratic” and “linear” scaling coefficients, ℓ is a mesh dependent length scale, and $\delta_s v$ is the directional measure of compression. The standard viscosity coefficient is fundamentally a first order function of the mesh dependent length scale, ℓ, due to the q_1 scaled linear component which will prevent high-order numerical convergence (defined as convergence greater than first order under mesh refinement) for any discretization as long as it is active. We therefore also consider the *hyperviscosity* coefficient which is defined as

$$\mu_{hyp} \equiv q_3 \rho \ell^{2m} \overline{\left| \Delta^m g(v) \right|}, \quad m = 1, 2, 3, \ldots \, . \tag{2.30}$$

The term q_3 is an additional scaling parameter in the spirit of the linear, q_1, and quadratic, q_2, terms of (2.29). The overbar in this case denotes a generic, mesh based smoothing operation. As in the case of [20], this smoothing process serves the purpose of eliminating “cusps introduced by the absolute value operator” which is required to ensure a positive definite coefficient. The term $g(v)$ is a scalar function of the velocity field which we define as

$$g(v) \equiv \ell_0^2 \left(\frac{|\mathbf{J}_z|}{|\mathbf{J}_z(0)|} \right)^{\frac{2}{d}} \sqrt{(\varepsilon : \varepsilon)} \, , \tag{2.31}$$

where d is the spatial dimension, $|\mathbf{J}_z|$ denotes the value of the Jacobian determinant (a point wise definition of volume) on the current Lagrangian mesh and $|\mathbf{J}_z(0)|$ the value of the Jacobian determinant on the initial mesh. The term ℓ_0 denotes the *initial* length scale, defined on the initial mesh as originally described in [9]. The hyper viscosity coefficient as defined via (2.30) retains the same spatial scaling as the one of [20], namely

$$\mu_{hyp} \sim \ell^{2m+2} \quad m = 1, 2, 3, \ldots \, ,$$

which is key to obtaining high-order convergence when using a high-order spatial discretization.

In implementing the hyperviscous term (2.30) we need to discretize a Laplacian operator, $\Delta^m v$, and a smoothing operator, $\overline{f}$, using *continuous* Galerkin high-order finite elements. Continuous Galerkin is used since we must ultimately compute high-order spatial derivatives of the underlying velocity field, which is itself approximated using a continuous Galerkin method (to provide a well defined field with which to move the Lagrangian mesh).

The Laplacian operator is defined by

$$f = \Delta v = \sum_{d=1}^{D} \frac{\partial^2 v}{\partial^2 x_d} \tag{2.32}$$

for D dimensions. Applying the same continuous Galerkin discretization process to this operator as we have for other continuum equations, we can derive a discrete form of the Laplacian operator as

$$\mathbf{M}_{\mathscr{V}} \mathbf{f} = \tilde{\mathbf{S}} \mathbf{v}, \tag{2.33}$$

where $\mathbf{M}_{\mathscr{V}}$ is the kinematic mass matrix, $\mathbf{f}$ and $\mathbf{v}$ are the vectors of the finite element coefficients for f and v, and the *full stiffness matrix* $\tilde{\mathbf{S}} = \mathbf{S}_{\text{bc}} - \mathbf{S}$ is composed of a boundary integral term $\mathbf{S}_{\text{bc}}$ and the textupstiffness matrix $\mathbf{S}$. Another way to express this is to say that the discrete form of the Laplacian is defined by the operator $\Delta \approx \mathbf{M}_{\mathscr{V}}^{-1}\tilde{\mathbf{S}}$. Higher order Laplacians are computed by successive applications of this discrete operator, i.e. $\Delta^m \approx \left(\mathbf{M}_{\mathscr{V}}^{-1}\tilde{\mathbf{S}}\right)^m$. As with the previously defined global matrices, the discrete hyperviscosity operator can benefit from the matrix-free partial assembly techniques discussed in Section 4.1.

2.2.2 High-order remesh phase

The goal of the remesh phase in an ALE calculation is to optimize the nodal position vector from a given Lagrangian state, $\mathbf{x} \longrightarrow \mathbf{x}^{new}$ to produce new mesh zones of a user specified quality. There are many criteria a user may employ in optimizing the coordinates of a given mesh, but in general the following high level goals are common:

- Improve the shape and size of highly distorted zones to enable larger explicit time steps
- Adapt the mesh to a region or regions of interest
- Apply constraints to certain regions / surfaces to prevent any remesh displacement (i.e. to keep certain regions / surfaces Lagrangian to prevent mixed or multimaterial zones)
- Prevent the mesh from being optimized in regions which have not yet experienced any Lagrangian motion

There are several numerical techniques that can be used to solve the general problem of ALE remesh. MARBL has multiple algorithms available to the user, ranging in complexity from simple and inexpensive linear mesh relaxation methods to more sophisticated and computationally expensive non-linear optimization techniques. The method of choice for complex multimaterial ALE problems is the Target Matrix Optimization Paradigm (TMOP) which solves a global minimization problem based on a user specified mesh quality objective function [21, 22]. In practice, the mesh optimization strategy a user employs in their simulation will involve multiple trade-offs between computational performance (run time) and physics fidelity (e.g.resolving regions of interest). An example is shown in Figure 2.8.

High-Order Linear Mesh Relaxation

The most basic options available in MARBL for the remesh phase of ALE are the class of simple linear iteration relaxation schemes, which apply a user specified number of iterations of a linear smoothing operator to the Lagrangian mesh coordinate vector. Due to their simplicity, these methods are computationally inexpensive, but are not robust (i.e. do not guarantee non-inverted elements under iteration) in more complex problems. These are not recommended in practice for anything other than simple test problems or simplified geometric configurations.

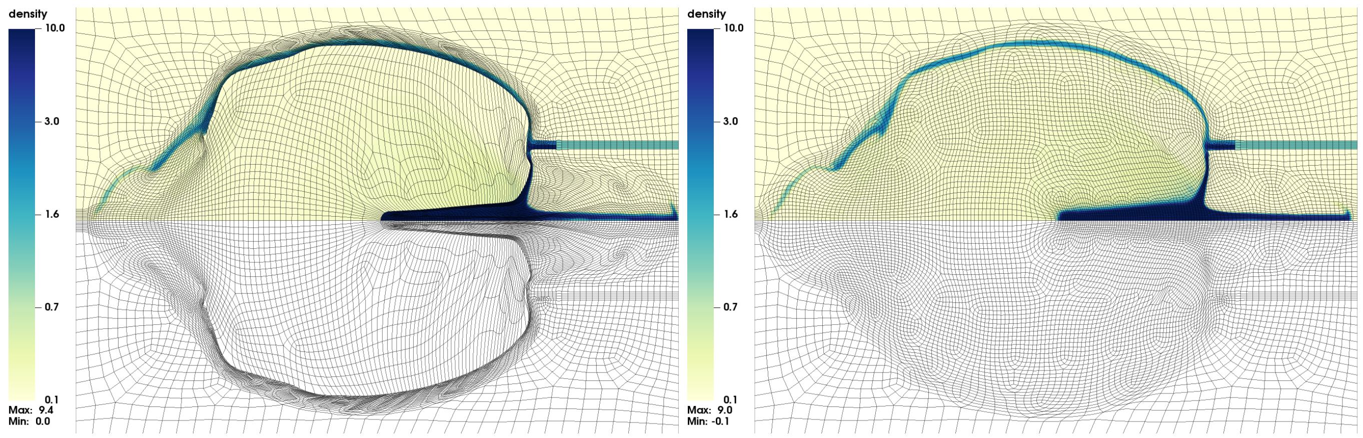


Figure 2.8: Density and high-order mesh for a $2Drz$ MARBL simulation of the BRL81A shaped charge using high-order ALE + TMOP for mesh optimization. On the Left, we utilize Lagrangian surface constraints to keep the inside of the case and the back surface of the copper liner Lagrangian (cycle count: 24,939; runtime: 24 minutes). On the right we utilize material adaptivity to concentrate zones in the regions of interest (cycle count: 8,662; runtime: 11 minutes).

We begin with the vector of (high-order) mesh position values decomposed into interior and boundary sets such that

$$\mathbf{x} = (\mathbf{x}_I, \mathbf{x}_B).$$

Now define a "mesh Laplacian" operator $\mathbf{L} = (\mathbf{L}_{II}, \mathbf{L}_{IB})$. The matrix $\mathbf{L}$ should be a *topological* operator (independent of the geometry $\mathbf{x}$), with

$$\sum_j \mathbf{L}_{ij} = 0.$$

Then

$$\mathbf{x}_I^{n+1} = \mathbf{x}_I^n + (\mathbf{f}_I - \mathbf{L}_{II}\mathbf{x}_I^n)$$

where $\mathbf{f}_I = -\mathbf{L}_{IB}\mathbf{x}_B^0$ and $\mathbf{x}_B^n = \mathbf{x}_B^0$. This is a simple linear iteration, which in limit converges to the harmonic extension of the boundary nodes to the interior.

More generally, we consider a class of linear harmonic relaxation schemes:

$$\mathbf{x}^{n+1} = \mathbf{x}^n + \mathbf{M}^{-1}(\mathbf{f} - \mathbf{L}\mathbf{x}^n) \tag{2.34}$$

where

- $\mathbf{x}^n$ are the nodes of the high-order mesh after n relaxation steps.
- $\mathbf{L}$ is a "mesh Laplacian" extended with $\mathbf{L}_{BI} = 0$, $\mathbf{L}_{BB} = I$, and $\mathbf{f} = (-\mathbf{L}_{IB}\mathbf{x}_B^0, \mathbf{x}_B^0)$.
- $\mathbf{M}$ is a smoother (preconditioner) for $\mathbf{L}$.

A particular algorithm is specified by the choice of $\mathbf{L}$, $\mathbf{M}$ and the number of smoothing steps. The mesh Laplacian $\mathbf{L}$ can be defined in different ways, including:

- Using the FEM sparsity to connect the high-order nodes with *equal weights*.
- Assembling the high-order stiffness matrix on the *reference element*.
- $\mathbf{L} = \mathbf{G}^t\mathbf{G}$, where $\mathbf{G}$ is the *"discrete gradient"* matrix between the high-order nodal H^1 and Nedelec $H(curl)$ spaces.

There are also several choices for the smoothing matrix **M**, for example:

- Simple diagonal scaling
$$\mathbf{M}_{ii} = \sum_j |\mathbf{L}_{ij}| \,.$$
- Custom low-frequency preserving *polynomial smoothers* [23].

High-Order Non-linear Mesh Relaxation

More generally, we consider an integral minimization process applied to a so called "mesh energy function" of the form

$$E(\mathbf{x}) = \sum_z \int_{\hat{\Omega}_z} B(\hat{\mathbf{J}}_z(\hat{\mathbf{x}})).$$

for some functional B. In this form, harmonic optimization simply corresponds to the choice

$$B(\hat{\mathbf{J}}_z) \equiv \frac{1}{2}(\hat{\mathbf{J}}_z : \hat{\mathbf{J}}_z) = \frac{1}{2}\,\mathrm{tr}(\mathbf{J}_z^T \mathbf{J}_z).$$

The inverse harmonic approach of Winslow minimizes the gradient norm of the *inverse* map $\hat{\Phi}_z^{-1}$:

$$E(\mathbf{x}) = \frac{1}{2}\sum_z \int_{\Omega_z} \mathbf{J}_z^{-1} : \mathbf{J}_z^{-1}.$$

Using a change of variables, we can write each zone integral on the reference zone as

$$\int_{\Omega_z} (\mathbf{J}_z^{-1} : \mathbf{J}_z^{-1}) = \int_{\hat{\Omega}_z} (\mathbf{J}_z^{-1} : \mathbf{J}_z^{-1}) |\mathbf{J}_z|$$

Thus, in the general setting the inverse harmonic optimization method corresponds to the energy functional

$$B(\hat{\mathbf{J}}_z) \equiv \frac{1}{2} |\mathbf{J}_z|\, \mathrm{tr}(\mathbf{J}_z^{-T} \mathbf{J}_z^{-1})\,.$$

In contrast with the harmonic case, the stationary point equations are *nonlinear*.

Newton's method is then applied for the stationary point equation $\nabla E(\mathbf{x}) = 0$, namely

$$\mathbf{x}^{n+1} = \mathbf{x}^n - [\mathscr{H} E(\mathbf{x}^n)]^{-1} \nabla E(\mathbf{x}^n)$$

For a given $\mathbf{x}$, the energy function $E(\mathbf{x})$, the gradient vector $\nabla E(\mathbf{x})$ and the Hessian matrix $\mathscr{H} E(\mathbf{x})$ can be assembled element-by-element based on $B(\hat{\mathbf{J}}_z)$ and its derivatives.

The most capable and flexible approach we have is based on the Target Matrix Optimization Paradigm (TMOP) which is a non-linear optimization method where the energy function can be customized by the user, including different terms for mesh quality (with various options taking into account zone size and shape), mesh displacement *limiting* (an objective term which keeps regions of the mesh from moving if they have not experienced Lagrangian motion) and mesh adaptivity, where the optimized mesh can spatially concentrate in material regions and interfaces [21, 22]. While TMOP has proven to be exceptionally powerful and useful for complex multimaterial ALE simulations, it is also computationally expensive and therefore benefits from the matrix-free partial assembly techniques discussed in Section 4.1.

2.2.3 High-order ALE remap

Once the Lagrangian mesh has been modified by the remesh phase, the simulation state data needs to be remapped onto this new mesh. It is important to emphasize that the remap phase does not involve any physics, it is best understood as an interpolation process whose goal is to conservatively, accurately and monotonically transfer the field data from the Lagrangian mesh, $\mathbf{x}$, to the newly optimized mesh from the remesh phase, $\mathbf{x}^{new}$. There are multiple ways to solve this problem, but one attractive approach is to formulate the remap process as a pseudo-time advection problem where we imagine each of the fields that need to be remapped are transported or advected form the Lagrange mesh to the new mesh. The main benefits of this approach are its inherent conservation properties and its relatively low computational cost compared to methods based on computing geometric intersections of high-order meshes.

In the high-order finite element approach, the remap phase is based on a high-order discontinuous Galerkin (DG) formulation of the advection equation for each conserved variable. Semi-discrete DG methods for advection equations can be formulated in terms of high-order finite element "mass" and "advection" matrices. These formulations result in high-order accuracy for sufficiently smooth fields, but produce non-monotonic results (spurious oscillations) for discontinuous fields. We therefore employ techniques from Flux Corrected Transport (FCT) methods which start with a low-order, strictly monotone method (based on a fully lumped mass matrix and a fully "up-winded" advection matrix) and add in "anti-diffusive", conservative, high-order flux corrections in a non-linear manner to recover high-order accuracy whenever the underlying solution is sufficiently smooth.

High-Order Discontinuous Galerkin Advection Based Remap

Here we provide a very brief overview of the high-order Discontinuous Galerkin (DG) advection-based ALE remap scheme. The goal of ALE remap is to transfer a field, e.g. the mass density ρ by solving the pseudo-time advection equation

$$\frac{\mathrm{d}\rho}{\mathrm{d}\tau} = u \cdot \nabla \rho \,, \tag{2.35}$$

where $u \equiv x^{new} - x$ is the mesh displacement vector and $\tau \in [0,1]$ is a fictitious time interval associated with a linearly interpolated transition between the original (Lagrangian) mesh defined by the coordinates x and a new (rezoned) mesh obtained by the remesh phase with coordinates x^{new}. Motivated by conservation of mass (the integral of ρ), we derive a weak formulation of the advection equation (2.35) using the Reynolds transport theorem, the fact that $\frac{\mathrm{d}\phi}{\mathrm{d}\tau} = 0$ and integrating by parts to get

$$\begin{aligned} \frac{\mathrm{d}}{\mathrm{d}\tau}\int_{\Omega(\tau)} \rho\phi &= \int_{\Omega(\tau)} \frac{\mathrm{d}\rho}{\mathrm{d}\tau}\phi + \rho\frac{\mathrm{d}\phi}{\mathrm{d}\tau} + \rho\phi\nabla\cdot u = \int_{\Omega(\tau)} (u\cdot\nabla\rho)\phi + \rho\phi\nabla\cdot u \\ &= \int_{\Omega(\tau)} \nabla\cdot(\rho u)\phi = -\int_{\Omega(\tau)} \rho u\cdot\nabla\phi + \int_{\partial\Omega(\tau)} \rho u\cdot n\phi \,. \end{aligned} \tag{2.36}$$

We approximate the density function (or any other thermodynamic field which is remapped) using the expansion

$$\rho(\mathbf{x},\tau) = \sum_i \rho_i(\tau)\phi_i(\mathbf{x},\tau) = \rho(\tau)^{\mathrm{T}}\phi(\mathbf{x},\tau) \,, \tag{2.37}$$

where $\{\phi_i\}_{i=1}^{N_\mathscr{V}}$ is a (moving) basis of a *discontinuous* finite element space, $\rho(\tau)$ is an unknown pseudo-time dependent vector of size $N_\mathscr{E} = \dim \mathscr{E}(\tau)$ and ϕ is a column vector of all the basis functions. The finite element functions in $\mathscr{E}(\tau)$ are discontinuous across the interior faces $f \in \partial\Omega_z(\tau)$ of the mesh, and therefore to fix notation we introduce the jump $[\![\cdot]\!]$ and average $\{\cdot\}$ operators:

$$[\![\phi]\!] = \phi^- - \phi^+ ,$$

$$\{\phi\} = \frac{1}{2}\left(\phi^- + \phi^+\right) ,$$

$$\text{where} \quad \phi^\pm(x) = \lim_{s\to 0+} \phi(x \pm s n_f(x)), \quad x \in f ,$$

and n_f is a normal vector field on each interior face f of the mesh, which varies continuously on f. Given these definitions, we can define a semi-discrete advection equation as

$$\mathbf{M}\frac{\mathrm{d}\rho}{\mathrm{d}\tau} = \mathbf{K}\rho\,, \tag{2.38}$$

where the mass and advection matrices are defined as:

$$(\mathbf{M})_{ij} = \int_{\Omega(\tau)} \phi_i\phi_j$$
$$(\mathbf{K})_{ij} = \sum_z \int_{\Omega_z(\tau)} u\cdot\nabla\phi_j\phi_i - \sum_f \int_{\partial\Omega(\tau)} (u\cdot n)[\![\phi_j]\!](\phi_i)_d$$

The high-order DG semi-discrete formulation of (2.38) with unmodified matrices/vectors will result in high-order accuracy for advection of smooth fields, but will cause spurious oscillations for discontinuous fields; which are present for the important case of material contacts/interfaces. It is essential for the ALE remap process to be "monotonic" which means that it does not introduce any new local extremum. To enforce this constraint, we must employ a non-linear process to modify both the mass matrix $\mathbf{M}\to\mathbf{M}^*$ and the advection matrix $\mathbf{K}\to\mathbf{K}^*$ to render the solution vector local extremum diminishing (LED). There are multiple ways to achieve this in practice, but we have found that the most robust and efficient approach is to use a form of flux corrected transport (FCT) modified for use with arbitrary order DG basis functions, the details of which are described in [12, 14]. In this method, the advection equation is modified to blend a provably monotone first order accurate method in the neighborhood of discontinuities with the general high-order solution everywhere else to achieve both accuracy and monotonicity.

2.2.4 Multimaterial Extension

MARBL is a single fluid, multi-material code; which means that at any given point in space x, the simulation may consist of a collection of different user defined "materials" (sometimes referred to as phases or species) with distinct material properties (e.g. EOS, strength, conductivity, opacity, etc ...) and with distinct thermodynamic state (material specific density, energy, pressure, etc ...). The spatial configuration of a material with index k is defined by a volume fraction field, $\eta_k(x)$. We don't assume that the material quantities specified by η_k are in a mixture state at the point x ; instead, we think of the materials as being fundamentally separate (imagine there is an immiscible boundary between them).

The volume fraction fields (also known as "material indicators") are discretized in the same discontinuous finite element space as the rest of the thermodynamic quantities. This means that volume fractions can be perfectly discontinuous if the initial mesh is aligned with the fields and remain this way for all time if the simulation is evolved in a purely Lagrangian manner (see for example Figure 2.2). However, for ALE simulations where it is possible for the mesh to move across material interfaces, we will encounter so called multi-material zones (more specifically, multi-material quadrature points).

At each point x on the computational mesh we require $\eta_k(x) > 0$ (to prevent infinite density), $\sum_k \eta_k(x) = 1$ (partition of unity is required to cover the entire spatial domain), well-defined material densities ρ_k and material energies e_k throughout the mesh, and a means for computing material specific constitutive properties, like pressure p_k. The initialization of $\eta_k(x)$ is carried out in two steps. First, the initial jump of each η_k is approximated by a finite element function in the positive Bernstein basis:

$$\eta_k(x,t=0) = \begin{cases} 1 & \text{if material } k \text{ is present at } x \\ 0 & \text{otherwise} \end{cases} \tag{2.39}$$

Second, these finite element functions are interpolated at the quadrature points, by which the initial values are obtained. During the Lagrangian phase, we treat the volume fraction fields as functions on the quadrature points, just like we do with density via (2.10). The utilization of the positive Bernstein basis results in a smooth transition between 0 and 1 within a multi-material zone. Note that this procedure might introduce an initial volume error for more complicated interfaces that are not aligned with the finite element degrees of freedom within a zone.

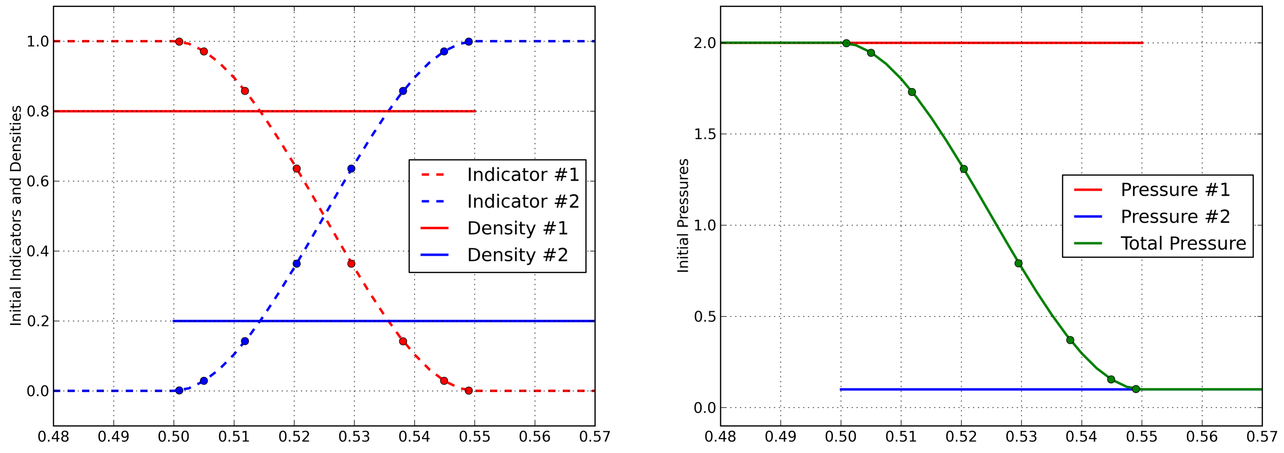


Figure 2.9: A 1D example of of material-specific initialization. Left: Material volume fractions (indicators) and densities. Right: Material pressures and total pressure. Solid dots correspond to values at quadrature points.

The material specific densities and internal energies are initialized according to

$$\rho_k(x,t=0) = \begin{cases} \rho(x,0)|_k & \eta_k(x,0) > \eta_{cut}\,, \\ 0 & \eta_k(x,0) \le \eta_{cut}\,, \end{cases} \qquad e_k(x,t=0) = \begin{cases} e(x,0)|_k & \eta_k(x,0) > \eta_{cut}\,, \\ 0 & \eta_k(x,0) \le \eta_{cut}\,. \end{cases} \tag{2.40}$$

Here $\rho|_k$ and $e|_k$ provide additional values for material k in mixed cells, at points where the material's values are not defined by the initial conditions, i.e., $\rho|_k$ and $e|_k$ extend the initial conditions for material k wherever $\eta_k(x,0) > \eta_{cut}$, where η_{cut} is a volume fraction cutoff value which we default to 10^{-12}. The aim of these extensions is to define the material-dependent quantities throughout the whole zone and obtain pressures that agree with the initial conditions. An example of the initialization between two materials in 1D is shown in Figure 2.9. At present, we do not perform interface reconstruction in MARBL, we instead have "diffuse interfaces" as defined by the intermediate values of the material volume fraction fields (i.e. regions where $0 < \eta_k < 1$); however, one can interpret the 50% iso-contour of a given material volume fraction field as the location of an "interface."

Because we have only a single velocity (required to have a single Lagrangian mesh), we have only a single value for strains at a point x during the Lagrangian phase (specifically, volumetric strain). However, the multiple materials that may exist at point x as defined by their volume fraction $\eta_k(x)$ may have wildly different responses to that single strain value (e.g. Hydrogen and Copper have very different compressibilities). To reconcile this, we require a multi-material "closure model" which changes the volume fractions during the Lagrangian phase in a more physically realistic manner. There are multiple models in the literature for defining such closure models. We adopt an approach based on the original method of R. Tipton [24] which invokes pressure equilibrium to adjust the volume fractions during the Lagrangian phase. We augment the semi-discrete Euler equations with an additional evolution equation for the indicator functions, namely

$$\frac{d\eta_k}{dt} = \alpha_k\,.$$

The special case of equal volumetric strain corresponds to the case $\alpha_k = 0$ (i.e. the volume fractions are transported in a pure Lagrangian manner). Use of Tipton like methods can be applied to define a more physically motivated closure model based on a pressure relaxation process which is described in detail in [13]. The main difference for the high-order finite element case used in MARBL is that the closure model is evaluated in a semi-discrete way at every element quadrature point, making it valid for arbitrarily high-order space/time finite element formulations.

The multi-material case also presents new challenges for the remap phase, where we must now transport the material volume fractions in a monotonic and conservative manner. The main challenge is that to ensure conservation during the remap phase, we must remap (or advect) the conserved quantities like material mass and total material energy; however we want to enforce monotonicity on the quantities we evolve during the Lagrangian phase, namely density and internal energy. The methods we use to overcome these challenges are described in detail in [14]. Here, we will simply describe the end results. We can summarize the overall high-order finite element multi-material ALE process in semi-discrete form as:

	Lagrangian Phase	**Remap Phase**
Momentum:	$\mathbf{M}_{\mathscr{V}} \frac{\mathrm{d}\mathbf{v}}{\mathrm{d}t} = -\sum_k \mathbf{F}_k \cdot 1$	$\mathbf{M}_{\mathscr{V}} \frac{\mathrm{d}\mathbf{v}}{\mathrm{d}\tau} = \mathbf{K}_{\mathscr{V}} \mathbf{v}$
Volume Fraction:	$\frac{\mathrm{d}\eta_k}{\mathrm{d}t} = \alpha_k$	$\mathbf{M}^* \frac{\mathrm{d}\eta_k}{\mathrm{d}\tau} = \mathbf{K}^* \eta_k$
Mass:	$\eta_k \rho_k \lvert\mathbf{J}_z\rvert(t) = \eta_{k0}\rho_{k0}\lvert\mathbf{J}_z\rvert(0)$	$\mathbf{M}^* \frac{\mathrm{d}\langle \eta\rho_k \rangle}{\mathrm{d}\tau} = \mathbf{K}^* \langle \eta\rho_k \rangle$
Energy:	$\mathbf{M}_{\mathscr{E},k} \frac{\mathrm{d}\mathbf{e}_k}{\mathrm{d}t} = \mathbf{F}_k^{\mathrm{T}} \mathbf{v} - \mathbf{c}_k$	$\mathbf{M}^* \frac{\mathrm{d}\langle \eta\rho e_k \rangle}{\mathrm{d}\tau} = \mathbf{K}^* \langle \eta\rho e_k \rangle$
Position:	$\frac{\mathrm{d}\mathbf{x}}{\mathrm{d}t} = \mathbf{v}$	$\frac{\mathrm{d}\mathbf{x}}{\mathrm{d}\tau} = \mathrm{Remesh}(\mathbf{x}\rvert_{\tau=0})$

During the Lagrange phase, the total momentum is computed from the sum of material specific forces. During the remap phase, we employ an unmodified, high order continuous Galerkin (CG) advection formulation to remap the momentum from the Lagrange mesh to the optimized mesh. We evolve the material volume fractions during the Lagrange phase according to the multi-material closure model while in the remap phase they are advected using an FCT based DG formulation (presently with no material interface awareness). The material specific density and internal energy are evolved during the Lagrangian phase as described in previous sections; however during the remap phase, we advect material masses total energies in a conservative and monotonic manner.

As an example, high-order multi-material ALE results are shown in Figure 2.10 using a Q_3Q_2 finite elements (i.e. bi-cubic continuous finite elements for the kinematic space and discontinuous bi-quadratic finite elements for the thermodynamics space) on a sequence of uniformly refined meshes, beginning with a base resolution of 56×24 zones. Every 100 Lagrange steps are followed by a remesh/remap step. Figure 2.10 shows the total density and the continuous material map, defined as $\sum_{k=1}^{N} k\eta_k$, at different times during the simulation. In all cases we observe sub-zone resolution of the material map and densities and increased vortical motion at the triple point as the mesh resolution is increased.

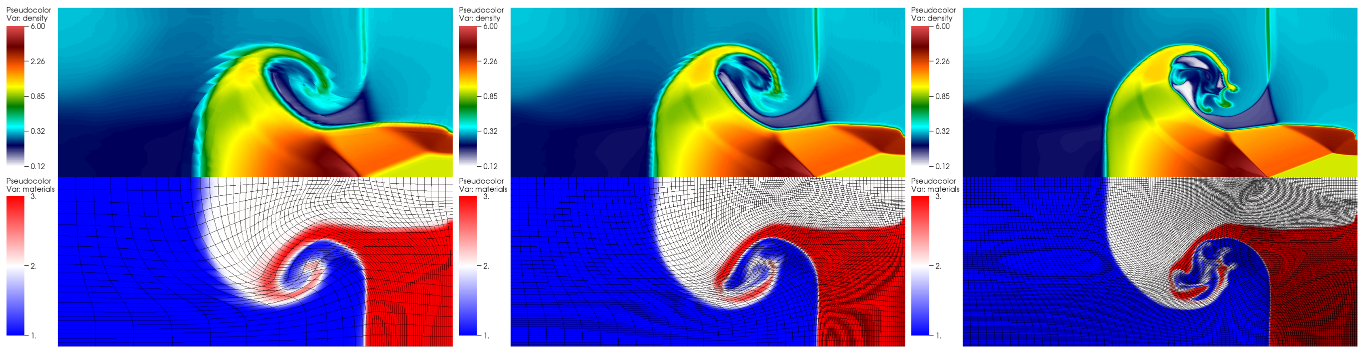


Figure 2.10: Density and continuous material map ($\sum_{k=1}^{N} k\eta_k$) at three different mesh resolutions (increasing left to right) using Q_3 finite elements and limited-harmonic mesh smoothing strategies for the 2D multi-material shock triple point benchmark.

2.2.5 Spherical 1D, Cylindrical 2D and Cartesian 3D Geometries

The need for high-resolution 3D simulations is a major motivation for our effort in targeting exascale performance; however, we still require the ability to perform routine 1D and 2D simulations as recommended in [1]. There are several reasons for this, including: the ability for users to perform rapid parameter space studies, the need for ensembles of thousands of lower-fidelity simulations to inform larger high fidelity simulations and perhaps most important, the inherent ease in debugging complicated physics problems in 1D and 2D vs the much more complex 3D case. To accommodate this, MARBL supports 1D spherical, 2D cylindrical and Cartesian as well as full 3D Cartesian options.

For 2D cylindrical (or axisymmetric) as well as 1D spherical configurations, we begin by assuming a general 3D domain Ω and consider specific restrictions (or cuts) of this domain specific to cylindrical or spherical symmetry. In cylindrical coordinates (r,θ,z), Ω can be obtained from a “meridian cut” Γ in the r-z plane by a rotation around the axis $r=0$. In spherical coordinates (r,θ,ϕ), Ω can be obtained from a “ray cut” R in the r-direction by a rotation around the θ and ϕ angles. These cases are depicted in Figure 2.11.

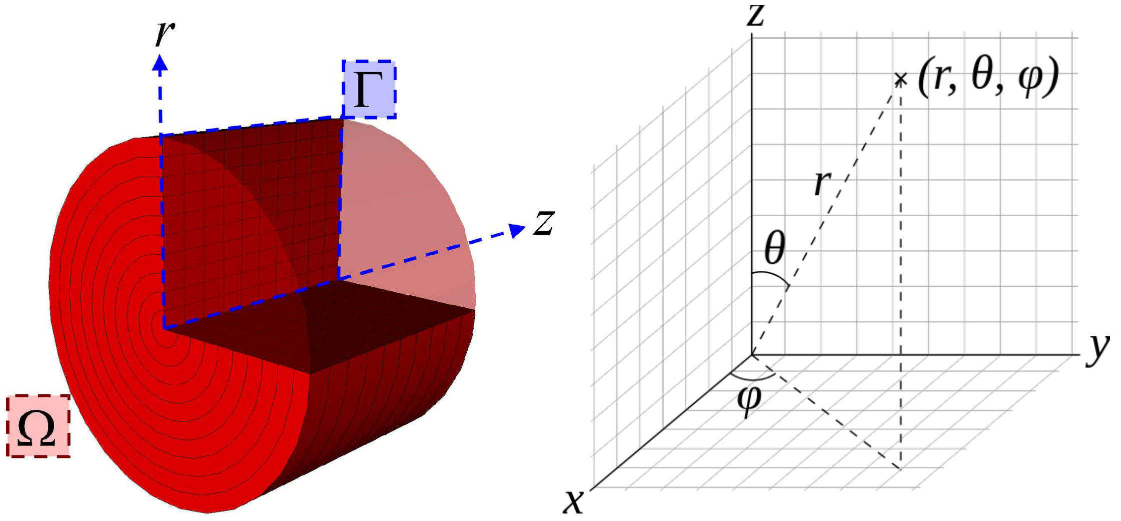


Figure 2.11: Schematic depiction of 2D cylindrical (*left*) and 1D spherical (*right*) coordinate representations on reduced computational domains.

A scalar function f, defined on the axisymmetric domain Ω, is called axisymmetric if it is independent of θ, i.e. $f(r,\theta,z) = f(r,z)$, so f is uniquely determined by its values in Γ. Likewise, a scalar function f, defined on the spherically symmetric domain Ω, is called spherically symmetric if it is independent of θ and ϕ, i.e. $f(r,\theta,\phi) = f(r)$, so f is uniquely determined by its values in R. This means we can reduce the computational mesh to 2D for the cylindrical case and 1D for the spherical case as shown in Figure 2.12, resulting in a substantial savings in computational cost relative to the full 3D case. A key property of axisymmetric functions is that their integrals over Ω can be reduced to integrals over Γ:

$$\int_\Omega f(r,\theta,z) = 2\pi \int_\Gamma r f(r,z)\,. \tag{2.41}$$

For spherically symmetric functions, their integrals over Ω can be reduced to integrals over R:

$$\int_\Omega f(r,\theta,\phi) = 4\pi \int_R r^2 f(r)\,. \tag{2.42}$$

In order to properly handle these specific symmetries in 2D and 1D computational domains, we need to consider not only scalar fields but also vector fields as well as their gradients in each reduced coordinate system. A vector field v, defined on the axisymmetric domain Ω, is called axisymmetric if

$$v = v_r(r,z)\vec{e}_r + v_\theta(r,z)\vec{e}_\theta + v_z(r,z)\vec{e}_z\,,$$

i.e. if v remains invariant under arbitrary rotation around the axis $r = 0$. The gradient operator in cylindrical coordinates is given by

$$\nabla_{rz} f = \frac{df}{dr}\vec{e}_r + \frac{1}{r}\frac{df}{d\theta}\vec{e}_\theta + \frac{df}{dz}\vec{e}_z\,. \tag{2.43}$$

Therefore, $\nabla_{rz} f$ is axisymmetric if and only if f is. In this case, the formula simplifies to

$$\nabla_{rz} f = \frac{df}{dr}\vec{e}_r + \frac{df}{dz}\vec{e}_z\,,$$

which is just the regular 2D gradient in Γ. A vector field

$$v = v_r\vec{e}_r + v_\theta\vec{e}_\theta + v_\phi\vec{e}_\phi$$

defined on the spherically symmetric domain Ω, is called spherically symmetric if

$$v = v_r(r)\vec{e}_r\,.$$

The gradient operator in spherical coordinates is given by

$$\nabla_{r\theta\phi} f = \frac{df}{dr}\vec{e}_r + \frac{1}{r}\frac{df}{d\theta}\vec{e}_\theta + \frac{1}{r\sin\theta}\frac{df}{d\phi}\vec{e}_\phi\,. \tag{2.44}$$

Therefore, $\nabla_{r\theta\phi} f$ is spherically symmetric if and only if f is. In this case, the formula simplifies to

$$\nabla_{r\theta\phi} f = \frac{df}{dr}\vec{e}_r\,,$$

which is just the regular 1D derivative in R.

Using the above definitions as well as several other properties related to vector gradients (i.e. tensor fields, see [10] for details), we can write the 3D momentum conservation equation of (2.1) in semi-discrete form as:

$$\begin{aligned} c\int_{\Gamma(t)} \alpha\rho\frac{d\vec{v}}{dt}\cdot\vec{w} = c\int_{\Gamma(t)} \alpha p\nabla\cdot\vec{w} + c\int_{\Gamma(t)} \beta p w_r \\ - c\int_{\Gamma(t)} \alpha\mu\nabla\vec{v} : \nabla\vec{w} - c\int_{\Gamma(t)} \gamma\mu v_r w_r \end{aligned}\,, \tag{2.45}$$

where Γ defines the reduced 2D cut of the general 3D domain.

Both the axisymmetric (axi) and spherically symmetric (sph) cases can be written in the form (2.45) by choosing the coefficients as follows:

$$c_{axi} = 2\pi, \quad \alpha_{axi} = r, \quad \beta_{axi} = 1, \quad \gamma_{axi} = \frac{1}{r}$$

and

$$c_{sph} = 4\pi, \quad \alpha_{sph} = r^2, \quad \beta_{sph} = 2r, \quad \gamma_{sph} = 2\,.$$

This means that problems with either cylindrical or spherical symmetry can be solved on reduced computational meshes as shown in Figure 2.12 by adjusting the c, α, β and γ weights.

2.3 High-Order finite difference multi-material Eulerian hydrodynamics

In contrast to ALE methods, direct Eulerian methods (which solve the conservation laws of continuum mechanics in a fixed frame of reference) offer unique capabilities and features for capturing complex flows which are dominated by shocks, material interfaces, and vortical motion/turbulence. As such, MARBL provides a high-order direct Eulerian hydrodynamics option based

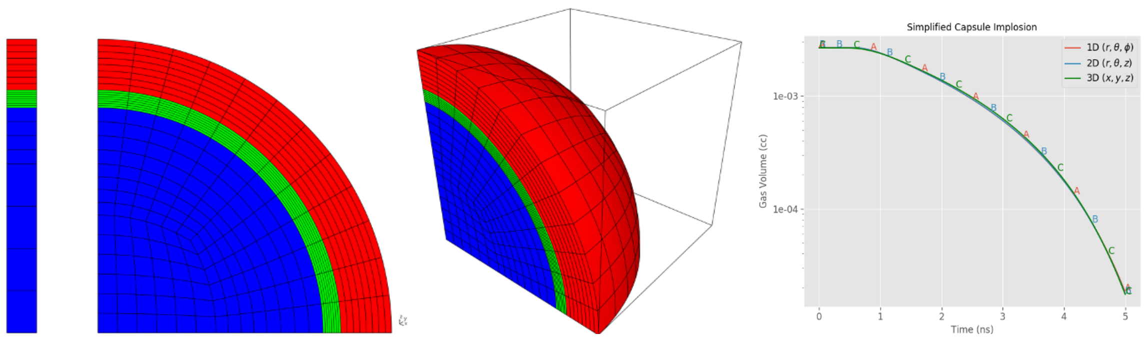


Figure 2.12: Examples of 1D spherical, 2D axisymmetric and 3D Cartesian meshes of a simplified ICF capsule in MARBL. The plot on the left depicts the volume of the blue region under shock compression for each case, indicating that all three symmetries/meshes produce the same volume history as a function of time.

on the Miranda code. Problems involving mixing, for example, greatly benefit from high-order and high-resolution numerical schemes which have little numerical dissipation and allow for accurate evolution of turbulent fields. Furthermore, mesh generation is greatly simplified for Cartesian meshes and features can be resolved using adaptive mesh refinement (AMR) rather than by conformally zoned meshes. There are trade-offs when using a Cartesian mesh, but for a class of problems relevant to the national labs, this approach is advantageous.

The equations of motion for hydrodynamic motion of N-species (or materials) of compressible fluids is given by:

$$\text{Momentum Conservation:}\quad \frac{\partial(\rho v)}{\partial t} = -\nabla\cdot\left(\rho v v^T - \sigma\right) + \rho f\,, \tag{2.46}$$

$$\text{Species Mass Conservation:}\quad \frac{\partial(\rho Y_i)}{\partial t} = -\nabla\cdot\left(\rho Y_i v + J_i\right)\,, \tag{2.47}$$

$$\text{Total Energy Conservation:}\quad \frac{\partial E}{\partial t} = -\nabla\cdot\left(E v - \sigma\cdot v - q\right) + \rho v f\,, \tag{2.48}$$

$$\text{Total Mass Conservation:}\quad \rho = \sum_i^N (\rho Y_i) \tag{2.49}$$

with fluid density ρ, velocity vector v, species mass fraction for species-i Y_i, species mass flux J_i, body force vector f, total energy given by $E = e + \frac{1}{2}\rho v\cdot v$ where e is the internal energy, heat flux vector q, and stress tensor given by $\sigma = -pI + \tau$, where τ is the stress tensor, I is the identity matrix, and p is the pressure. For Newtonian fluids, τ is given by:

$$\tau = \mu S + (\beta - \frac{2}{3}\mu)(\nabla\cdot v)I \;\text{ with} \tag{2.50}$$

$$S = \frac{1}{2}(\nabla v + \nabla v^T) \tag{2.51}$$

where μ is the kinematic viscosity, and β is the bulk viscosity.

In general $p = f(\rho, e)$ is assumed as a constitutive closure for the equation of state of the thermodynamic variables. The species mass flux model J_i can be quite complex, however, for demonstration of the form of the artificial interface capturing scheme we use, it is assumed to be compute as

$$J_i = -\rho\left(D_i\nabla Y_i - Y_i\sum_{j=1}^{N} D_j\nabla Y_j\right) \tag{2.52}$$

where D_i is the Fickian diffusion coefficient of species-i for N total species. In addition, the heat flux is computed as

$$q = -\kappa\nabla T + \sum_{k=1}^{N} h_k J_k, \tag{2.53}$$

where κ in the thermal conductivity, and T and h is the material temperature and enthalpy which can be computed (like p) as a function, e.g. $T = f(\rho, e)$ and $h_k = f(\rho, e)$.

2.3.1 A 5-stage RK4 Scheme

The governing equations of motion are advanced in time via a five-step fourth-order Runge-Kutta method and writing the PDEs from equations 2.46-2.48 generally as $\dot{\Phi} = F$.

$$Q^{\eta} = \Delta t F^{\eta-1} + A^{\eta} Q^{\eta-1} \tag{2.54}$$

$$\Phi^{\eta} = \Phi^{\eta-1} + B^{\eta} Q^{\eta} \qquad \eta = 1,...,5 \tag{2.55}$$

The values for A^{η} and B^{η} are given in [25] and [26], with $A^1 = 0$. This scheme has broad stability properties for the convection and viscous terms of the equations of motion.

2.3.2 Compact finite difference (Pade) schemes

All first derivatives used for the divergence and gradient operators in the equations of motion are computed using a 10th order compact finite difference scheme on structured, Cartesian mesh. Partial derivatives in x, y, z map to logical grid directions and index space. Here, we define the form of these derivatives along a particular direction (i) but the operator applies equally to all directions.

Give variable f on the computational grid, its derivative is approximated by:

$$\sum_{p=0}^{2} \alpha_p (f'_{i+p} + f'_{i-p}) = \sum_{e=1}^{3} b_e \frac{(f_{i+e} - f_{i-e})}{2e\Delta}. \tag{2.56}$$

For each derivative call, this approximation yields a system of equations for all the derivatives along a particular global index line. The linear system has the generic form for $Ax = b$, where x are the array of unknown derivatives, f'. Given the left-hand side of equation 2.56, A becomes a banded, penta-diagonal matrix which needs to be solved globally for the entire domain. Details on the implementation and performance of this operator will be discussion in Section 4.3.1.

2.3.3 Artificial Fluid Properties (Hyper-diffusivities) for shock/interface capturing and SGS viscosity

Given the non–dissipative nature of the derivative operator, explicit numerical diffusion is required to help stabilize regions of the flow that become under-resolved either from sharp discontinuities or from small scale turbulent structure. The equations of motion include physical diffusivities for shocks, material interfaces, thermal contacts, and vorticity in the form of bulk viscosity, species diffusivity, thermal conductivity, and kinematic viscosity.

The method of artificial fluid properties large-eddy simulation (AFLES) proposed by Cook [25] is utilized here, where artificial/numerical diffusivities are added to their physical counter-parts. Therefore, we have:

$$\beta = \beta_f + \beta^* \tag{2.57}$$

$$D_i = D_{i,f} + D_i^* \tag{2.58}$$

$$\kappa = \kappa_f + \kappa^* \tag{2.59}$$

$$\mu = \mu_f + \mu^* \tag{2.60}$$

Here, the sub-script f denotes the physical property, where the super-script $*$ denotes the artificial fluid property. The form of the artificial property can be written generally as ψ^*, with:

$$\psi^* = C_\psi \overline{f(U)G(\phi)\Delta^2} \tag{2.61}$$

where the over-bar notation indicates application of a truncated-Gaussian filter, and Δx is the local mesh spacing. $G(\phi)$ represents the application of an eighth-derivative operator such that for a scalar ϕ,

$$G(\phi) = \max\left(\left| \frac{\partial^8 \phi}{\partial x^8} \Delta x^8 \right|, \left| \frac{\partial^8 \phi}{\partial y^8} \Delta y^8 \right|, \left| \frac{\partial^8 \phi}{\partial z^8} \Delta z^8 \right| \right) \tag{2.62}$$

and for a vector ϕ_i,

$$G(\phi) = \max\left(G(\phi_x), G(\phi_y), G(\phi_z)\right). \tag{2.63}$$

Table 2.1: Summary of AFLES terms used for artificial dissipation.

ψ^*	C_ψ	$f(U)$	ϕ
β^*	7.0×10^{-2}	ρ	$\nabla \cdot v$
D_k^*	1.0×10^{-2}	$\frac{1}{\Delta t}$	Y_k
κ^*	1.0×10^{-3}	$\frac{\rho c_v}{T \Delta t}$	T
μ^*	1.0×10^{-4}	$\frac{\rho}{\Delta}$	v_i

2.4 3T + Thermonuclear Burn

The modeling of Inertial Confinement Fusion (ICF) experiments is an important target application for the MARBL code because of the ongoing interest at LLNL to achieve ignition at the National Ignition Facility (NIF) as well as related experiments at other Department of Energy (DOE) facilities. The ICF concept relies on the inertial mass of a burning plasma to hold it together long enough to achieve an appreciable energy release due to thermonuclear burn. An estimate of the time it takes for a burning plasma to fly apart is given by the spatial extent of the burning plasma divided by the sound speed, typically a tenth of a nanosecond. The main energy producing thermonuclear reaction for energy applications is this reaction,

$$D + T = n + \alpha, \tag{2.64}$$

which releases 17.6 MeV per reaction. To achieve significant energy release due to thermonuclear burn at these inertial time scales, deuterium/tritium fuels must be compressed $\sim$ 1000 times solid density. Consequently, the necessity of a fast thermonuclear burn rate means the assumption of a single matter temperature in thermal equilibrium is no longer valid. The matter temperature must now be subdivided into ions and electrons, with each separate species in their own thermal equilibrium, which couple at a rate determined by an electron-ion coupling term. Because the concept of a three temperature plasma, or 3T, and thermonuclear burn are so tightly intertwined we describe the validation of these physics capabilities as a whole describing the 3T fields associated with: electrons, ions, and radiation. Thermonuclear energy source terms deposit energy into the electron and ion field on a timescale that is fast or comparable to the time scale at which electrons and ions equilibrate and the radiation field is coupled to the system only through the much lighter electrons.

A modular TN reaction package, SELENE[27], was developed and integrated into MARBL to compute the source terms in the 3T equations. In addition to interfacing with the TDF library to obtain "sigma-vbar" data needed to compute thermonuclear burn reaction rates, it also contains the necessary plasma physics models necessary to partition the amount of energy that is deposited in the electrons and ions. The plasma physic models in SELENE can be used in other areas, such as electron-ion coupling, to provide an overall consistent picture of plasma physics (i.e. consistent coulomb log assumptions, etc...). As part of a broader code development strategy, the SELENE modular physics package is already implemented in other ICF capable codes at LLNL and it is also GPU enabled.

2.4.1 Simplified 3T Equations

The physics described in this section takes place within a cell or quadrature point with no dependence on neighboring cells (a so called "0D" physics model). Therefore, for the purposes of introducing the relevant concepts for this section, the mass flux and transport terms described in the previous sections associated with hydrodynamics and radiation transport are not considered here. We also note that there is a material dependence that is suppressed, in general, each material within a cell would have a 3T equation associated with it. With those simplifications the equations which represent the three temperature field can be written in terms of a radiation, total, and electron temperature. Note, some codes are organized in terms of a radiation, ion, and electron energy fields, but the way it is written here the 2T equations can be recovered simply by eliminating the last equation. In the present formulation, the ion energy is computed by subtracting the electron energy from the total energy.

To verify that MARBL is doing 3T physics correctly we derive here a set of analytic benchmark solutions for a 0D infinite medium problem and verify that MARBL reproduces those solutions in both the Lagrangian (or ALE, referred to as MARBL-lag) and direct Eulerian (MARBL-eul) cases previously described. These test problems are part of the MARBL test suite and are used frequently to asses correctness of new code features. In addition, these test problems can be used to verify order of convergence of numerical algorithms.

We start by writing down Eqns 2.65 through 2.67 which describe the time evolution of the radiation, material (total), and electron fields.

$$\frac{dE_r}{dt} = c\sigma_p\left(B(T_e) - E_r\right) \tag{2.65}$$

$$\rho\frac{de}{dt} = c\sigma_p\left(E - B(T_e)\right) - Q\frac{dn}{dt} \tag{2.66}$$

$$\rho\frac{de_e}{dt} = \sigma_p\left(E - B(T_e)\right) + \omega_{ei}\rho\left(T_e - T_i\right) - \gamma Q\frac{Dn}{Dt} \tag{2.67}$$

In the first equation E_r $[erg/cc]$ is the radiation energy density, c is the speed of light $[cm/s]$, and σ_p is absorption opacity $[cm^{-1}]$. $B(T_e) = (4\sigma/c)T_e^4$ $[erg/cc]$ is the blackbody function where σ is the Stefan-Boltzmann constant.

In the next two equations e and e_e $[erg/g/cc]$ are total and electron specific energy respectively with the density given by ρ $[g/cc]$. The thermo-nuclear energy source term is described by Q $(erg/\#)$, the energy of a TN reaction. n, the reaction density $(\#/cc)$, and *gamma* the split between the electron and ion deposition. Finally, ω_{ei} $(erg/keV/g/s)$ is the electron-ion coupling coefficient.

We begin by transforming these equations into the familiar 3T equations. We first define radiation energy density to be $E_r = (4\sigma/c)T_r^4$ and the Planck opacity by $\kappa_p = \sigma_p/\rho$ $[cm^2/g]$,

$$\frac{dT_r^4}{dt} = c\rho\kappa\left(T_e^4 - T_r^4\right) \tag{2.68}$$

Similarly the other two equation can be transformed by noting $de/dt = de_e/dt + de_i/dt$ and subtracting 2.67 from 2.66,

$$\rho\frac{de_i}{dt} = -\omega_{ei}\rho\left(T_e - T_i\right) - (1-\gamma)Q\frac{dn}{dt} \tag{2.69}$$

We transform the electron and ion specific energy into electron and ion temperatures through the relations $de_e/dt = C_{ve}dT_e/dt$ and $de_i/dt = C_{vi}dT_i/dt$. C_{ve} and C_{vi} are defined to be the electron and ion specific heats respectively, with units of $[erg/keV]$. Finally, if we assume an ideal case, the electron-ion coupling coefficient can be written $\omega_{ei} = C_{ve}K_{ie}$, where K_{ie} is the electron-ion equilibration rate and has units $[s^{-1}]$.

$$\rho C_{ve}\frac{dT_e}{dt} = ac\rho\kappa\left(T_r^4 - T_e^4\right) + \rho C_{ve}K_{ie}\left(T_i - T_e\right) - \gamma Q\frac{dn}{dt} \tag{2.70}$$

$$\rho C_{vi}\frac{dT_i}{dt} = \rho C_{ve}K_{ie}\left(T_e - T_i\right) - (1-\gamma)\,Q\frac{dn}{dt} \tag{2.71}$$

2.4.2 Linearization of 3T equations

We next proceed to linearize the 3T equations to make them amenable to analytic solution. First we define,

$$\Phi = T_r^4 \tag{2.72}$$

$$X = T_e^4 \tag{2.73}$$

$$U = T_i^4 \tag{2.74}$$

substituting these into Eqs. 2.68, 2.70, and 2.71 gives,

$$\frac{d\Phi}{dt} = c\rho\kappa (X - \Phi) \tag{2.75}$$
$$\frac{\rho C_{ve}}{4T_e^3}\frac{dX}{dt} = ac\rho\kappa \left(T_r^4 - T_e^4\right) + \rho C_{ve} K_{ie} (T_i - T_e) - \gamma Q \frac{dn}{dt} \tag{2.76}$$
$$\frac{\rho C_{vi}}{4T_i^3}\frac{dU}{dt} = \rho C_{ve} K_{ie} (T_e - T_i) - (1-\gamma) Q \frac{dn}{dt} \tag{2.77}$$

Our first assumption is that C_{ve} and C_{vi} have the analytic form,

$$C_{ve} = C_e \left(\frac{T_e}{T_{e0}}\right)^3 \tag{2.78}$$

$$C_{vi} = C_i \left(\frac{T_i}{T_{i0}}\right)^3 \tag{2.79}$$

where T_{e0} and T_{i0} are the initial electron and ion temperatures respectively. C_e and C_i are parameters with units of $[erg/keV^4]$. Plugging those in gives,

$$\frac{d\Phi}{dt} = c\rho\kappa (X - \Phi) \tag{2.80}$$
$$\frac{dX}{dt} = \frac{4ac\kappa T_{e0}^3}{C_e}\left(T_r^4 - T_e^4\right) + 4T_e^3 K_{ie} (T_i - T_e) - \frac{4T_{e0}^3}{\rho C_e}\gamma Q \frac{Dn}{Dt} \tag{2.81}$$
$$\frac{dU}{dt} = 4T_e^3 \frac{C_e T_{i0}^3}{C_i T_{e0}^3} K_{ie} (T_e - T_i) - \frac{4T_{i0}^3}{\rho C_i}(1-\gamma) Q \frac{Dn}{Dt} \tag{2.82}$$

Our second assumption is that K_{ie} has the form,

$$K_{ie} = \frac{\alpha}{4}\frac{(T_e + T_i)\left(T_e^2 + T_i^2\right)}{T_e^3} \tag{2.83}$$

and noting the math identity,

$$(T_e + T_i)\left(T_e^2 + T_i^2\right)(T_i - T_e) = \left(T_i^4 - T_e^4\right) \tag{2.84}$$

we find,

$$\frac{d\Phi}{dt} = c\rho\kappa (X - \Phi) \tag{2.85}$$
$$\frac{dX}{dt} = \frac{4ac\kappa T_{e0}^3}{C_e}(\Phi - X) + \alpha (U - X) - \frac{4T_{e0}^3}{\rho C_e}\gamma Q \frac{Dn}{Dt} \tag{2.86}$$
$$\frac{dU}{dt} = \frac{C_e T_{i0}^3}{C_i T_{e0}^3}\alpha (X - U) - \frac{4T_{i0}^3}{\rho C_i}(1-\gamma) Q \frac{Dn}{Dt} \tag{2.87}$$

It now convenient to define the following parameters to cast the equations in a simple form

$$\beta = c\rho\kappa \tag{2.88}$$

$$D = \frac{4ac\kappa T_{e0}^3}{C_e} \tag{2.89}$$

$$r = \frac{C_e T_{i0}^3}{C_i T_{e0}^3} \tag{2.90}$$

$$q_e = \frac{4T_{e0}^3}{\rho C_e}\gamma Q \tag{2.91}$$

$$q_i = \frac{4T_{i0}^3}{\rho C_i}(1-\gamma)\,Q \tag{2.92}$$

Then the 3T equations take their final linear form,

$$\frac{d\Phi}{dt} = \beta\,(X-\Phi) \tag{2.93}$$
$$\frac{dX}{dt} = D\,(\Phi - X) + \alpha\,(U-X) - q_e\frac{Dn}{Dt} \tag{2.94}$$
$$\frac{dU}{dt} = r\alpha\,(X-U) - q_i\frac{Dn}{Dt} \tag{2.95}$$

2.4.3 Solving the Linearized 3T Equations

The linearized 3T equations, Eqs. 2.93, 2.94, and 2.95 can be written in the following matrix form:

$$\begin{bmatrix}\Phi' \\ X' \\ U'\end{bmatrix} = \begin{bmatrix}-\beta & \beta & 0 \\ D & -D-\alpha & \alpha \\ 0 & r\alpha & -r\alpha\end{bmatrix}\begin{bmatrix}\Phi \\ X \\ U\end{bmatrix} + \begin{bmatrix}0 \\ -q_e\frac{Dn}{Dt} \\ -q_i\frac{Dn}{Dt}\end{bmatrix} \tag{2.96}$$

First, we find the eigenvalue and corresponding eigenvectors of the homogeneous version of Eqn 2.96:

$$\begin{bmatrix}\Phi' \\ X' \\ U'\end{bmatrix} = \begin{bmatrix}-\beta & \beta & 0 \\ D & -D-\alpha & \alpha \\ 0 & r\alpha & -r\alpha\end{bmatrix}\begin{bmatrix}\Phi \\ X \\ U\end{bmatrix} \tag{2.97}$$

The eigenvalues are:

$$\lambda_1 = 0 \tag{2.98}$$

$$\lambda_2 = \frac{1}{2}\left(-\alpha-\beta-D-\alpha r-\sqrt{(\alpha+\beta+D+\alpha r)^2-4\,(\alpha\beta+\alpha\beta r+\alpha D r)}\right) \tag{2.99}$$

$$\lambda_3 = \frac{1}{2}\left(-\alpha-\beta-D-\alpha r+\sqrt{(\alpha+\beta+D+\alpha r)^2-4(\alpha\beta+\alpha\beta r+\alpha D r)}\right) \tag{2.100}$$

To simply Eqns 2.99 and 2.100 we define the following substitutions:

$$R = \alpha+\beta+D+\alpha r \tag{2.101}$$

$$T = \alpha\beta+\alpha\beta r+\alpha D r \tag{2.102}$$

Inserting Eqns 2.101 and 2.102 into Eqns 2.99 and 2.100:

$$\lambda_2 = -\frac{1}{2}\left(R+\sqrt{R^2-4T}\right) \tag{2.103}$$

$$\lambda_3 = -\frac{1}{2}\left(R-\sqrt{R^2-4T}\right) \tag{2.104}$$

We further simplify the eigenvalues by defining the following substitutions:

$$g = \frac{1}{2}\left(R-\sqrt{R^2-4T}\right) \tag{2.105}$$

$$h = \frac{1}{2}\left(R+\sqrt{R^2-4T}\right) \tag{2.106}$$

Inserting Eqns 2.105 and 2.106 into Eqns 2.103 and 2.108:

$$\lambda_2 = -h \tag{2.107}$$

$$\lambda_3 = -g \tag{2.108}$$

Substituting in Eqns 2.101, 2.102, 2.105 and 2.106 into the results, the eigenvectors are:

$$v_1 = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} \tag{2.109}$$

$$v_2 = \begin{bmatrix} 1+\frac{\beta h-T}{\alpha D r} \\ 1-\frac{h}{\alpha r} \\ 1 \end{bmatrix} \tag{2.110}$$

$$v_3 = \begin{bmatrix} 1+\frac{\beta g-T}{\alpha D r} \\ 1-\frac{g}{\alpha r} \\ 1 \end{bmatrix} \tag{2.111}$$

The general solution to the homogeneous system in Eqn 2.97 is:

$$\begin{bmatrix} \Phi \\ X \\ U \end{bmatrix} = C_1 \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} + C_2 \begin{bmatrix} 1+\frac{\beta h - T}{\alpha D r} \\ 1-\frac{h}{\alpha r} \\ 1 \end{bmatrix} e^{-ht} + C_3 \begin{bmatrix} 1+\frac{\beta g - T}{\alpha D r} \\ 1-\frac{g}{\alpha r} \\ 1 \end{bmatrix} e^{-gt} \tag{2.112}$$

This produces the fundamental matrix:

$$M(t) = \begin{bmatrix} 1 & \left(1+\frac{\beta h - T}{\alpha D r}\right) e^{-ht} & \left(1+\frac{\beta g - T}{\alpha D r}\right) e^{-gt} \\ 1 & \left(1-\frac{h}{\alpha r}\right) e^{-ht} & \left(1-\frac{g}{\alpha r}\right) e^{-gt} \\ 1 & e^{-ht} & e^{-gt} \end{bmatrix} \tag{2.113}$$

Now we have to find the particular solution associated with our electron and ion sources given in Eqns 2.91 and 2.92, and represented by the source vector:

$$b = \begin{bmatrix} 0 \\ -q_e \frac{Dn}{Dt} \\ -q_i \frac{Dn}{Dt} \end{bmatrix} \tag{2.114}$$

For this derivation, we assume a fixed reactivity resulting in the following neutron density function:

$$n(t) = \frac{n_0}{1+\sigma n_0 t} \tag{2.115}$$

Inserting Eqn 2.115 into 2.114 produces:

$$b = \begin{bmatrix} 0 \\ q_e \frac{n_0^2 \sigma}{(1+n_0 \sigma t)^2} \\ q_i \frac{n_0^2 \sigma}{(1+n_0 \sigma t)^2} \end{bmatrix} \tag{2.116}$$

The particular solution $M \int M^{-1} b$ from Mathematica after some simplification becomes:

$$p = \begin{bmatrix} \frac{E(g,t)F(h)g(\beta g + \alpha D r - T) - E(h,t)F(g)h(\beta h + \alpha D r - T)}{D(g-h)\sigma T} \\ -n(t)\, q_e - \frac{E(g,t)F(h)g(g-\alpha r) - E(h,t)F(g)h(h-\alpha r)}{(g-h)\sigma T} \\ -n(t)\, q_i + \frac{E(g,t)F(h)\alpha g r - E(h,t)F(g)\alpha h r}{(g-h)\sigma T} \end{bmatrix} \tag{2.117}$$

where:

$$E(a,t) = e^{-a(\nu+t)} \mathrm{EI}(a(\nu+t)) \tag{2.118}$$

$$F(a) = \beta a (q_e - q_i) - D a q_i + (-q_e + q_i) T \tag{2.119}$$

$$\nu = \frac{1}{n_0 \sigma} \tag{2.120}$$

Note that $n(t)$ is given in Eqn 2.115 and EI denotes the exponential integral. The general solution to the non-homogeneous system becomes:

$$\begin{aligned}\begin{bmatrix}\Phi\\X\\U\end{bmatrix} &= C_1\begin{bmatrix}1\\1\\1\end{bmatrix} + C_2\begin{bmatrix}1+\frac{\beta h-T}{\alpha Dr}\\1-\frac{h}{\alpha r}\\1\end{bmatrix}e^{-ht} + C_3\begin{bmatrix}1+\frac{\beta g-T}{\alpha Dr}\\1-\frac{g}{\alpha r}\\1\end{bmatrix}e^{-gt}\\ &+ \begin{bmatrix}\frac{E(g,t)F(h)g(\beta g+\alpha Dr-T)-E(h,t)F(g)h(\beta h+\alpha Dr-T)}{D(g-h)\sigma T}\\-n(t)\,q_e-\frac{E(g,t)F(h)g(g-\alpha r)-E(h,t)F(g)h(h-\alpha r)}{(g-h)\sigma T}\\-n(t)\,q_i+\frac{E(g,t)F(h)\alpha gr-E(h,t)F(g)\alpha hr}{(g-h)\sigma T}\end{bmatrix}\end{aligned} \tag{2.121}$$

The final step is to use the following initial conditions to determine the three coefficients in Eqn 2.121:

$$\Phi(t=0)=\Phi_0 \tag{2.122}$$

$$X(t=0)=X_0 \tag{2.123}$$

$$U(t=0)=U_0 \tag{2.124}$$

The coefficients are not shown here for brevity, but inserting the coefficients back into Eqn 2.121 we arrive at the following analytic solutions:

$$\begin{aligned}\Phi(t) = M_1&\\ &+M_2 f_1(h)\,e^{-gt}\\ &-M_3 f_1(g)\,e^{-ht}\\ &+\frac{g}{\sigma}M_2 f_2(h)\,e^{-g(\nu+t)}\left(EI(g\nu)-EI(g(\nu+t))\right)\\ &-\frac{h}{\sigma}M_3 f_2(g)\,e^{-h(\nu+t)}\left(EI(h\nu)-EI(h(\nu+t))\right)\end{aligned} \tag{2.125}$$

$$\begin{aligned}X(t) = -q_e n(t)&\\ &+M_4\\ &-M_5 f_1(h)\,e^{-gt}\\ &+M_6 f_1(g)\,e^{-ht}\\ &-\frac{g}{\sigma}M_5 f_2(h)\,e^{-g(\nu+t)}\left(EI(g\nu)-EI(g(\nu+t))\right)\\ &+\frac{h}{\sigma}M_6 f_2(g)\,e^{-h(\nu+t)}\left(EI(h\nu)-EI(h(\nu+t))\right)\end{aligned} \tag{2.126}$$

$$\begin{aligned}U(t) = &-q_i n(t)\\ &+M_4\\ &+M_7 f_1(h) e^{-gt}\\ &-M_7 f_1(g) e^{-ht}\\ &+\frac{g}{\sigma} M_7 f_2(h) e^{-g(\nu+t)}\left(EI(g\nu)-EI(g(\nu+t))\right)\\ &-\frac{h}{\sigma} M_7 f_2(g) e^{-h(\nu+t)}\left(EI(h\nu)-EI(h(\nu+t))\right)\end{aligned} \tag{2.127}$$

where the following constants and functions were defined:

$$M_1 = \frac{\alpha}{T}\left(\beta n_0 (q_i + q_e r) + U_0\beta + rD\Phi_0 + r\beta X_0\right) \tag{2.128}$$

$$M_2 = \frac{\beta g + \alpha D r - T}{D(g-h)T} \tag{2.129}$$

$$M_3 = \frac{\beta h + \alpha D r - T}{D(g-h)T} \tag{2.130}$$

$$M_4 = n_0 q_i + \frac{\alpha}{T}\left(U_0\beta + rD\Phi_0 + r\beta X_0 + n_0 r\left(\beta q_e - q_i(\beta + D)\right)\right) \tag{2.131}$$

$$M_5 = \frac{g - \alpha r}{(g-h)T} \tag{2.132}$$

$$M_6 = \frac{h - \alpha r}{(g-h)T} \tag{2.133}$$

$$M_7 = \frac{\alpha r}{(g-h)T} \tag{2.134}$$

$$f_1(a) = Da(\Phi_0 - n_0 q_i - U_0) + (\beta a - T)\left(n_0(q_e - q_i) - U_0 + X_0\right) \tag{2.135}$$

$$f_2(a) = Daq_i + \beta a(-q_e + q_i) + (q_e - q_i)T \tag{2.136}$$

2.4.4 Define Constants

In this section we define the constants required by the analytic solution. First we need to define the initial conditions:

- n_0, T_{r0}, T_{e0}, and T_{i0}.

Then we need to define the energy released per fusion reaction and the reaction cross section:

- Q_{MeV} and σ.

Then we need to define the constants which determine the degree of coupling between the three temperature fields:

- κ, γ, and α.

Finally, we have a list of fixed constants which have been matched with Selene source code (selene/src/material/Constants.hh):

$$N_a = 6.02214179 \times 10^{23} \tag{2.137}$$

$$c = 2.99792458 \times 10^{10} \tag{2.138}$$

$$m_d = 2.0141018 \tag{2.139}$$

$$\rho = \frac{m_d n_0}{N_a} \tag{2.140}$$

$$e_c = 1.602176487 \times 10^{-19} \tag{2.141}$$

$$\text{evtoerg} = e_c \text{evtoerg} \tag{2.142}$$

$$\text{mevtoerg} = 10^6 \text{evtoerg} \tag{2.143}$$

$$Q_{\text{erg}} = Q_{\text{MeV}} \text{mevtoerg} \tag{2.144}$$

$$\sigma_{sb} = 1.02830116 \times 10^{24} \tag{2.145}$$

$$a = \frac{4\sigma_{sb}}{c} \tag{2.146}$$

2.4.5 Limitations

There are limitations to the selected parameters based on our derivation:

- $\alpha > 0$ is required i.e., there must be some level of ion-electron coupling. If $\alpha = 0$ then Eqn. 2.102 will produce $T = 0$, which subsequently results in a divided-by-zero error for Eqns 2.128 through 2.134.
- $\kappa > 0$ is required i.e., there must be some level of electron-radiation coupling. If $\kappa = 0$ then Eqn. 2.102 will produce $T = 0$ since both β and D are zero. This results in a divided-by-zero error for Eqns 2.128 through 2.134.

Parameter	Value
ρ	2.0141018 g/cc
T_{e0}	1.0 keV
T_{i0}	1.0 keV
T_{r0}	1.0 keV
κ	0.0
C_{vi}	$5.2329245226131531 \times 10^14$
K_{ie}	0.0
$\overline{\sigma v}$	1.0×10^{17} cm^3/s
Q	4.0 MeV

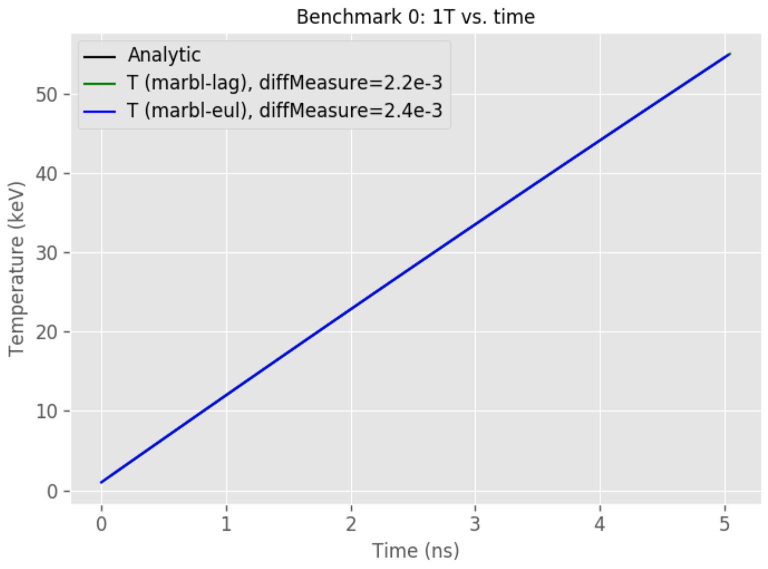


Figure 2.13: Comparison of material temperature vs. time between analytic (black), MARBL-lag (green), and MARBL-eul (blue) for benchmark 0.

2.4.6 Selene Benchmarks

Benchmark 0

First we return to 2.71 consider the case where there is no coupling between the electrons and ions (e.g. $K_{ie} = 0$) and all of the energy is sourced into the ions (e.g. $\gamma = 0.0$)

$$\rho C_{vi} \frac{dT_i}{dt} = Q \frac{dn}{dt} \tag{2.147}$$

We also consider a gamma law gas (e.g. constant specific heat). Integrating 2.147 gives,

$$\rho C_{vi} \int_{t=0}^{t} \frac{dT_i}{dt} = Q \int_{t=0}^{t} \frac{dn}{dt} \tag{2.148}$$

which using 2.115 integrates to,

$$\rho C_{vi} (T_i - T_0) = Q \left(\frac{n_0}{1 + \sigma n_0 t} - n_0 \right) \tag{2.149}$$

Benchmark 1

Next, we return to our linearized 3T equations 2.95 and the same assumptions,

Parameter	Value
ρ	2.0141018 g/cc
T_{e0}	1.0 keV
T_{i0}	1.0 keV
T_{r0}	1.0 keV
κ	0.0
C_e	$7.1853266962970788 \times 10^2$
C_i	$7.1853266962970788 \times 10^2$
K_{ie}	0.0
$\overline{\sigma v}$	1.0×10^{17} cm^3/s
Q	4.0 MeV

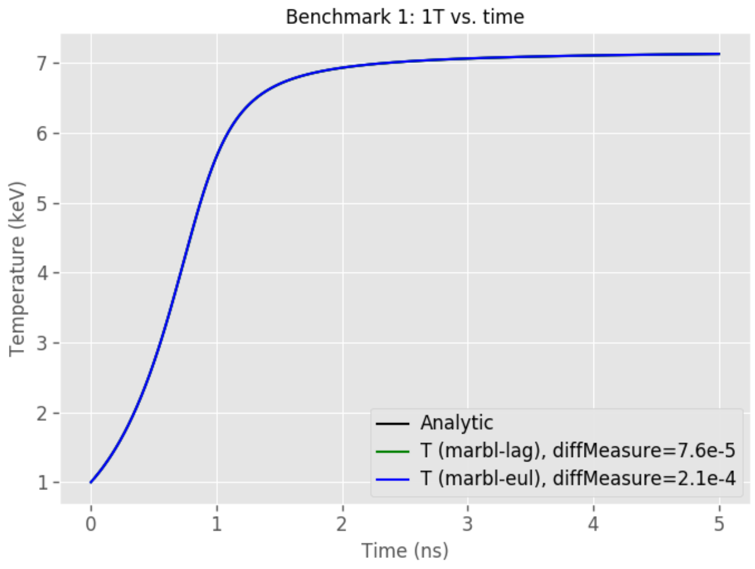


Figure 2.14: Comparison of material temperature vs. time between analytic (black), MARBL-lag (green), and MARBL-eul (blue) for benchmark 1.

$$\frac{dU}{dt} = q_i \frac{Dn}{Dt} \tag{2.150}$$

...which integrates to,

$$(U - U_0) = q_i \left(\frac{n_0}{1 + \sigma n_0 t} - n_0 \right) \tag{2.151}$$

Parameter	Value
ρ	2.0141018 g/cc
T_{e0}	1.0 keV
T_{i0}	10.0 keV
T_{r0}	1.0 keV
κ	0.0
C_e	$7.1798077208840825 \times 10^2$
C_i	$7.1798077208840825 \times 10^2$
K_{ie}	1000.0
$\overline{\sigma v}$	1.0×10^{17} cm^3/s
Q	4.0 MeV

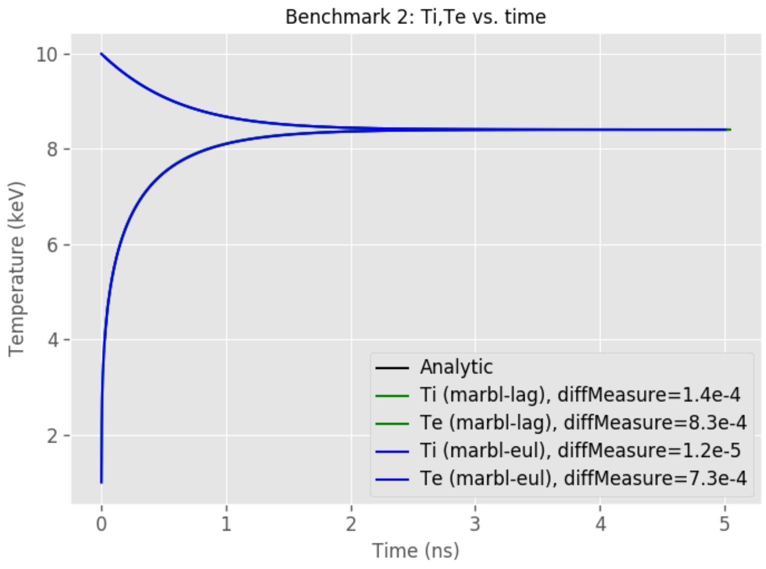


Figure 2.15: Comparison of ion (top) and electron (bottom) temperature vs. time between analytic (black), MARBL-lag (green), and MARBL-eul (blue) for benchmark 2.

Parameter	Value
ρ	0.10078250 g/cc
T_{e0}	0.8 keV
T_{i0}	0.8 keV
T_{r0}	0.1 keV
κ	0.0
C_e	$9.5730769611787762 \times 10^2$
C_i	$9.5730769611787762 \times 10^2$
K_{ie}	0.0
$\overline{\sigma v}$	1.0×10^{17} cm^3/s
Q	4.0 MeV

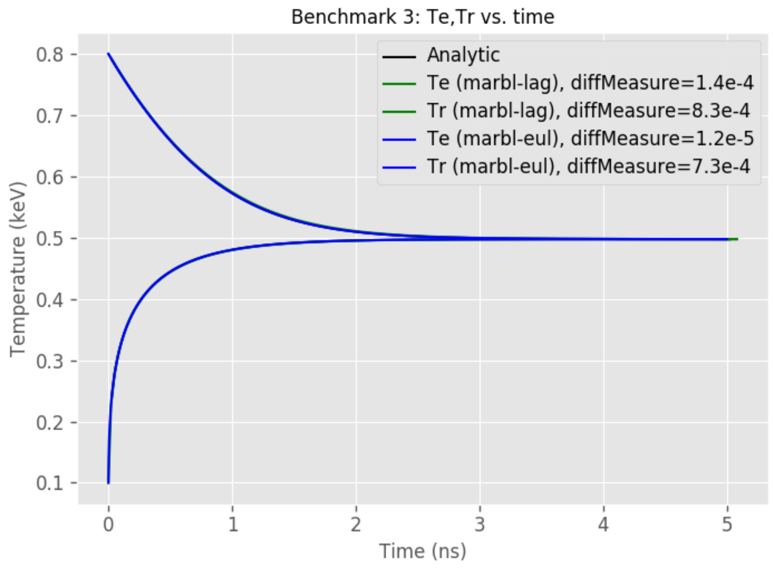


Figure 2.16: Comparison of electron (top) and radiation (bottom) temperature vs. time between analytic (black), MARBL-lag (green), and MARBL-eul (blue) for benchmark 3.

Parameter	**Value**
ρ	20.141018 g/cc
T_{e0}	1.0 keV
T_{i0}	1.0 keV
T_{r0}	1.0 keV
κ	1.0×10^{-4}
C_e	$7.1798077208840825 \times 10^2$
C_i	$7.1798077208840825 \times 10^2$
K_{ie}	1000.0
$\overline{\sigma v}$	1.0×10^{17} cm^3/s
Q	4.0 MeV

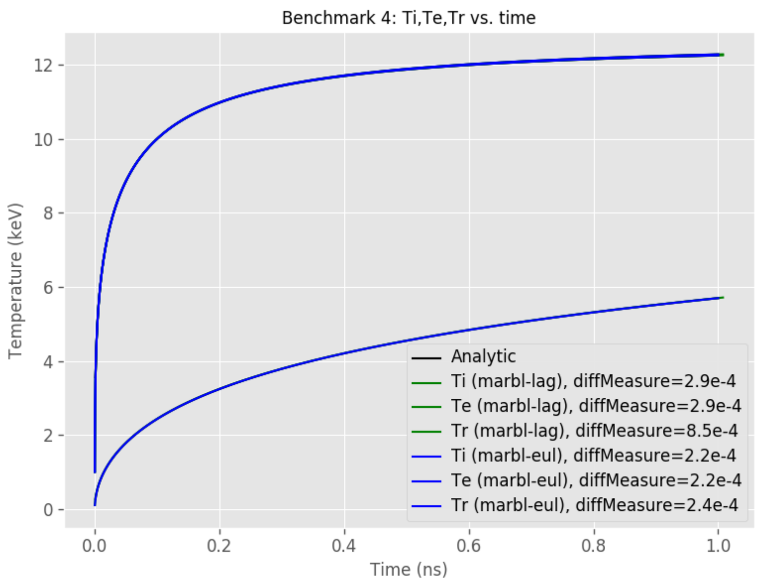


Figure 2.17: Comparison of ion (top), electron (top), and radiation (bottom) temperature vs. time between analytic (black), MARBL-lag (green), and MARBL-eul (blue) for benchmark 4.

Parameter	**Value**
ρ	20.141018 g/cc
T_{e0}	1.0 keV
T_{i0}	1.0 keV
T_{r0}	1.0 keV
κ	1.0
C_e	$7.1798077208840825 \times 10^2$
C_i	$7.1798077208840825 \times 10^2$
K_{ie}	1000.0
$\overline{\sigma v}$	1.0×10^{17} cm^3/s
Q	4.0 MeV

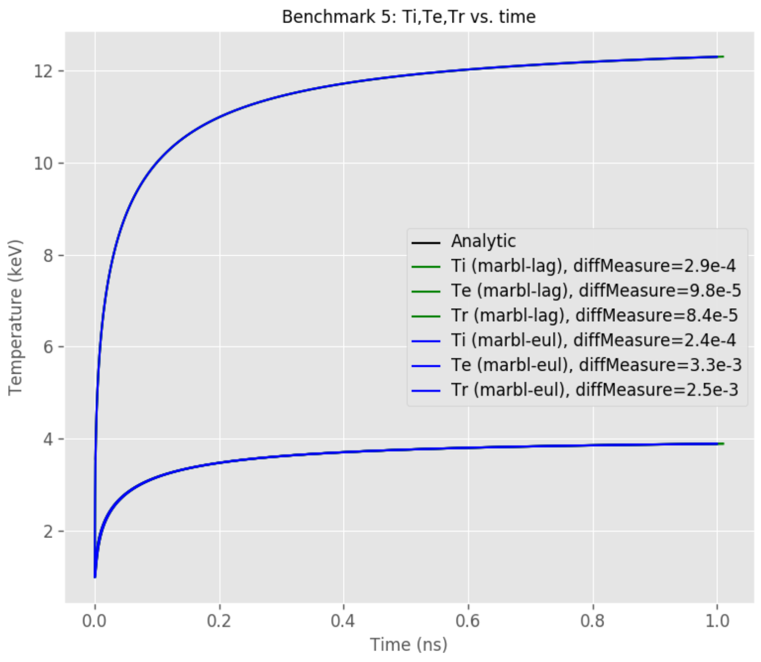


Figure 2.18: Comparison of ion (top), electron (top), and radiation (bottom) temperature vs. time between analytic (black), MARBL-lag (green), and MARBL-eul (blue) for benchmark 5.

Parameter	**Value**
ρ	20.141018 g/cc
T_{e0}	1.0 keV
T_{i0}	2.0 keV
T_{r0}	0.5 keV
κ	1.0
C_e	$7.1798077208840825 \times 10^2$
C_i	8.974759651105
K_{ie}	150.0
$\overline{\sigma v}$	1.0×10^{17} cm^3/s
Q	4.0 MeV

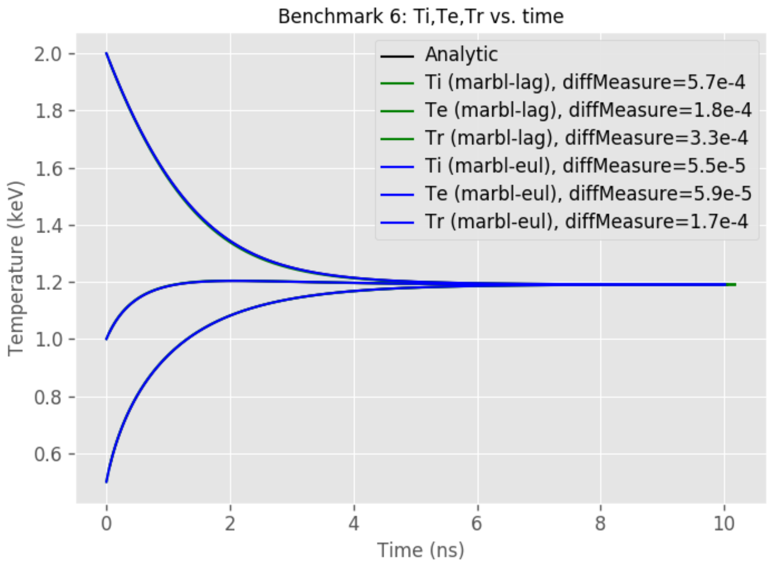


Figure 2.19: Comparison of ion (top), electron (middle), and radiation (bottom) temperature vs. time between analytic (black), MARBL-lag (green), and MARBL-eul (blue) for benchmark 6.

Parameter	**Value**
ρ	20.141018 g/cc
T_{e0}	0.8 keV
T_{i0}	0.8 keV
T_{r0}	0.2 keV
κ	1.0×10^{-5}
C_e	3.0×10^{-4}
C_i	3.0×10^{-4}
K_{ie}	1.5×10^4
$\overline{\sigma v}$	1.0×10^{17} cm^3/s
Q	4.0 MeV

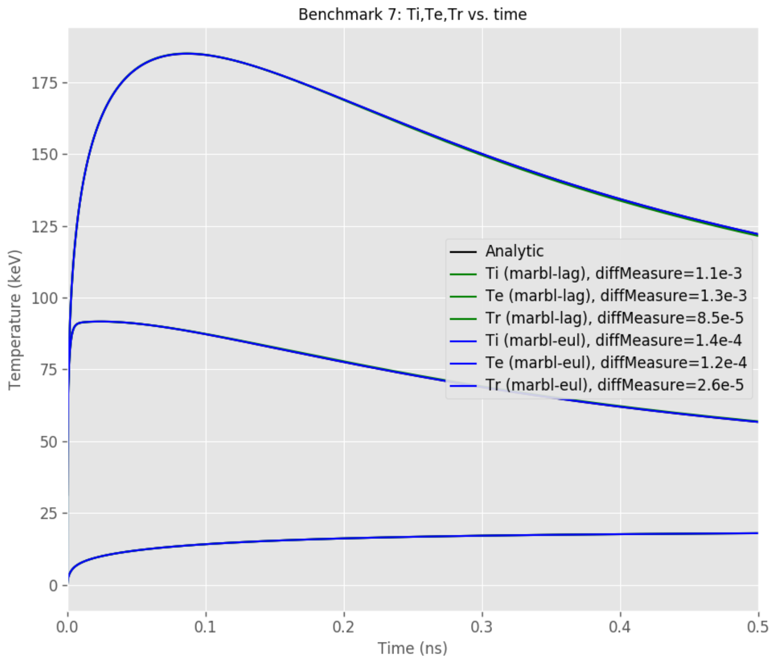


Figure 2.20: Comparison of ion (top), electron (middle), and radiation (bottom) temperature vs. time between analytic (black), MARBL-lag (green), and MARBL-eul (blue) for benchmark 7.

Benchmark 2

Benchmark 3

Benchmark 4

Benchmark 5

Benchmark 6

Benchmark 7

2.5 Thermal Conduction

In MARBL, we approximate thermal conduction via Fourier's Law by adding the second-order thermal diffusion terms $\nabla \cdot (\kappa_{ek} \cdot \nabla T_{ek})$ and $\nabla \cdot (\kappa_{ek} \cdot \nabla T_{ek} + \kappa_{ik} \cdot \nabla T_{ik})$ to the electron and total internal energy equations, respectively, where for $\alpha \in \{i, e\}$, $\kappa_{\alpha k}$ is the ion/electron thermal conductivity tensor for material k. The tensor components of $\kappa_{\alpha k}$ are known functions of the density and temperature, the latter of which is itself a function of the material specific internal energy via some known equation of state. For example, a non-degenerate electron gas has a thermal conductivity $\kappa_{ek} \propto (T_{ek})^{2.5}$.

2.5.1 Evolution Equations

To reduce the order of the thermal diffusion operator from second to first order, we introduce the material-dependent thermal flux, $F_{\alpha k} = -\eta_k \kappa_{\alpha k} \cdot \nabla T_{\alpha k}$, and add corresponding boundary constraint equations. The resulting thermal conduction subsystem is then given by

$$\eta_k \rho_k \frac{\mathrm{d}e_k}{\mathrm{d}t} = -\nabla \cdot (F_{ik} + F_{ek}), \tag{2.152a}$$

$$\eta_k \rho_k \frac{\mathrm{d}e_{ek}}{\mathrm{d}t} = -\nabla \cdot F_{ek}, \tag{2.152b}$$

$$\nabla e_{\alpha k} = -C_{V\alpha k}\, \kappa_{\alpha k}^{-1} \cdot F_{\alpha k}, \tag{2.152c}$$

$$T_{\alpha k} = \hat{T}_\alpha(\mathbf{x}, t) \qquad \text{on } \partial\Omega, \tag{2.152d}$$

$$n \cdot F_{\alpha k} = \hat{F}_{\alpha n}(\mathbf{x}, t) \qquad \text{on } \partial\Omega, \tag{2.152e}$$

where

$$C_{V\alpha k} = \left.\frac{\partial e_{\alpha k}}{\partial T_{\alpha k}}\right|_\rho \tag{2.153}$$

is the constant-volume specific heat capacity, assumed to be a smooth function of the EOS, $\kappa_{\alpha k}^{-1} \equiv (\kappa_{\alpha k})^{-1}$, and $\hat{T}_\alpha(\mathbf{x}, t)$ and $\hat{F}_{\alpha n}(\mathbf{x}, t)$ are specified boundary functions for the temperature and normal component of the thermal flux. For $\det(\kappa_{\alpha k}) > 0$, which is a physically reasonable assumption, the tensor $\kappa_{\alpha k}$ is invertible. The goal is to solve equations (2.152a)-(2.152e) for (e_k, e_{ek}) for all materials k and all groups g, where e_{ik} is not independent, but instead depends on e_k and e_{ek}.

One way to solve these equations is on a per-material basis, i.e., independently for each k. However, continuity of the thermal energy flux is required at material boundaries so that the energy conducted out of one material corresponds to the energy conducted into its neighboring material at such boundaries. This would require additional constraints be placed on the system, which complicates the numerical solution procedure. An alternative is to assume that the materials are locally held at a single temperature everywhere such that a single thermal flux can be used to transfer energy throughout the entire mesh. This does not require continuity of thermal energy at material boundaries between unmixed zones; only continuity of the thermal energy flux is required there. In mixed zones, however, this approximation does require different materials to have the same temperature, since we do not currently reconstruct a subzonal material interface there.

To derive the single-temperature system of equations, we first define the mass-weighted total and electron energies, respectively, as

$$e \equiv \sum_k Y_k e_k, \tag{2.154a}$$

$$e_e \equiv \sum_k Y_k e_{ek}, \tag{2.154b}$$

where $Y_k = \eta_k \rho_k / \rho$ is the mass fraction for material k in a given zone and $\rho \equiv \sum_k \eta_k \rho_k$ is the total mass density there. By the Product Rule, linearity of the material derivative, and the continuity equation, it follows that

$$\sum_k \eta_k \rho_k \frac{\mathrm{d}e_{\alpha k}}{\mathrm{d}t} = \rho \frac{\mathrm{d}e_\alpha}{\mathrm{d}t}. \tag{2.155}$$

Then, defining $F_\alpha \equiv \sum_k F_{\alpha k}$, equations (2.152a) and (2.152b) can then be written as

$$\rho \frac{\mathrm{d}e}{\mathrm{d}t} = -\nabla \cdot (F_i + F_e), \tag{2.156a}$$

$$\rho \frac{\mathrm{d}e_e}{\mathrm{d}t} = -\nabla \cdot F_e, \tag{2.156b}$$

which correspond to the single-material energy equations with $\eta = 1$. Finally, we define a volume-weighted mean thermal conduction coefficient and a mass-weighted, constant-volume heat capacity as

$$\kappa_\alpha \equiv \sum_k \eta_k \kappa_{\alpha k}, \tag{2.157}$$

$$C_{V\alpha} \equiv \sum_k Y_k C_{V\alpha k}, \tag{2.158}$$

respectively. Since the local material temperatures are the same, i.e., $T_{\alpha k} = T_\alpha$ for all k, it follows that

$$F_\alpha = -\kappa_\alpha \cdot \nabla T_\alpha. \tag{2.159}$$

If we assume further that $\nabla Y_k = 0$, at least locally in mixed zones, then

$$\nabla e_\alpha = C_{V\alpha} \nabla T_\alpha. \tag{2.160}$$

Thus, equations (2.152c) can be written as

$$\nabla e_\alpha = -C_{V\alpha}\, \kappa_\alpha^{-1} \cdot F_\alpha. \tag{2.161}$$

Equation (2.161) corresponds to the single-material thermal flux equation with $\eta = 1$. The same averaging procedure can be applied to the boundary temperatures and fluxes in equations (2.152d) and (2.152e).

2.5.2 Discretization

The ion/electron thermal diffusion flux plays a role analogous to the radiation flux and can be expanded in the same Raviart-Thomas (R-T) basis, $\mathscr{H}(\text{div})$. Thus, we have for the thermal energy and flux

$$e_\alpha = \sum_j e_{\alpha j} \phi_j, \qquad \phi_j \in L^2(\Omega), \tag{2.162}$$

$$F_\alpha = \sum_j F_{\alpha j} f_j, \qquad f_j \in \mathscr{H}(\text{div}, \Omega). \tag{2.163}$$

Multiplying equations (2.156a) and (2.156b) by a test function $\phi_i \in L^2(\Omega)$ in the thermodynamic basis and integrating, we obtain the weak forms given by

$$\int_\Omega \phi_i \rho \frac{\mathrm{d}e}{\mathrm{d}t} = -\int_\Omega \phi_i \nabla \cdot (F_i + F_e), \tag{2.164}$$

$$\int_\Omega \phi_i \rho \frac{\mathrm{d}e_e}{\mathrm{d}t} = -\int_\Omega \phi_i \nabla \cdot F_e. \tag{2.165}$$

Expanding e and e_e in the thermodynamic basis functions $\phi_j \in L^2(\Omega)$ and F_α in the R-T basis functions $f_j \in \mathscr{H}(\text{div}, \Omega)$, we obtain the semi-discrete form of equations (2.156a) and (2.156b) given by

$$L_\rho \frac{\mathrm{d}\mathbf{e}}{\mathrm{d}t} + D(\mathbf{F}_i + \mathbf{F}_e) = 0, \tag{2.166}$$

$$L_\rho \frac{\mathrm{d}\mathbf{e}_e}{\mathrm{d}t} + D\mathbf{F}_e = 0, \tag{2.167}$$

where $\mathbf{F}_e$ and $\mathbf{F}_i$ represent the degree-of-freedom (DOF) vectors for the electron and ion thermal fluxes, respectively, and where $\mathbf{e}$ and $\mathbf{e}_e$ represent the DOF vectors for the total and electron thermal energies, respectively. The matrix D represents the weak form of the divergence operator, and the matrix $L_\rho = \sum_k L_{\rho_k}$, where L_{ρ_k} is the energy mass matrix for material k.

Multiplying equation (2.161) by a test function $f_i \in \mathscr{H}(\text{div}, \Omega)$ in the R-T basis and integrating, we obtain the weak form given by

$$\int_\Omega f_i \cdot \nabla e_\alpha = -\int_\Omega C_{V\alpha}\, \kappa_\alpha^{-1} \cdot f_i \cdot F_\alpha. \tag{2.168}$$

Integrating by parts and substituting the natural boundary condition from equation (2.152d), we obtain

$$\int_{\partial\Omega_n} \hat{e}_\alpha f_i \cdot n - \int_\Omega e_\alpha \nabla \cdot f_i = -\int_\Omega C_{V\alpha}\, \kappa_\alpha^{-1} \cdot f_i \cdot F_\alpha, \tag{2.169}$$

where for each material k, $\hat{e}_{\alpha k}$ is derived from $\hat{T}_k^\alpha$ via a call to the EOS. Expanding e_α and $\mathbf{F}_\alpha$ as before, we obtain the semi-discrete form of equation (2.161) given by

$$R_{\kappa\alpha}\mathbf{F}_\alpha - D^T \mathbf{e}_\alpha = -\mathbf{b}_{n\alpha}, \tag{2.170}$$

where

$$[R_{\kappa\alpha}]_{ij} \equiv \int_\Omega C_{V\alpha}\, \kappa_\alpha^{-1} : f_i f_j, \tag{2.171}$$

defines a mass matrix for the ion/electron thermal flux, and

$$\mathbf{b}_{n\alpha} = \int_{\partial\Omega_n} \hat{e}_\alpha f_i \cdot n \tag{2.172}$$

defines a right-hand-side vector.

For temporal discretization, we consider the simplest case of a single first-order (Backward Euler) implicit time step. We solve for the vector $\mathbf{k} \equiv (\mathbf{k}_e, \mathbf{k}_{e_e}, \mathbf{F}_i, \mathbf{F}_e)^T$, where

$$\mathbf{k}_e \equiv \frac{\mathrm{d}\mathbf{e}}{\mathrm{d}t}, \tag{2.173}$$

and

$$\mathbf{k}_{e_e} \equiv \frac{\mathrm{d}\mathbf{e_e}}{\mathrm{d}t} \tag{2.174}$$

represent the total and electron energy coefficient increments (slopes), respectively. The implicit time dependence of κ_α via its dependence on temperature can be included by recomputing this coefficient at each Newton iteration using the previous iteration's temperature. For computational expediency in the case of weak conduction (i.e., for a conduction time scale long compared to the hydrodynamical time scale), we can simply lag this coefficient by taking it as constant over the time step. The resulting fully-discrete system is then given by

$$L_\rho \mathbf{k}_e + D(\mathbf{F}_i + \mathbf{F}_e) = 0, \tag{2.175a}$$

$$L_\rho \mathbf{k}_{e_e} + D\mathbf{F}_e = 0, \tag{2.175b}$$

$$-\Delta t D^T \mathbf{k}_{e_\alpha} + R_{\kappa\alpha}\mathbf{F}_\alpha = D^T \mathbf{e}_\alpha - \mathbf{b}_{n\alpha}. \tag{2.175c}$$

2.5.3 Solving the Linear Subsystems

The linear subsystems in equations (2.175) are given by

$$\left[\begin{array}{ccc|ccc} \ddots & & \mathbf{0} & \ddots & & \mathbf{0} \\ & L_\rho & & & D & \\ \mathbf{0} & & \ddots & \mathbf{0} & & \ddots \\ \hline \ddots & & \mathbf{0} & \ddots & & \mathbf{0} \\ & -\Delta t D^T & & & R_{\kappa i} & \\ \mathbf{0} & & \ddots & \mathbf{0} & & \ddots \end{array}\right] \left[\begin{array}{c} \mathbf{k}_e - \mathbf{k}_{e_e} \\ \\ \mathbf{F}_i \end{array}\right] = \left[\begin{array}{c} \mathbf{0} \\ \\ D^T(\mathbf{e} - \mathbf{e}_e) - \mathbf{b}_{ni} \end{array}\right], \tag{2.176}$$

and

$$\left[\begin{array}{ccc|ccc} \ddots & & \mathbf{0} & \ddots & & \mathbf{0} \\ & L_\rho & & & D & \\ \mathbf{0} & & \ddots & \mathbf{0} & & \ddots \\ \hline \ddots & & \mathbf{0} & \ddots & & \mathbf{0} \\ & -\Delta t D^T & & & R_{\kappa e} & \\ \mathbf{0} & & \ddots & \mathbf{0} & & \ddots \end{array}\right] \left[\begin{array}{c} \mathbf{k}_{e_e} \\ \\ \mathbf{F}_e \end{array}\right] = \left[\begin{array}{c} \mathbf{0} \\ \\ D^T \mathbf{e}_e - \mathbf{b}_{ne} \end{array}\right]. \tag{2.177}$$

For the 2T model we ignore $\mathbf{k}_{e_e}$ and $\mathbf{e}_e$ in equation (2.176) and ignore equation (2.177) entirely. The solutions to these subsystems can be found by first solving

$$\left[\Delta t D^T L_\rho^{-1} D + R_{\kappa i}\right] \mathbf{F}_i = D^T(\mathbf{e} - \mathbf{e}_e) - \mathbf{b}_{ni}, \tag{2.178}$$

and

$$\left[\Delta t D^T L_\rho^{-1} D + R_{\kappa e}\right] \mathbf{F}_e = D^T \mathbf{e}_e - \mathbf{b}_{ne}, \tag{2.179}$$

for $\mathbf{F}_i$ and $\mathbf{F}_e$, respectively, using standard Newton solvers. Then $\mathbf{k}_{e_e}$ may be obtained via back substitution as

$$\mathbf{k}_{e_e} = -L_\rho^{-1} D \mathbf{F}_e, \tag{2.180}$$

and $\mathbf{k}_e$ may be obtained via back substitution as

$$\mathbf{k}_e = \mathbf{k}_{e_e} - L_\rho^{-1} D \mathbf{F}_i. \tag{2.181}$$

Note that the mass matrix L_ρ is symmetric positive definite (SPD), hence $\Delta t D^T L_\rho^{-1} D$ is also SPD. Though the matrices $R_{\kappa i}$ and $R_{\kappa e}$ are symmetric by construction, it remains to be shown that they are also positive definite. It follows that $\Delta t D^T L_\rho^{-1} D + R_{\kappa i}$ and $\Delta t D^T L_\rho^{-1} D + R_{\kappa e}$ are also both symmetric; again, it remains to be shown that they are also positive definite, but it's at least a plausible assertion. For this reason, we have had very good success so far using Conjugate Gradient solvers.

2.5.4 Multi-material Solution

Once the single-temperature total and electron energy increments, $\mathbf{k}_e$ and $\mathbf{k}_{e_e}$, respectively, have been obtained, we then determine the corresponding multi-material coefficient increments for each material k using

$$\mathbf{k}_{e_k} = w_k \mathbf{k}_e, \tag{2.182}$$
$$\mathbf{k}_{e_{ek}} = w_k \mathbf{k}_{e_e}, \tag{2.183}$$

where $w_k \in [0,1]$ and $\sum_k w_k = 1$. We determine the weights w_k differently depending on the sign of $\boldsymbol{\phi}_i \cdot \mathbf{k}_{e_{\alpha k}}$, where $\boldsymbol{\phi}_i$ is the vector of thermodynamic basis functions $\phi_j \in L^2(\Omega)$ evaluated at quadrature point i. If $\boldsymbol{\phi}_i \cdot \mathbf{k}_{e_{\alpha k}}$ is positive such that materials are locally heated via thermal conduction, then the per-material weight is given by

$$w_k = \frac{C_{V\alpha k}}{C_{V\alpha}} = \frac{\rho C_{V\alpha k}}{\sum_k \eta_k \rho_k C_{V\alpha k}}, \tag{2.184}$$

so that materials are heated according to their capacity to store heat. However, if $\phi_i \cdot \mathbf{k}_{e_{\alpha k}}$ is negative such that materials are locally cooled via thermal conduction, then the per-material weight is instead given by

$$w_k = \frac{\rho C_{V\alpha k} T_{\alpha k}}{\sum_k \eta_k \rho_k C_{V\alpha k} T_{\alpha k}}, \tag{2.185}$$

so that materials are cooled according to how much energy they currently have. This helps prevent energies from going negative in multi-material zones where some material is significantly hotter than another.

2.6 Magnetohydrodynamics (MHD)

Magnetohydrodynamics is the coupling of electromagnetics with solid and fluid mechanics. The electromagnetics couples to the hydrodynamics via both a force term and a heating term. The hydrodynamics couples to the electromagnetics through motion of the materials, and temperature dependent electrical conductivity. Applications of MHD to DOE problems include:

- Laser fusion implosions, where the magnetic field may be self generated (Nernst effect) or intentionally applied
- Pulsed power EOS experiments where magnetic fields are used to compress materials to conditions that are difficult to achieve by other means
- Manufacturing processes such as inductive melting, stirring, and mixing
- Magnetically driven pumps, actuators, centrifuges, and related electromechanical systems

In 2017, MARBL began development of a high-order ALE formulation of MHD which uses high-order finite element discretizations for handling aspects of 3D magnetic diffusion in the presence of external voltage/current sources on general, high-order unstructured grids, the computation of magnetic stress (or force) and resistive Joule heating terms to couple to the hydrodynamics equations of Section 2.2 and divergence preserving advection of the magnetic field for high-order ALE remap.

2.6.1 Magnetic Diffusion

Maxwell's equations (without motion) are

$$\frac{\partial B}{\partial t} = \nabla \times E \tag{2.186}$$

$$\frac{\partial \varepsilon E}{\partial t} = -\nabla \times \frac{1}{\mu} B + J \tag{2.187}$$

$$J = \sigma E \tag{2.188}$$

$$\nabla \cdot J = 0 \tag{2.189}$$

$$\nabla \cdot B = 0 \tag{2.190}$$

where B is the magnetic flux density, E is the magnetic field, J is the electric current density, ε is the electric permittivity, μ is the magnetic permeability. Equations (2.186)-(2.188) are known as Faraday's law, Ampere's Law, and Ohm's law, respectively. Equation (2.189) is conservation of electric charge, which is not true in a general plasma, but is a key approximation of MHD. Equation (2.190) is conservation of magnetic charge, which is always true.

Another key assumption of magnetohydrodynamics is $\frac{\partial \varepsilon E}{\partial t} << \sigma E$ and therefore the term $\frac{\partial \varepsilon E}{\partial t}$ is neglected resulting in

$$J = \nabla \times \frac{1}{\mu} B \tag{2.191}$$

This means there are no speed of light effects, and the resulting equations will become diffusion equations rather than wave equations. Note that the divergence of (2.186) is zero, hence if the initial B-field is divergence-free then Equation (2.190) is always satisfied. A convenient source term for many applications is an electrostatic potential Φ, resulting in

$$J = \sigma E = \nabla \times \frac{1}{\mu} B - \sigma \nabla \Phi \tag{2.192}$$

Charge conservation then requires

$$\nabla \cdot \sigma \nabla \Phi = 0 \tag{2.193}$$

which is a Poisson equation for the electrostatic potential. Our problems of interest involve metals, insulators, and vacuum regions. In the conducting regions the equations are parabolic, in the insulating/vacuum regions the equations are elliptic. In the conducting regions there is a non-zero diffusion time (discussed below), whereas in insulating/vacuum the diffusion time is zero. Instead of decomposing the problem into elliptic and parabolic regions, it is possible to use implicit time integration of the diffusion equations with a small but non-zero conductivity in the insulating/vacuum regions. Backward Euler is the natural starting point. One possible formulation is

$$\begin{aligned} \nabla \cdot \sigma \nabla \Phi^{n+1} &= 0 && (2.194)\\ \left(\sigma I - \delta t \nabla \frac{1}{\mu} \times \nabla\right) E^{n+1} &= \nabla \times \frac{1}{\mu} B^{n} - \sigma \nabla \Phi^{n+1} && (2.195)\\ B^{n+1} &= B^{n} + \delta t \nabla \times E^{n+1} && (2.196) \end{aligned}$$

Note that Φ and E are computed from scratch at every time time step, the only state variable is B. This time integration is particularly advantageous when the spatial discretization is given by:

- Φ is discretized using standard $H1$ finite elements
- E is discretized using $H(curl)$ finite elements
- B is discretized using $H(div)$ finite elements

Physical fields can be categorized according to continuity and differentiability properties as

$$\begin{array}{lllllll} \omega^0 & \in & H^1 & & \nabla \omega^0 & \in & H(curl) \\ \omega^1 & \in & H(curl) & & \nabla \times \omega^1 & \in & H(div) \\ \omega^2 & \in & H(div) & & \nabla \cdot \omega^2 & \in & L_2 \\ \omega^3 & \in & L_2 & & & & \end{array} \tag{2.197}$$

It is possible to define finite element basis functions that respect these differential relations, denoted by

$$\begin{aligned} W^0 &\in H_h^1 && (2.198)\\ W^1 &\in H_h(curl) && (2.199)\\ W^2 &\in H_h(div) && (2.200)\\ W^3 &\in L_{2_h} && (2.201) \end{aligned}$$

where the subscript h denotes that these spaces are are polynomial spaces defined on a computational mesh. We have The

$$\begin{tikzcd}
\omega^0 \arrow[r, "d"] \arrow[d, "\pi"] & \omega^1 \arrow[r, "d"] \arrow[d, "\pi"] & \omega^2 \arrow[r, "d"] \arrow[d, "\pi"] & \omega^3 \arrow[r, "d"] \arrow[d, "\pi"] & 0 \\
W^0 \arrow[r, "d"] & W^1 \arrow[r, "d"] & W^2 \arrow[r, "d"] & W^3 \arrow[r, "d"] & 0
\end{tikzcd}$$

Figure 2.21: The DeRham diagram for compatible finite elements, where d denotes derivative and π denotes projection. This diagram specifies the commuting properties that compatible finite element must satisfy, i.e. derivative of the projection must equal the projection of the derivative. This property is used throughout MARBL for discretizing fields on a mesh.

electromagnetic fields will be approximated by

$$\Phi = \sum_{W^0} v_i W_i^0 \tag{2.202}$$

$$E = \sum_{W^1} e_i W_i^1 \tag{2.203}$$

$$B = \sum_{W^2} b_i W_i^2 \tag{2.204}$$

and this means that the vector identities $\nabla \times \nabla \Phi = 0$ and $\nabla \cdot \nabla \times E = 0$ are satisfied exactly, which leads to strict conservation of electric charge and magnetic flux. For more background on compatible finite elements for electromagnetics see [28] [29].

The resulting finite element system of equations is then

$$S_0 v^{n+1} = V \tag{2.205}$$

$$(M_1 - \delta t S_1) e^{n+1} = D_{12}^T b^n - \sigma K_{01} v^{n+1} \tag{2.206}$$

$$b^{n+1} = b^n + \delta t K_{12} e^{n+1} \tag{2.207}$$

where S_0 is the H^1 stiffness matrix and V is the applied voltage boundary conditions, M_1 and S_1 are the $H(curl)$ mass and stiffness matrices, D_{12}^T is the weak curl operator, K_{01} is the strong gradient operator, and K_{12} is the strong curl operator. These matrices are given by

$$S_0 = \left\langle \sigma \nabla W^0, \nabla W^0 \right\rangle \tag{2.208}$$

$$M_1 = \left\langle \sigma W^1, W^1 \right\rangle \tag{2.209}$$

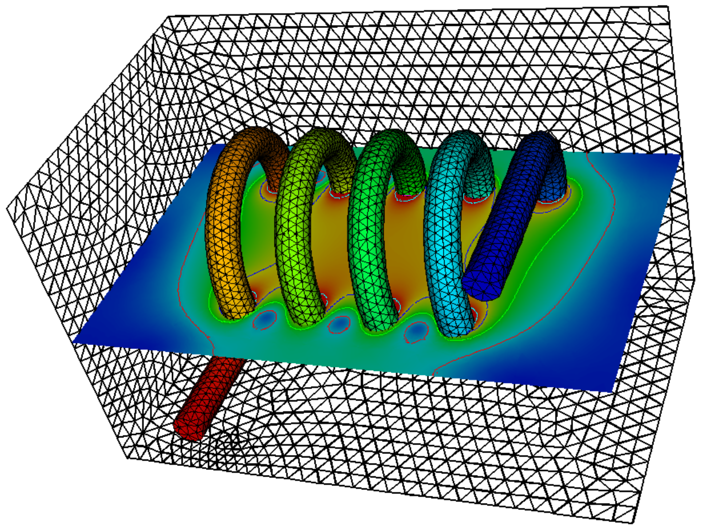
$$S_1 = \left\langle \frac{1}{\mu} \nabla \times W^1, \nabla \times W^1 \right\rangle \tag{2.210}$$

$$D_{12} = \left\langle \nabla \times W^1, W^2 \right\rangle \tag{2.211}$$

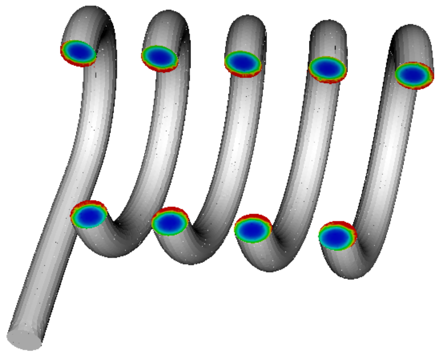
where $\langle \cdot, \cdot \rangle$ denotes integration over the entire domain. Note the matrices K_{01} are K_{12} so-called topological derivatives, they do not involve integrals over the mesh. Instead they are purely algebraic, they depend upon the mesh topology but not the mesh nodal coordinates

The LLNL MFEM library is used for the higher order basis functions, quadrature rules, and matrix assembly. Note the S_1 matrix is singular even when essentially boundary conditions are applied, hence depending upon the linear solver we can't set $\sigma = 0$ in vacuum regions, instead a small non-zero value of σ is used to regularize the linear system. Using the Hypre AMS solver, a ratio of $\sigma_{max}/\sigma_{min} = 10^6$ does not pose any significant issue.

The following example illustrates the use of the higher order magnetic field solver in MARBL. The problem consists of a metal coil in air, a voltage is applied to the ends of the coil. This induced current to flow in the coil, producing a magnetic field in the surrounding air. This is a transient problem, a snapshot at one instant of time is shown in Figure 2.22. Notice that the mesh is tetrahedral, which facilitates automatic meshing of complex shapes. In addition the mesh elements are curved which provides excellent modeling of the curved coil surface. The magnetic field is shown in Figure 2.22a and the current density is shown in Figure 2.22b. While the mesh looks coarse, because the fields are represented by higher order finite element basis functions within each element the fields are in fact highly resolved, as can be seen by closely examining the thin layer of current density in the coil. Because of the $H(curl) - H(div)$ formulation, both the magnetic field and the current density are *exactly* divergence-free. This is very important when the magnetic forces are coupled to the hydrodynamics because the notion of a magnetic stress tensor is only valid when $\nabla \cdot B = 0$. The convergence properties of the above $H(curl) - H(div)$ formulation of electromagnetic diffusion has been studied in [30], with the key conclusion that arbitrary high order convergence is achieved when increasing the order of the polynomial basis and quadrature rules.



(a) Voltage and Magnetic Field



(b) Current Density

Figure 2.22: Higher Order Modeling of a Coil. This problem demonstrates the ability of MARBL to run on higher order tetrahedral meshes. Note that while the mesh looks coarse, because the basis functions are higher order the fields and currents are well resolved.

2.6.2 Electromagnetics with Motion

In this section electromagnetics with motion is reviewed, with the key conclusion that higher order $H(div)$ basis functions are ideally suited for advection of magnetic fields in the Lagrangian (material) frame.

Faraday's law is given by

$$\frac{\partial B}{\partial t} = -\nabla \times E \tag{2.212}$$

however the integral form is more general,

$$\frac{d}{dt}\int_S B\cdot dS = -\int_C E\cdot dl \tag{2.213}$$

where S is a surface with closed contour C as shown in Figure 2.23. This applies to any curve C, C may be fixed to space, C may be moving with the fluid, C may be moving arbitrarily. The curve C can be considered an instrument that measures B and E, and Faraday's law states how these measurements are related. The right hand side is known as "electromotive force", E.M.F., or simply voltage.

Consider an solenoidal field G integrated over a moving surface with velocity u, giving a time dependent flux. The flux may change for two reasons: 1) the surface may be constant but the field G is varying in time, 2) the field G is time independent but the surface is moving. The first term is simply

$$\int_S \frac{\partial G}{\partial t}\cdot dS \tag{2.214}$$

Now consider the second term. In a time interval δt the surface adjacent to line element dl increases by an amount $dS = (u\times dl)\,\delta t$, so the change in flux is

$$\delta\int_S G\cdot dS = \int_C G\cdot(u\times dl)\,\delta t = -\int_C (u\times G)\cdot dl\delta t \tag{2.215}$$

using Stokes theorem the last line integral can be converted to a surface integral, giving

$$\delta\int_S G\cdot dS = -\int_C \nabla\times(u\times G)\cdot dS \tag{2.216}$$

The total change in flux is therefore

$$\frac{d}{dt}\int_S G\cdot dS = \int_S\left[\frac{\partial G}{\partial t} - \nabla\times(u\times G)\right]\cdot dS \tag{2.217}$$

This is a kinematic equation that holds for any solenoidal field G. The area of integration is illustrated in Figure 2.23.

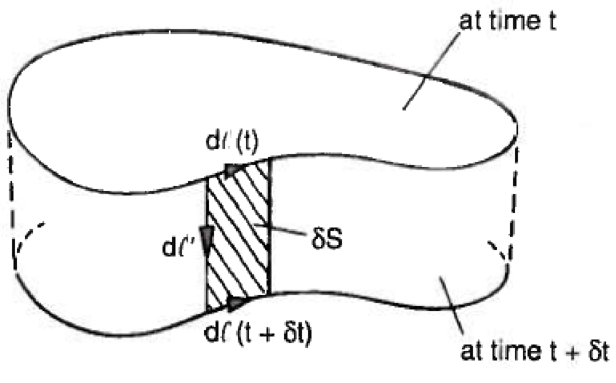


Figure 2.23: The moving circuit

The full Leibniz rule is

$$\frac{d}{dt}\int_S G\cdot dS = \int_S\left[\frac{\partial G}{\partial t} + (\nabla\cdot G)\,u\cdot dS - \nabla\times(u\times G)\right]\cdot dS \tag{2.218}$$

Now this can be combined with the integral form of Faraday's law to give

$$\int_S\left[\frac{\partial B}{\partial t} - \nabla\times(u\times B)\right]\cdot dS = -\int_C E\cdot dl \tag{2.219}$$

Now lets let the surface S with contour C be moving with the conducting fluid. In this material frame we have $J = \sigma E$, where σ is the electrical conductivity. In the limit of $\sigma \to \infty$ we have $E \to 0$, the electric field measured in the material frame of a perfect conductor is zero. This gives

$$\frac{d}{dt}\int_S B \cdot dS = 0 \tag{2.220}$$

$$\frac{d}{dt}\int_S B \cdot dS = \int_S \left[\frac{\partial B}{\partial t} - \nabla \times (u \times B)\right] \cdot dS = 0 \tag{2.221}$$

which is known as the frozen-in-flux theorem, the magnetic flux through any surface moving with the fluid is constant. The time evolution of the magnetic field in a perfectly conducting fluid is therefore given by

$$\frac{\partial B}{\partial t} = \nabla \times (u \times B) \tag{2.222}$$

this is the magnetic advection equation of ideal MHD.

Let $\hat{K}$ be a reference element and let K be an actual element given by

$$K = F\left(\hat{K}\right) \tag{2.223}$$
$$F(\hat{x}) = J(\hat{x}) + b \tag{2.224}$$

where J is the Jacobian from $\hat{x}$ to x. The Piola transformation of a vector field $q(\hat{x})$ is

$$q(x) = \frac{1}{\det(J)} \cdot J \cdot q(\hat{x}) \tag{2.225}$$

The Piola transformation is dual to the flux integral (2.220), any vector field that transforms according to (2.225) will satisfy the frozen-in-flux condition (2.220) exactly. In section Section 2.6.1 above it was mentioned that in MARBL the magnetic field is represented by higher order $H(div)$ basis functions. This means that at any point x we have

$$B(x) = \sum_i b_i W_i(x) \tag{2.226}$$

where b_i are the degrees-of-freedom and W are the $H(div)$ basis functions. By definition the $H(div)$ basis functions transform according to the Piola transformation, giving

$$B(x) = \sum_i b_i \frac{1}{\det(J)} \cdot J \cdot \hat{W}_i(\hat{x}) \tag{2.227}$$

Inserting (2.227) into (2.220) it can be seen that the degrees-of-freedom b_i have units of flux, and the frozen-in-flux theorem (and hence the magnetic advection equation (2.222)) is satisfied *exactly* by simply keeping the degrees-of-freedom b_i constant. This is a unique consequence of solving the equations in the Lagrangian frame and using $H(div)$ basis functions, the basis functions themselves satisfy the magnetic advection equation.

2.6.3 Magnetic Advection

As derived above the MHD advection equation is

$$\frac{\partial B}{\partial \tau} = -\nabla \times (U \times B) \tag{2.228}$$

and in the Lagrangian frame this equation is satisfied exactly by simply using $H(div)$ basis functions for the magnetic field B. But in the ALE setting the pseudo velocity U is not a physical velocity but rather a mesh relaxation velocity, meaning we need to replace mesh X^{old} (time $\tau = 0$) with mesh X^{new} (time $\tau = 1$) following the path described by the velocity

$$U(\tau) = \frac{d}{d\tau} X(\tau). \tag{2.229}$$

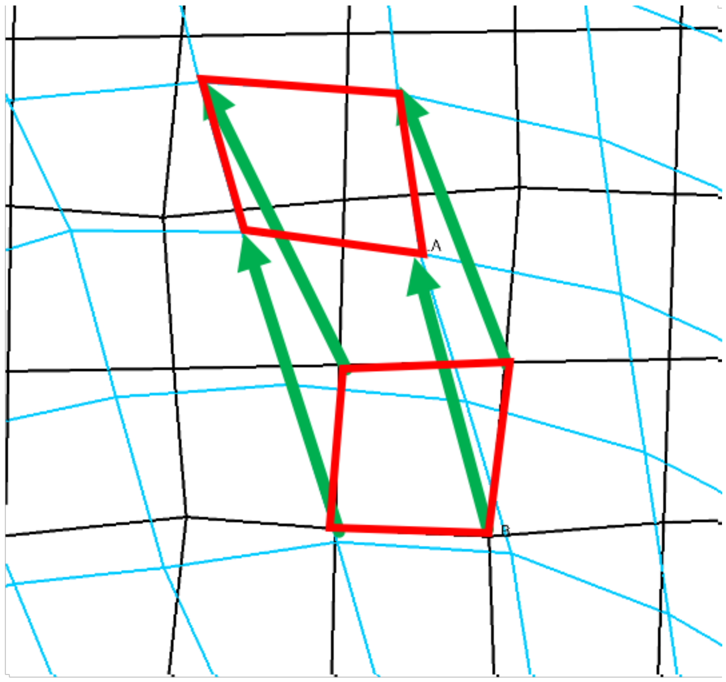


Figure 2.24: Illustration of mesh relaxation. In MHD the magnetic field must be mapped from the old mesh to the new mesh, and this can be interpeted as integrating the magnetic advection equation using a pseudo-velocity. Black: old mesh, Blue: new mesh; Green: psuedo velocity $U(x,\tau)$.

We introduce an auxiliary field $\tilde{E}$ such that

$$\tilde{E} = U \times B \tag{2.230}$$

$$\frac{\partial B}{\partial \tau} = -\nabla \times \tilde{E} \tag{2.231}$$

The equations 2.230 and 2.231 will be integrated in time using a high-order explicit time integrator e.g. 4th order Runge-Kutta.

We take a finite element approach, we are given the finite element expansions

$$B = \sum_i b_i F_i \tag{2.232}$$

$$E = \sum_i e_i W_i \tag{2.233}$$

$$U = \sum_i u_i H_i \tag{2.234}$$

where F represents $H(div)$ or RT basis , W represents $H(curl)$ or ND basis, and H represents H^1 basis. Equations 2.230 and 2.231 will be solved in a variational manner, with all spatial integrations $\langle\rangle$ performed over the mesh $X(\tau)$. This gives for the E-field

$$\langle W_i, W_j \rangle e = \langle U \times B, W_j \rangle \tag{2.235}$$

where $\langle U \times B, W_j \rangle$ is a load vector for the given source term $U \times B$, and $\langle W_i, W_j \rangle$ is a mass matrix and is easily ”inverted”.

To maintain perfect divergence-free character of the B-field, we need to have

$$\frac{\partial b}{\partial t} = Ke \tag{2.236}$$

where K is a discrete curl operator. The idea is that

$$\langle F_i, F_j \rangle b = \langle \nabla \times W_i, F_j \rangle e \tag{2.237}$$

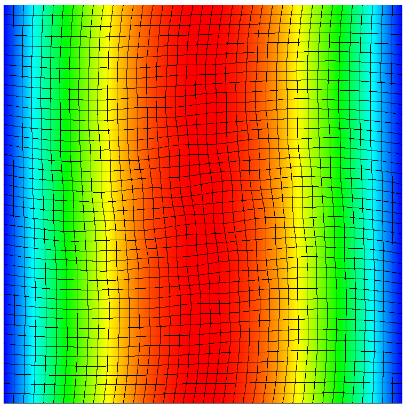
but we have $\nabla \times W_i \in RT$ for all W_i, giving

$$\langle F_i, F_j \rangle b = \langle \nabla \times W_i, F_j \rangle e = \langle F_i, F_j \rangle Ke \tag{2.238}$$

therefore the mass matrix cancels on each side of the equation an is not needed. The matrix K is is algebraic (independent of the nodal coordinates of the mesh) and is thus sometimes referred to as a topological derivative, it depends only upon the mesh connectivity and is therefore independent of the pseudo-velocity U.

A note on flux limiting. Under certain conditions e.g. very high electrical conductivity (little diffusion) and sharp jumps in magnetic field magnitude, flux limiting may be required to prevent non-physical oscillations in the magnetic field. Numerous flux limiters were investigated, and in addition discontinuous Galerkin methods were also investigated. Flux limiting of higher order magnetic field advection is still an open question, and in the results below no flux limiters were applied.

The following are some verification runs of the above MHD advection algorithm. The first type of verification experiment has an initial divergence-free magnetic field, there is no physical material motion but instead the mesh has a prescribed psuedo-velocity. As the mesh moves the magnetic field is advected from the old mesh to the new mesh. Since there is no physical motion of material, the exact solution is simply that the magnetic field should not change. Figure Figure 2.25 shows snapshots of this verification experiment at two instants of time. In these figures the magnetic field is in the $\hat{y}$-direction, and has a spatially varying magnitude as a function of x. The magnetic field is approximated using 3rd order basis functions. The prescribed motion of the mesh is a sinusoid.


(a) Magnetic Field at Early Time

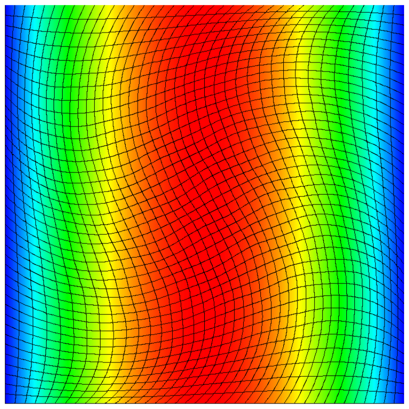
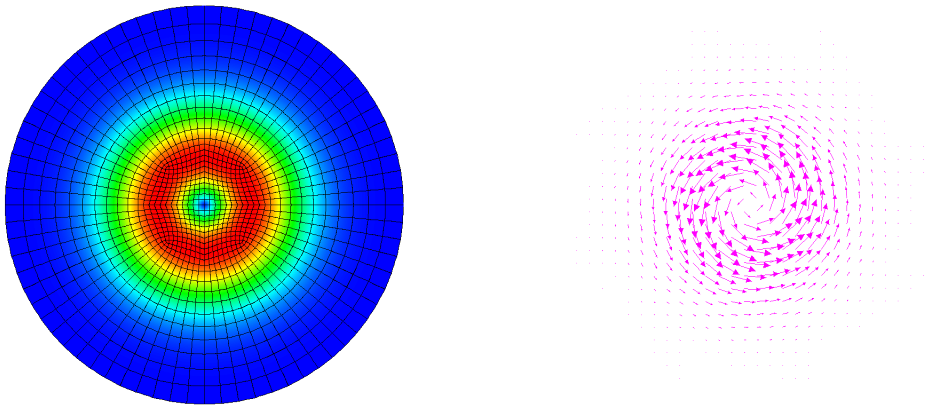

(b) Magnetic Field at Later Time

Figure 2.25: Advection of Magnetic Field. In this test there is no physical motion of material and hence the magnetic field should remain unchanged. But there is prescribed mesh motion, hence the magnetic field must be remapped (advected) over and over again as the mesh moves. Since the initial condition is smooth, higher order convergence is expected.

In this second verification experiment the initial magnetic field is given by the *curl* of a z-directed vector potential with magnitude of a Gaussian in the $x-y$-plane Figure 2.26. In this example the material is rotating at a constant angular velocity, but the mesh is remapped back to its original position. Thus this is an Eulerian via Remap test. Again, the exact solution is that the initial magnetic field should remain unchanged as the material rotates. The computational mesh and the initial magnetic field are shown in Figure 2.26. For both of the verification experiments above the convergence can be verified by refining the mesh and computing the L_2 error between the advected field and the initial field. This is done for various orders of the magnetic field basis functions.



(a) Mesh and Magnetic Field Magnitude (b) Vector Magnetic Fied

Figure 2.26: Eulerian via Remap Test. In this problem the material rotates at a constant angular velocity, but the mesh is relaxed back to its initial configuration every 20 cycles. Thus the magnetic field is remapped (advected) over and over again. Since the mesh is nonuniform and unstructured this is an excellent test of the advection algorithm.

The time integration remains fixed and is limited to 4th order. The problem is run for a fixed time interval. The convergence results are shown in Figure 2.27 The results verify that the magnetic advection algorithm is higher order.

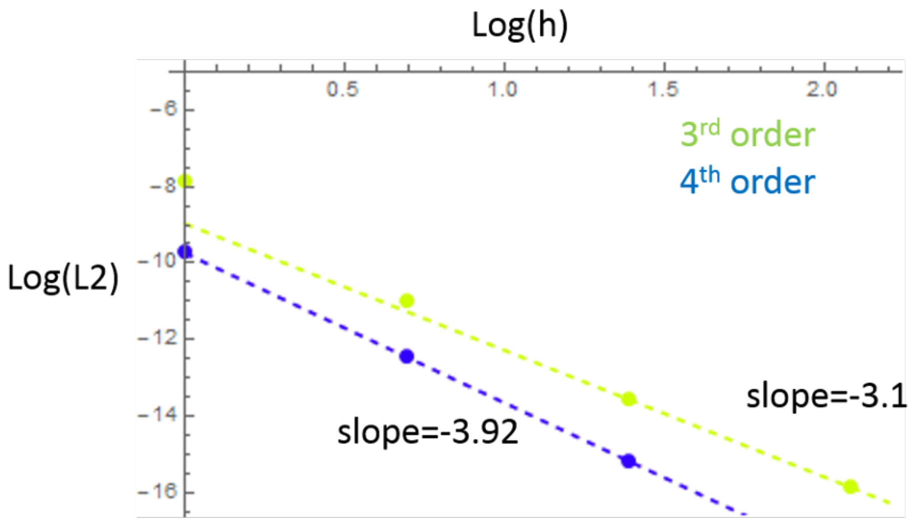
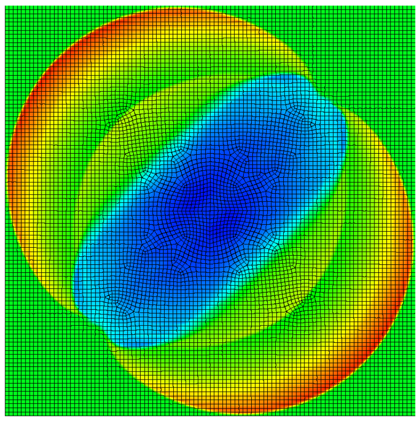
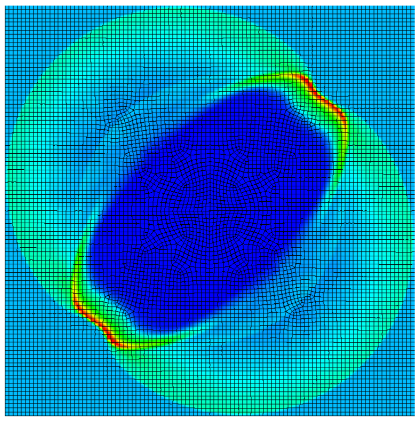


Figure 2.27: Convergence of magnetic field advection for 3^{rd} and 4^{th} order basis functions.

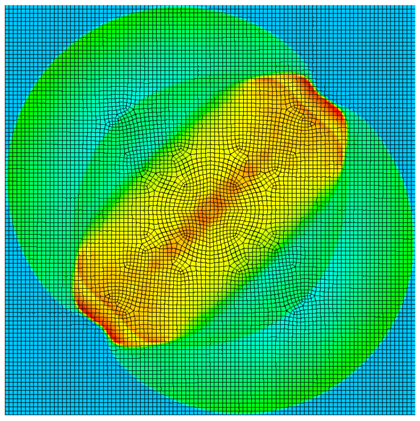
The Magnetic Sedov problem is a standard test for MHD codes. There is no exact analytical solution, but certain quantities should be conserved, and performance can be evaluated on intentionally-weird computational meshes. In the example shown in Figure 2.28 the mesh is a semi-random paving of quadrilateral elements, and it is shown to have no effect on the solution.



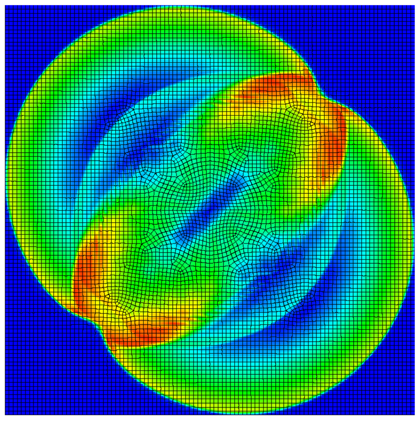
(a) Magnetic Field



(b) Density



(c) Pressure



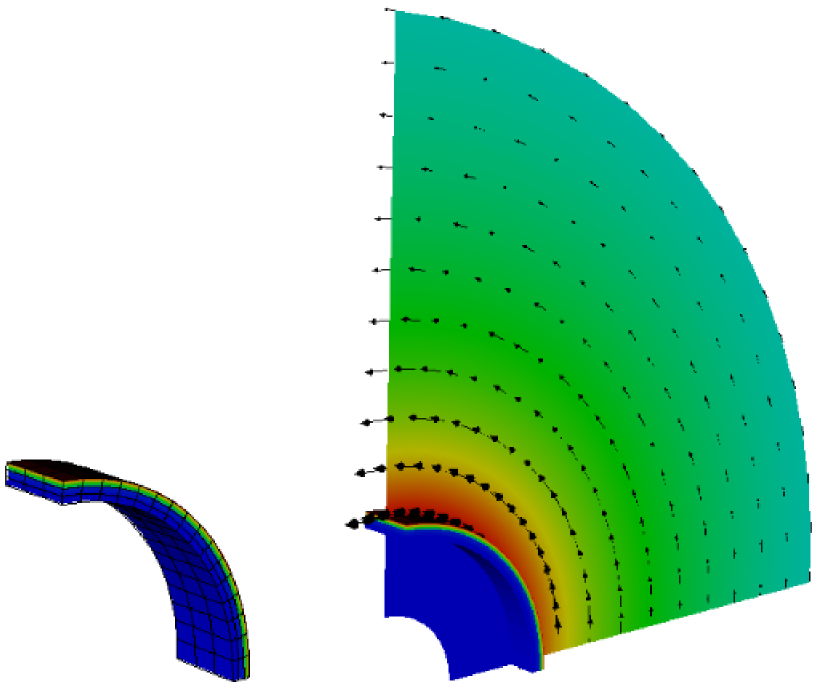
(d) Velocity

Figure 2.28: In the magnetic Sedov problem the magnetic field imposes an anisotropy in the solution, as waves in one direction propagate at the Alfven velocity, whereas waves in the orthogonal direction propagate at the magnetosonic velocity. Since there is no diffusion, this is a good test of the advection algorithm.

2.7 Mutirate Implicit / Explicit (IMEX) Time Integration

2.7.1 A Motivating Problem for Multirate Time Integration

In this section the magnetic time step will be examined for a simple example MHD problem. Consider a hollow metal tube as shown in Figure 2.29. At $t = 0$ a voltage is applied across the ends of the tube resulting in an electrostatic potential. A current will be induced in the metal, this current will diffuse from the outside to the interior. At early times the current density is confined to a thin layer as shown in Figure 2.29a. This current generates a magnetic inside and outside of the cylinder as shown Figure 2.29b. The magnetics is coupled to the hydrodynamics through two mechanisms, magnetic force term is added to the momentum equations, and a Joule heating term is added to the energy equation. The computational mesh must contain a significant vacuum



(a) Current Density (b) Magnetic Field

Figure 2.29: Magnetic fields crushing a hollow metal tube

region outside the tube if the problem is to correctly model a tube in infinite space, otherwise the magnetic field will not have the correct $1/r$ dependence. Since the conductivity is essentially zero outside the tube the magnetic field diffuses across the vacuum in a single time step. The diffusion time in the metal is given (approximately - this formula is the 1D planar case) by

$$\tau = \sigma \mu h^2 \tag{2.239}$$

Using values for copper $\sigma = 580$, $\mu = 4\pi$, and a mesh element size of $h = 0.01667$ gives $\tau = 2.025\mu s$. The sound speed in copper is given by $\sqrt{Y/\rho}$ where Y is Youngs modulus and ρ is the density. Using $Y = 1.17$ (copper) and $\rho = 8.93$ (copper) gives $s = 0.362$, and the standard Courant condition is then $\Delta t_{Courant} = 0.046\mu s$. However if second order finite element basis functions are used for the hydrodynamics the time step is reduced, and if artificial viscosity is used the time step is reduced further. For this particular simulation the MARBL chosen time step is $\Delta t_{hydro} = 0.0085$. We see that the hydro time step is $253\times$ smaller than the magnetic diffusion time. Thus, for physical reasons, it may possible to use a larger time step for the magnetic diffusion than that used for explicit hydrodynamics. This is part of the motivation for multirate time stepping discussed in Section 2.7.

Perhaps counter-intuitively, there is a also a mathematical argument for taking large magnetic time steps. Consider the left hand side of (2.206). For analysis purposes, consider the 1-dimensional case, uniform grid, linear basis functions. The mass and stiffness matrices are

$$M_1 = \sigma/h \begin{bmatrix} 1/2 & 1/6 & 0 & 0 & 0 \\ 1/6 & 1/2 & 1/6 & 0 & 0 \\ 0 & 1/6 & 1/2 & 1/6 & 0 \\ * & * & * & * & * \end{bmatrix} \tag{2.240}$$

$$S_1 = 1/\left(\mu h^3\right)\begin{bmatrix} -2 & 1 & 0 & 0 & 0 \\ 1 & -2 & 1 & 0 & 0 \\ 0 & 1 & -2 & 1 & 0 \\ * & * & * & * & * \end{bmatrix} \tag{2.241}$$

Clearly, the combined matrix $(M_1 - \delta t S_1)$ will be an M-matrix if and only if

$$\delta t > \sigma\mu h^2/6 \tag{2.242}$$

If δt is too small, the mass matrix dominates, the discrete operator is no longer a diffusion operator, and there will be non-physical ”wiggles” in the solution. Clearly the criteria (2.242) is related to the physical diffusion time, the computational time step can't be significantly smaller (in this case 1/6th) of the physical diffusion time. This has been previously analyzed [31]. If this were a magnetics-only simulation, the solution is to keep δt fixed at the desired temporal resolution and then refine the mesh until (2.242) is satisfied. But in a MHD simulation, refining the mesh will decrease the hydrodynamics time step. Another solution is to perform mass-lumping on M_1, while trivial in 1-dimension this is non-trivial on 2D or 3D unstructured meshes, and impossible for higher order $H(curl)$ finite elements. The proposed solution is to use a small time step for the hydrodynamics and an larger time step for the magnetics, this is discussed further in Section 2.7.

2.7.2 Infinitesimal Step Multirate Methods

Consider the ODE $y' = f(y)$. In many applications this ODE can partitioned as

$$y' = f(y) = f^a(y) + f^b(y) \tag{2.243}$$

where the partition could be linear and nonlinear terms, or expensive and inexpensive terms, or fast and slow terms, etc. While a standard explicit or implicit integrator can be used for (2.243), there can be challenges satisfying stability, accuracy, and efficiency requirements.

The goal of a multirate integrator is to use different integration schemes for $f^a(y)$ and $f^b(y)$ with appropriate coupling to ensure stability and accuracy requirements while maximizing efficiency. There are a wide variety of multirate methods, explicit-explicit, implicit-implicit, explicit-implicit, implicit-explicit, etc.

In our application $f^a(y)$ represents hydrodynamics and $f^b(y)$ represents diffusion e.g. magnetic fields, temperature, radiation, etc. We have the following *ansatz*:

- The hydrodynamics is fast, the diffusion is slow.
- We wish to resolve the hydrodynamics.
- The diffusion must be done implicitly (e.g. 10^6 contrast in material diffusivity properties)
- An implict diffusion step is much more expensive than an explicit hydrodynamics step

Given the above, a natural choice is to treat the hydrodynamics explicitly using a small time step, treat the diffusion implicitly with a large time step. For the remainder we will use the notation

$$y' = f(y) = f^s(y) + f^f(y) \tag{2.244}$$

While ”temporal subcycling” of different physics is not new, the key point is that formal multirate methods do the coupling of the fast and slow terms correctly, with high order accuracy.

In addition, for our application the solution y may be spatially partitioned,

$$\begin{bmatrix} y^s \\ y^f \end{bmatrix} = \begin{bmatrix} f^s\left(y^s, y^f\right) \\ f^f\left(y^s, y^f\right) \end{bmatrix} \tag{2.245}$$

A second-order multirate method is not too difficult to implement and should provide much needed accuracy, efficiency, flexibility. Higher-order multirate methods are not inexpensive, and these methods may require significant storage, hence the utility of higher-order multirate methods for real-world multiphysics problems is still a topic or research.

The are numerous variants of multirate infinitesimal methods, we are interested in the fast-explicit slow-implicit type. We begin with an example method, the implicit trapezoidal MRI-GARK (Multirate Infinitesimal Generalized Additive Runge-Kutta). method. This is a relatively new method for solving multirate ODE's, see [32] [33] for background information. Again we assume the additive splitting (2.243). A single step of the method takes y_n to y_{n+1} with step size H. The key aspect of this method is the introduction of an auxiliary ODE $v' = f(v)$ that is integrated exactly (in practice the exact integration is replaced by a very accurate numerical integration e.g. M steps of 4th order RK).

$$\begin{aligned} v(0) &= y_n && (2.246)\\ v' &= f^f(v) + f^s(y_n) \quad \theta \in [0,H] && (2.247)\\ Y_2^s &= v(H) && (2.248)\\ y_{n+1} &= Y_2^s - \frac{1}{2}Hf^s(y_n) + \frac{1}{2}Hf^s(y_{n+1}) && (2.249) \end{aligned}$$

The auxiliary ODE is given by (2.247) and consists of the fast term plus a constant slow term. This auxiliary ODE is integrated "exactly". The slow field is updated with a final implicit step (2.249). This method is second order accurate.

The equations can be re-written to make the implementation somewhat more clear. The state y is split into slow and fast, giving

$$\begin{aligned} \begin{bmatrix} v^f(0) \\ v^s(0) \end{bmatrix} &= \begin{bmatrix} y_n^f \\ y_n^s \end{bmatrix} && (2.250)\\ \begin{bmatrix} v^f \\ v^s \end{bmatrix}' &= \begin{bmatrix} f^f(v^f, v^s, t) \\ f^s\left(v_n^f, v_n^s, t_n\right) \end{bmatrix} \quad \theta \in [0,H] && (2.251)\\ y_{n+1}^f &= v^f(H) && (2.252)\\ y_{n+1}^s - \frac{1}{2}Hf^s\left(y_{n+1}^f, y_{n+1}^s, t_{n+1}\right) &= y_n^s + \frac{1}{2}Hf^s\left(y_n^f, y_n^s, t_n\right) && (2.253) \end{aligned}$$

Note that in (2.253) the right hand side "rewinds" the slow state to time $t + H/2$, and then integrates using implicit backward Euler from $t + H/2$ to $t + H$ using the fast state at $t + H$. The value of $f^s\left(y_{n+1}^f, y_{n+1}^s, t_{n+1}\right)$ must be saved for the next cycle.

Note that in (2.251) the fast physics is integrated using extrapolated values of the slow physics from time t_n. A possible modification is to start the step by first updating the slow physics to time $t_{n+\frac{1}{2}}$. This is in fact required at $t = 0$ to bootstrap the time integration.

$$\begin{aligned} \begin{bmatrix} v^f(0) \\ v^s(0) \end{bmatrix} &= \begin{bmatrix} y_n^f \\ y_n^s \end{bmatrix} && (2.254)\\ \hat{y}_{n+1}^s - Hf^s\left(y_n^f, \hat{y}_{n+1}^s, t_{n+1}\right) &= y_n^s && (2.255)\\ \begin{bmatrix} v^f \\ v^s \end{bmatrix}' &= \begin{bmatrix} f^f(v^f, v^s, t) \\ \alpha f^s\left(v_n^f, \hat{v}_{n+1}^s, t_{n+1}\right) + \beta f^s\left(v_n^f, v_n^s, t_n\right) \end{bmatrix} \quad \theta \in [0,H] && (2.256)\\ y_{n+1}^f &= v^f(H) && (2.257)\\ y_{n+1}^s - \frac{1}{2}Hf^s\left(y_{n+1}^f, y_{n+1}^s, t_{n+1}\right) &= y_n^s + \frac{1}{2}Hf^s\left(y_n^f, y_n^s, t_n\right) && (2.258) \end{aligned}$$

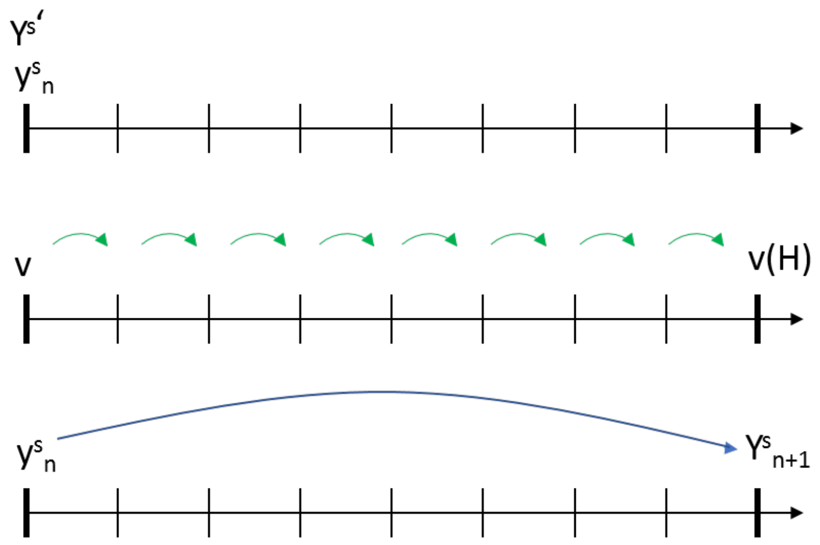


Figure 2.30: Illustration of Infinitesimal Multirate of Order 2 with Implicit Slow Stage. The Green Arrows Represent the M Steps of Runge-Kutta, the Blue Arrow Represents the Implict Slow Step.

Higher order versions of fast-explicit slow-implicit MRI-GARK can be written as a sequence of stages, with stage i given by

$$\begin{aligned}
v(0) &= Y^s_{i-1} \\
v' &= \Delta c^s_i f^f(v) + \sum_{j=1}^{i-1} \gamma_{i-1,j} f^s\left(Y^s_j\right) \quad \theta \in [0,H] \\
Y^s_i &= v(H) \\
Y^s_{i+1} &= Y^s_i + H \sum_{j=1}^{i+1} \gamma_{i-1,j} f^s\left(Y^s_j\right)
\end{aligned}$$

For example, the third order method written out completely is

$$\begin{aligned}
Y^s_1 &= y_n \\
v(0) &= Y^s_1 \\
v' &= (c_2 - c_1) f^f(v) + \gamma_{1,1} f^s(Y^s_1) \quad \theta \in [0,H] \\
Y^s_2 &= v(H) \\
Y^s_3 &= Y^s_2 + \gamma_{2,1} f^s(Y^s_1) + \gamma_{2,3} f^s(Y^s_3) \\
v(0) &= Y^s_3 \\
v' &= (c_4 - c_3) f^f(v) + \gamma_{3,1} f^s(Y^s_1) + \gamma_{3,3} f^s(Y^s_3) \quad \theta \in [0,H] \\
Y^s_4 &= v(H) \\
Y^s_5 &= Y^s_4 + \gamma_{4,3} f^s(Y^s_3) + \gamma_{4,5} f^s(Y^s_5) \\
v(0) &= Y^s_5 \\
v' &= (c_6 - c_5) f^f(v) + \gamma_{5,1} f^s(Y^s_1) + \gamma_{5,3} f^s(Y^s_3) + \gamma_{5,5} f^s(Y^s_5) \quad \theta \in [0,H] \\
Y^s_6 &= v(H) \\
Y^s_7 &= Y^s_6 + \gamma_{6,3} f^s(Y^s_3) + \gamma_{6,5} f^s(Y^s_5) + \gamma_{6,7} f^s(Y^s_7) \\
y_{n+1} &= Y^s_7
\end{aligned}$$

In the above equation the coefficients c and γ are chosen to satisfy stability and accuracy requirements. Note that there are three implicit solves, and note that the fast ODE is integrated three times. Thus this is a significant increase in complexity compared to the above 2nd order method.

2.7.3 An Analytical Multirate Example

Consider the ODE

$$\begin{bmatrix} y_1 \\ y_2 \end{bmatrix}' = A \begin{bmatrix} y_1 \\ y_2 \end{bmatrix} \tag{2.259}$$

where the matrix A is

$$A = \begin{bmatrix} \lambda_1 + \lambda_2 & -\lambda_1\lambda_2 \\ 1 & 0 \end{bmatrix} \tag{2.260}$$

The matrix A is diagonalizable, $A = PDP^{-1}$, with

$$D = \begin{bmatrix} \lambda_1 & 0 \\ 0 & \lambda_2 \end{bmatrix} \tag{2.261}$$

$$P = \begin{bmatrix} \lambda_2 & \lambda_1 \\ 1 & 1 \end{bmatrix} \tag{2.262}$$

The exact solution is

$$\begin{bmatrix} y_1 \\ y_2 \end{bmatrix} = P \begin{bmatrix} e^{\lambda_1 t} & 0 \\ 0 & e^{\lambda_2 t} \end{bmatrix} P^{-1} \begin{bmatrix} y_1 \\ y_2 \end{bmatrix}^0 \tag{2.263}$$

If we have $|\lambda_1| >> |\lambda_2|$ (we assume $\lambda_1 < 0$ and $\lambda_2 < 0$) the system is "stiff", the exact solution consists of a linear combination of rapidly decaying term and a slowly decaying term. We can decompose $A = A^{fast} + A^{slow}$ where

$$A^{fast} = P \begin{bmatrix} \lambda_1 & 0 \\ 0 & 0 \end{bmatrix} P^{-1} \tag{2.264}$$

$$A^{fast} = P \begin{bmatrix} 0 & 0 \\ 0 & \lambda_2 \end{bmatrix} P^{-1} \tag{2.265}$$

Before any discretization, this splitting is exact. We will compare the MRI method to the exact solution, and also to standard explicit and implicit method. In order to perform computational experiments we set $\lambda_1 = -100$ and $\lambda_2 = -1$ and use initial condition $y_1^0 = 1$ and $y_2^0 = 1$. The exact solution for these values is

$$\begin{aligned} y_1(t) &= 2.0202e^{-100t} - 1.0202e^{-t} && (2.266) \\ y_2(t) &= -.020202e^{-100t} + 1.0202e^{-t} && (2.267) \end{aligned}$$

In the experiments below, the error is computed via L2 error over the entire time interval. In Table 2.2 the two explicit methods, Forward Euler and RK4 , are unstable at $\Delta t = 0.1$. While RK4 has the best rate of convergence (as expected, it is 4th order accurate), it can be seen that the absolute error of MRI-GARK is quite good. Note that multirate gives equal error in y_1 and y_2, whereas the standard methods have an uneven error. Also note that if the slow term were expensive to evaluate MRI-GARK will be significantly more efficient than RK4.

2.7.4 Verification Results

Recall the MHD tube problem described in the Introduction to this section. The compression of a cylindrical metal tube by a magnetic field is representative of many electrically-driven EOS experiments. There is no analytical solution to this problem. To perform a convergence study, a highly resolved simulation (in space and time) is generated as a synthetic exact solution. This is often referred to as a self-convergence study. In this study the computation mesh is refined, as well as the time time step, by factors of 2. In addition, the multirate factor M is varied. Recall a multirate factor of $M = 8$ means there is one slow physics solve per 8 hydrodynamics time step, and since the MHD solve is quite expensive this equates to a factor of $8x$ speed-up.

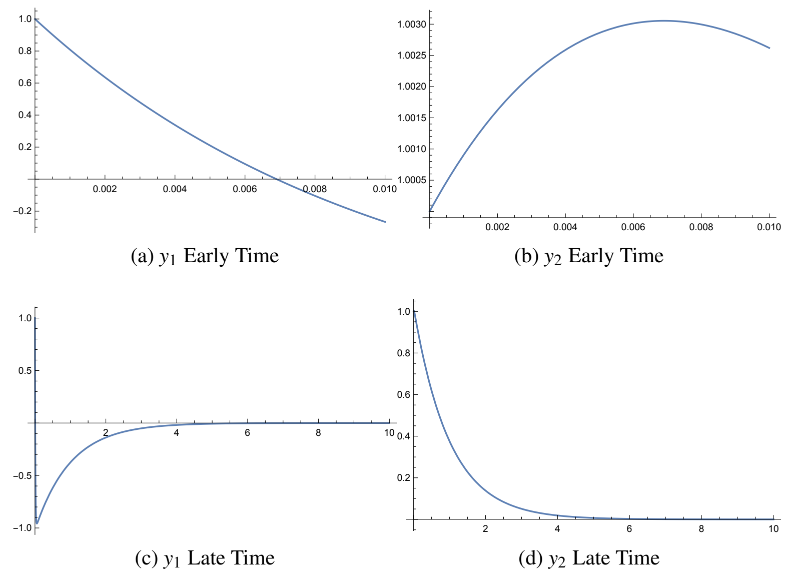

(a) y_1 Early Time
(b) y_2 Early Time
(c) y_1 Late Time
(d) y_2 Late Time

Figure 2.31: An example two-scale ODE that can be integrated using multirate time integration.

	$\Delta t = 0.1$	$\Delta t = 0.01$	slope
Forward Euler	*	0.0799/0.0016	1.3
RK-4	*	0.001917/0.0000191	4.5
Midpoint	0.5713/0.0057	0.009004/0.000090	2.03
DIRK-2	0.3598/0.00359	0.00453/0.00004	2.3
DIRK-3	0.2977/0.00297	0.00299/0.000029	3.01
IM2-EX2	0.000123/0.000123	0.000002/0.0000012	1.96
trapezoidal MRI-GARK	0.000251927/0.000251927	0.0000024/0.0000024	2.01

Table 2.2: Error for various methods, computed over the interval $t \in [0,1]$. The error is for both y_1/y_2. The trapezoidal MRI-GARK method was implemented in the MARBL code.

A second verification of the multirate time integration is the Taylor-Green-Brunner radiation hydrodynamics problem. This problem is run in pure Lagrange mode with a prescribed velocity according to the equations for a Taylor-Green vortex. The radiation source is such that it should exactly cancel the hydrodynamic pressure, the resulting density should remain constant. The motion is shown in Figure 2.34. Like the MHD tube problem above, this problem is run with different multirate factors of $M = 1,2,4,8$. In each case, the mesh and the time step are reduced by factors of $2\times$, and the $L2$ error is computed. The convergence is shown in Figure 2.35, with exhibited second order accuracy. Similarly to the above MHD problem, since the radiation solve is significantly more computationally expensive than the hydrodynamics update, a multirate factor of $M = 8$ is essentially $8\times$ faster than a non-multirate solution.

2.7.5 Multirate Implementation

In the MARBL code the class responsible for computing $\frac{dy}{dt}$ is referred to as a Processing Unit. As per (2.244) the physics is decomposed into fast (hydrodynamics and TN burn) and slow (magnetic diffusion, radiation diffusion, thermal conduction). To facilitate a relatively simple time integration implementation all diffusion Processing Units are derived from a common interface class named ImplictPU. Therefore within the multirate time integrator it is possible to simply iterate over all the Implicit Processing

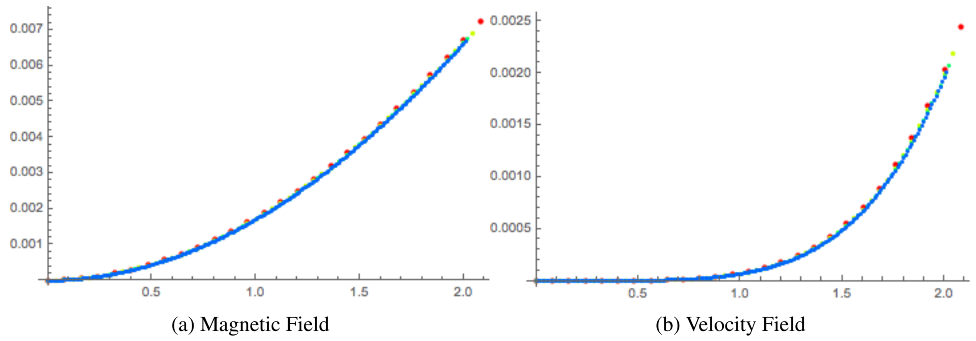


Figure 2.32: MHD fields versus time for the the magnetic tube problem with multirate time integration. Blue M=1, Green M=2, Yellow M=4, Red M=8. For this problem the multirate factor M is essentially the CPU speed-up since the MHD solve dominates the run rime.

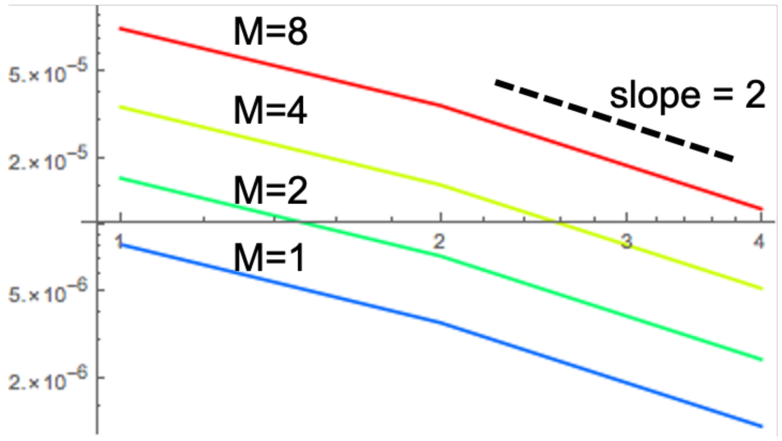


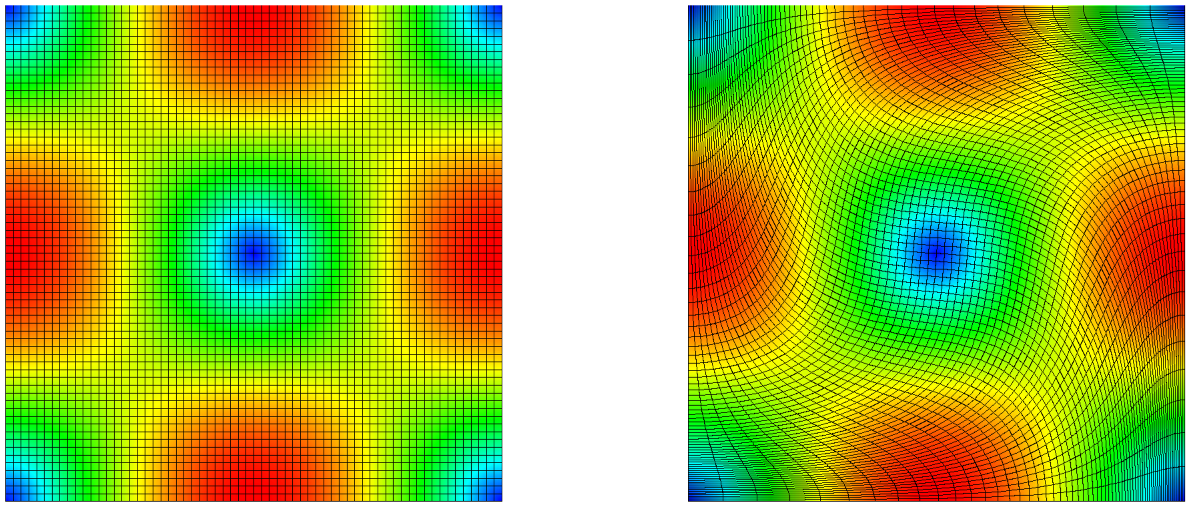
Figure 2.33: Convergence of multirate time integration for various multirate factors for the MHD tube problem. The y axis is the $L2$ error, the x-axis is the refinement level. Note the multirate factor M should not effect the *slope* of the convergence, but does effect the absolute error.

Units.

The Generic Multirate Integrator implementation consists of two key components the *outer iteration* (2.253) and the *inner iteration* (2.251). The outer iteration contains a loop over M hydrodynamic time steps, and a second loop over all the Implicit Processing Units. The inner iteration updates all the hydrodynamic quantities e.g. mass, position, velocity, energy, and material state. Within the inner hydrodynamic steps the slow state fields (magnetics, radiation, thermal) are interpolated.

Pseudocode for the outer iteration is shown below:

```
\\
\\ Outer Iteration of Generic Multirate Integrator
\\
    for each cycle do:
    {
      determine stable hydro time step dt;
      determine best M to use;
      for M steps do:
```



(a) Velocity $t = 0$ (b) Velocity $t = \tau$

Figure 2.34: Taylor-Green vortex with radiation. This problem has an exact analytical solution. The velocity field is completely prescribed, and there exists a radiation source such that the full radiation-hydrodynamics solution should have a constant density.

```
      HydroPU.time_step(t,dt);         \\ integrate hydro quantities from t to t+dt
    for each ImplicitPU do:
      ImplicitPU.ImplicitSolve(t,M*dt) \\ implict solve for each slow physics
      ImplicitPU.UpdateImplicitState
    if (ale time) perform ale;
    if (restart time) write restart/vis files;
  }
```

Pseudocode for the inner iteration (in this case a variant of RK2) is shown below:

```
\\
\\ RK2 integration of hydrodynamics
\\
    evaluate dv_dt;      // 1/2 step
    update velocity;
    update indicators;
    evaluate de_dt;     // 1/2 step
    update energies;
    update masses;      // for TN burn
    update position;
    if rate dependent EOS:
      update eos;
    if nonlinear materials:
      update material models;
    for each ImplictPU:
      interpolate implicit state variables;
```

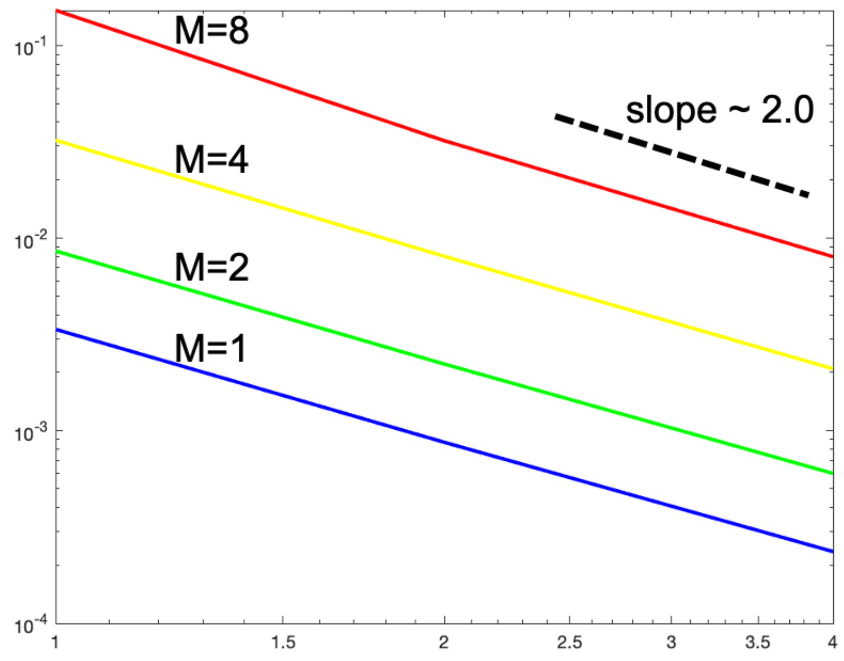


Figure 2.35: Convergence of multirate time integration for various multirate factors for the Taylor-Green vortex with Radiation problem. The y-axis is the $L2$ error, the x-axis is the refinement level. Note the multirate factor M should not effect the *slope* of the convergence, but does effect the absolute error.

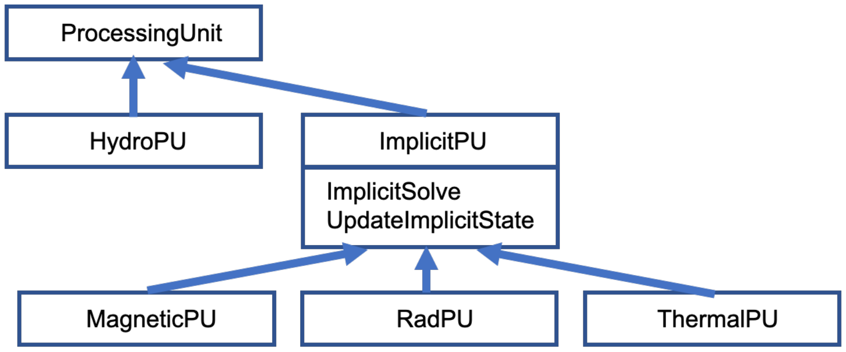


Figure 2.36: Class diagram for MARBL Processing Units. The slow physics e.g. magnetic, radiation, and thermal diffusion are derived from a common Implicit Processing Unit interface.

```
evaluate dv_dt;     // second 1/2 step
update velocity;
update indicators;
evaluate de_dt;     // second 1/2 step
update energies;
update masses;      // for TN burn
update position;
if rate dependent EOS:
  update eos;
if nonlinear materials:
  update material models;
for each ImplictPU:
  update implicit state;
```

2.7.6 Multirate Summary

Multirate time integration methods were originally invented for ODE's such as electrical circuits or chemical reactions. The method-of-lines approach for solving PDE's is based on 1) discretize in space (in our case using higher order finite elements), 2) integrate the resulting high dimensional system of ODE's in time. As discussed above, some physics must be solved using implicit time integration (due to extreme contrast in material properties) while other physics is best solved explicitly. This motivates the use of IMEX (implicit-explicit) time integration. In addition, since there can be vast differences in time scales between the "slow" and "fast" physics, a multirate method is beneficial. In MARBL, hydrodynamics and TN burn is considered fast physics, while magnetic diffusion, radiation diffusion, and thermal diffusion are considered slow physics. The trapezoidal MRI-GARK method is a second order implicit-explicit method with a parameter M that we call the multirate factor, in simple terms there are M fast hydrodynamic updates per one slow implicit update. This gives the code user complete control over accuracy vs efficiency, in early times it is possible to use a large value of M with significant speed-up and little loss of accuracy, while at later times we can reduce to $M = 1$ which is equivalent to a standard Runge-Kutta method. The multirate time integrator is demonstrated to be second order accurate for any value of multirate factor M , where the multirate factor M controls the frequency of slow implicit solves, and thus the accuracy of the slow physics. Since by ansatz the slow implicit physics is computationally expensive, the multirate factor M also controls the overall efficiency of this simulation.

2.8 Constitutive 0D Material Models

2.8.1 Material strength models

MARBL's elastoplastic material modeling capability is currently implemented in a statically linked library, named Leilak. Encapsulating the strength capability in a library implementation was adopted to avoid duplication of effort in both the direct Eulerian and ALE hydro packages since both need the high-pressure, high-temperature, and high strain rate material constitutive models that are implemented in the library. An additional advantage to the use of a shared implementation is that it helps to cushion the effects of the rapidly evolving landscape for hardware architecture, and high-bandwidth hierarchic memory, given that an updated library implementation for new platforms need not be repeated in client codes.

The Leilak library is interoperable with the Fortran based direct Eulerian hydro package (Miranda), and the high-order ALE hydrodynamics package (Blast), even though these codes are implemented in disparate Fortran and C++ programming languages respectively. Interoperability was accomplished using the LLNL-developed Shroud library that enables codes in different languages to exchange run-time data.

Support for material plasticity in MARBL is based on Wilkins' hypoelastic stress evolution model. The implementation can currently account for both purely elastic, and mixed elasto-plastic material flows regimes. Material strength is accounted for via the traceless deviatoric component of the Cauchy stress tensor. The complimentary component of the total stress tensor is the isotropic or hydrodynamic pressure, which is calculated from an equation of state library.

The components of the symmetric deviatoric stress tensor are treated as additional state variables that are evolved at each time step, using the well-known Jaumann objective stress rate measures to ensure that rigid body motion do not generate artificial stresses for zero strain. For purely elastic deformation, deviatoric stress tensor σ is directly proportional to the deviatoric strain tensor ε, and the proportionality constant is the Shear Modulus (G):

$$\sigma = 2G\varepsilon \tag{2.268}$$

Implemented plasticity models are employed for computing the flow and shear stresses at quadrature points, which are then compared to a von Mises yield criterion.

The 0-D constitutive plasticity models that are currently supported in MARBL are:

Elastic Perfectly Plastic (EPP) model

The is the simplest of the available constitutive strength models. The initial yield stress, Y_0, and the corresponding shear modulus G_0 are specified as constant values, where Y_0 is estimated from experimental stress strain curves, whilst the shear modulus Y_0 is calculated from the poisson ratio, and the elastic modulus, E:

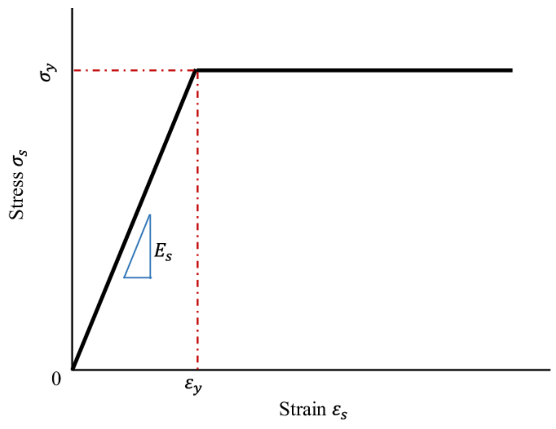
$$G = \frac{E}{2(1+\nu)} \tag{2.269}$$

where ν is the Poisson ratio.

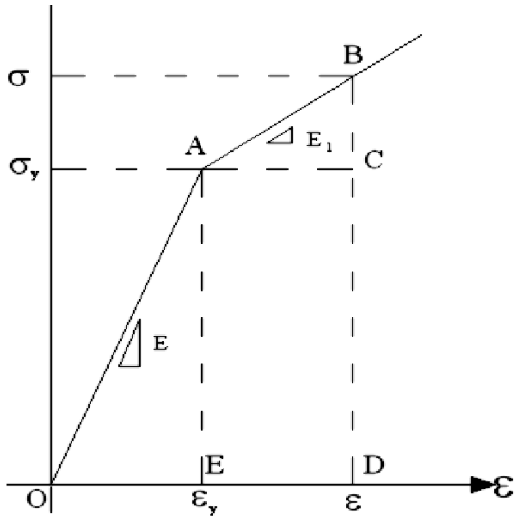
The yield surface is fixed for the EPP model at Y_0 as shown in Figure 2.37(a), i.e., model does not account for strain hardening, and plastic flow or strain is initiated at Y_0.

Elastic Linearly Plastic (ELP) model

The ELP model accounts for strain hardening, and it is characterized by two slopes - an elastic, and hardening slope regions as shown in Figure 2.37(b). The hardening slope, E_1, in Figure 2.37b represents the strain-hardening that is generated to isotropically expand the yield surface when the initial yield stress, Y_0, is exceeded.



(a) Elastic perfectly plastic stress-strain curve.



(b) Elastic linearly plastic stress-strain curve.

Figure 2.37: Elastic-Plastic stress-strain curves.

Steinberg-Guinan (SG) Model

The Steinberg-Guinan (SG) model is a semi-empirical plasticity model for flow stress and shear modulus that was formulated from extensive experimental campaigns for FCC (face-centered cubic), BCC (body-centered-cubic), and HCP (hexagonal closed-packed) materials that are subjected high strain rates due to blast and/or high impact loading. The formulation accommodates high pressure loading, and thermal softening, and it is suitable for strain rates of about 10^3 - $10^7 s^{-1}$. The SG model takes the form of a Taylor's series expansion, where the shear stress, G, and the flow stress Y are given by:

$$G = G_0\left[1 + A\left(\frac{P}{\eta^{1/3}}\right) - B\left(T - T_{ref}\right)\right]\exp\left(-\frac{y_p E}{E_m - E}\right) \tag{2.270}$$

$$Y = Y_0\left[1 + \beta\left(\varepsilon + \varepsilon_0\right)\right]^n \frac{G\left(P,T\right)}{G_0} \tag{2.271}$$

The compression ratio η, is given by: $\eta = \frac{\rho}{\rho_0}$

A is the pressure dependence of the shear modulus, and B is the temperature dependence of the shear modulus. T_{ref} is the reference temperature, which is typically set to 300K. Y_{max} is the maximum work hardening stress, ε_0 is the initial plastic strain if applicable. β is the SG model work hardening parameter, and Y_0 is the yield stress at Hugoniot elastic limit. P and T are the pressure and temperature respectively.

The SG model, which is a rate-independent elastic-plastic model, assumes that a value of strain rate $\dot{\varepsilon}$ exists beyond which the strain rate has a minimal effect on the yield stress, Y, so that equivalent plastic strain rate can be neglected. This is an assumption typically holds at high stress level.

Preston-Tonks-Wallace (PTW) model

The PTW model is useful for flow stress at extreme plastic strain rates (strain rates of up to $10^{12}s^{-1}$). This is a flow stress condition that is to be expected for shocked and reshocked flow regimes. The model assumes that the plastic flow stress is a function of the equivalent plastic strain, ε_p, the equivalent plastic strain rate, $\dot{\varepsilon_p}$, the material temperature T, and density ρ. The model ignores the load or stress history of the material. Material flow stress is given by:

$$Y = 2\hat{\tau}G\left(P,T\right) \tag{2.272}$$

where

$$\hat{\tau} = \hat{\tau}_s + \frac{1}{p}\left(s_o - \hat{\tau}_s\right)\ln\left\{1 - \left[1 - exp\left(-p\frac{\hat{\tau}_s - \hat{\tau}_y}{s_o - \hat{\tau}_y}\right)\right]\exp\left(-\frac{p\theta\varepsilon^p}{\left(s_o - \hat{\tau}\right)\left[exp\left(-p\frac{\hat{\tau}_s - \hat{\tau}_y}{s_o - \hat{\tau}_y}\right) - 1\right]}\right)\right\} \tag{2.273}$$

$$\hat{\tau}_y = \max\left\{y_o - \left(y_o - y_\infty\right) erf\left[\kappa\hat{T}\ln\left(\frac{\gamma\dot{\xi}}{\dot{\varepsilon}^p}\right)\right], \min\left[y_1\{\frac{\dot{\varepsilon}^p}{\gamma\dot{\varepsilon}}\}^{y2}, S_o\{\frac{\dot{\varepsilon}^p}{\gamma\dot{\varepsilon}}\}^{\beta}\right]\right\} \tag{2.274}$$

$$\hat{\tau}_s = \max\left\{S_o - \left(S_o - S_\infty\right) erf\left[\kappa\hat{T}\ln\left(\frac{\gamma\dot{\xi}}{\dot{\varepsilon}^p}\right)\right], S_o\{\frac{\dot{\varepsilon}^p}{\gamma\dot{\varepsilon}}\}^{\beta}\right\} \tag{2.275}$$

The PTW model, like the SG model, has a model for shear stress:

$$G(\rho, \bar{T}) = G_0(\rho)(1 - \alpha \bar{T}) \tag{2.276}$$

where α is the temperature-dependent term for material:

$$\hat{T} = \frac{T}{T_m(\rho)} \tag{2.277}$$

$$\hat{\xi} = \frac{1}{2}\left(\frac{4\pi\rho}{3M}\right)^{1/3}\left(\frac{G}{\rho}\right)^{1/2} \tag{2.278}$$

The SG formulation can also be employed as a surrogate model for shear stress, although this option is yet to be supported.

Johnson-Cook (JC) model

Johnson-Cook model is an empirical strain rate-dependent flow stress model. The flow stress at spatial positions of interest is calculated using:

$$Y = (A + B\varepsilon_n)\left(1 + C\ln\frac{\dot{\varepsilon}}{\dot{\varepsilon}_0}\right)(1 - T^{*m}) + C_p p \tag{2.279}$$

where $T^* = \frac{T - T_{ref}}{T_{melt} - T_{ref}}$. β. m and n are the exponent on the temperature term, and work hardening exponent respectively. A is a fixed term contribution to strength, B is the strain hardening coefficient, and C is the multiplier on the lon strain rate term. C_p is the linear pressure dependence parameter. T_{melt} is the melt temperature, and $\dot{\varepsilon}$ is a strain-rate normalization factor.

The JC model uses SG as its surrogate shear stress model, since the model was only formulated for yield or flow stress.

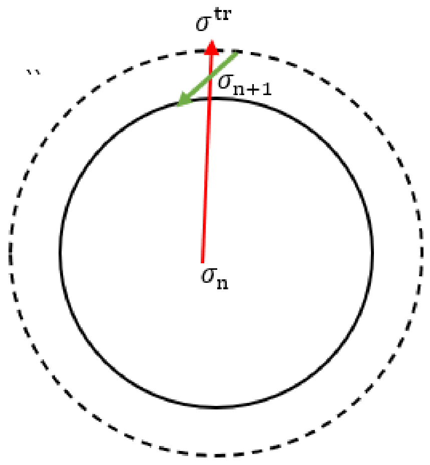


Figure 2.38: Graphical illustration of the concept of radial return

Radial Return algorithm

The radial return algorithm is a widely used procedure to account for plastic flow, or strain-hardening for material that is subjected to either an elastic-plastic, or a visco-plastic stress regime. In a radial return algorithm, the second invariant, J_2, or eigenvalue of a trial deviatoric stress tensor, σ^{tr}, is first computed, and compared to a von Mises yield stress criterion:

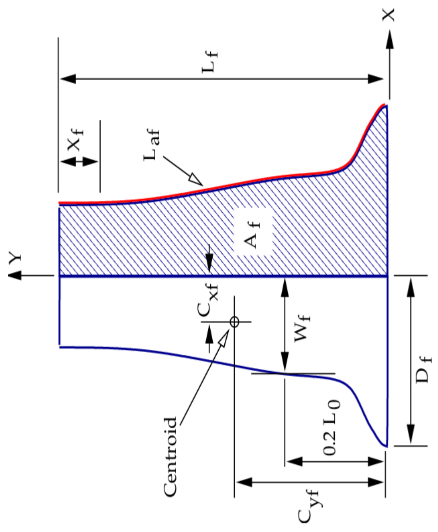
$$J_2 \leq \frac{1}{3}Y^2 \tag{2.280}$$

A von Mises yield criterion is shown graphically in Figure 2.38, where the stress return path (green arrow) to yield surface is shown to be non-orthogonal to the yield surface for the purpose of clarity, as the implemented algorithm assumes a return path that is normal to yield surface.

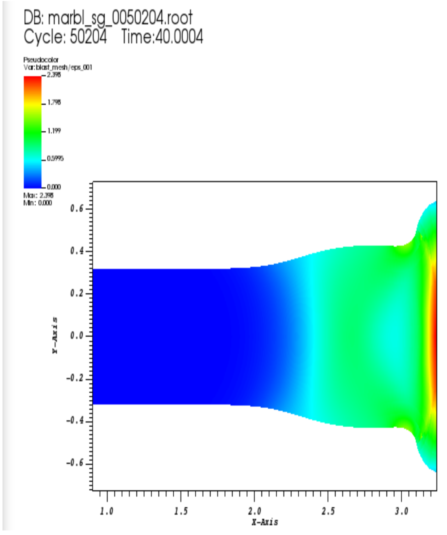
Calculated trial stress levels that exceed the current yield surface are restituted or "returned" to the yield surface as depicted in Figure 2.38. At the next time-step, a new "yield surface" is calculated using an equivalent plastic strain increment, and a hardening slope. The above check and return-to-yield-surface is repeated at the next time step if the current elastic limit is again exceeded.

Illustrative examples using SG, PTW, and JC models

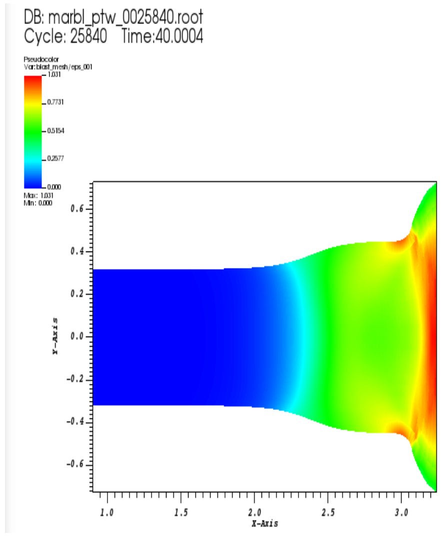
The fidelity of the implemented elasto-plastic constitutive models were verified using the well-known Taylor's anvil impact test problem. In the Taylor impact test, a cylindrical metal specimen traveling at some velocity is directed to impact a fixed frictionless wall, which results in a mushroom-like plastic deformation of the end of the cylinder that impacts the wall as shown in Figure 2.39a. For replication of this profile using the MARBL code, an OFHC (oxygen-free high conductivity copper) cylindrical object traveling horizontally to the right at a speed of 227 m/s is made to impact a fixed solid wall. Elastic-plastic deformation profile plots using SG, PTW, and JC models are illustrated in Figure 2.39b, Figure 2.39c and Figure 2.39d. Comparison of the computed profiles to the expected deformation profile in Figure 2.39a clearly demonstrates that use of these models can replicate the three principal features of the expected deformed shape: a base mushroom or webbed feet at the point of impact with the wall, an enlared twinnig zone immediately upstream of the mushroom end, and an elastic zone (the blue portions of Figure 2.39b- Figure 2.39d), with a length that is dependent on the travel velocity, and the duration of the impact. Two deformation metrics for all three implemented models, i.e., the ratio of the final radial distance of the mushroom end to the original radius of the cylinder, r_f/r_0, and the ratio of the final length of cylinder to the original length of the cylinder, L_f/L_0. are computed to within 8% of published experimental data.



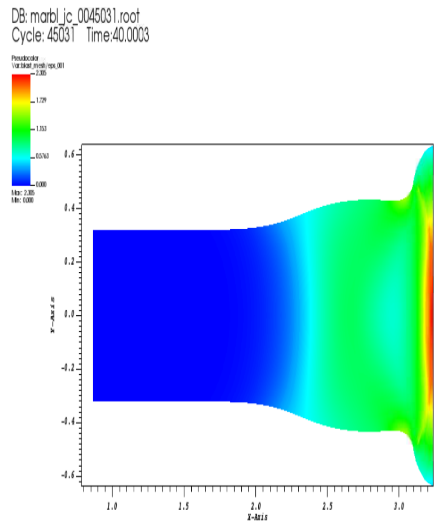
(a) Idealized Taylor impact deformation profile.



(b) Elastic-plastic deformation profile for copper rod using Steinberg-Guinan model.



(c) Elastic-plastic deformation profile for copper rod using Preston-Tonks-Wallace model.



(d) Elastic-plastic deformation profile for copper rod using Johnson-Cook model.

Figure 2.39: Elastic-plastic deformation profile for Taylor Anvil impact problem using MARBL ALE hydrodynamics. Impact velocity = 0.227 km/s.

Chapter 3

Computer Science

3.1 Introduction

Our computer science infrastructure was designed around the need to support modular physics, math and computer science packages. A main goal was to make it easy to incorporate and develop modular packages within the MARBL codebase. Due to the significant risks involved in basing our physics and math around novel high order discretizations, we mitigated our computer science risks by adopting best practices honed over many years in LLNL's multiphysics codes. However, rather than develop new internal infrastructure siloed within our code base, we opted to develop as much as possible as open source toolkits and performance portability libraries that were able to focus on clean, well tested, reusable code. We discuss the design, development and incorporation of Axom, LLNL's HPC computer science infrastructure toolkit in Section 3.2.

For our project's internal infrastructure that did not make sense to share, we designed *Exo*, our simulation's exoskeleton. Exo provides an abstract "Main" class as well as extensible `Simulation` and `Package` classes. We discuss this in Section 3.3.

One of our project's key risks is related to developing an application around high order discretizations. As depicted in Figure 3.1, the HPC ecosystem has many complex interrelated features and workflow tools that developers and users expect when working with a multiphysics simulation code. In addition, many of these capabilities only operate on low order meshes. In our aim to support feature parity as early as possible, we developed a multi-tiered strategy for incorporating these capabilities, with different advantages and risk levels. While our eventual goal is to provide native high order capabilities, we leverage low order refinement (LOR) to approximate our meshes with linear meshes. We discuss this strategy in Section 3.4 and highlight a use case for transferring problems between our Lagrangian hydro package (Blast) and our Eulerian hydro package (Miranda).

We carefully designed our modular code base to facilitate integration of modular packages. Specifically, we strove to make it straightforward for developers who are not intimately familiar with our codebase to integrate their packages quickly. As such, our build system (Section 3.5) was designed to streamline the process of accessing, building, developing and testing the code and to support rapid integration and simplified testing and upgrading of third party libraries. In anticipation of the code's rapid growth, we also prioritized software quality assurance (SQA) needs, such as extensive continuous integration (CI) testing, covering regressions, build configurations, memory leaks and performance. We discuss this and how it has been instrumental to our continued confidence in the code's results in Section 3.6.

Our performance portability abstractions were developed using open source, shared components. Namely, we rely on RAJA for portable C++ loop abstractions and OpenMP for portable Fortran loop abstractions that can run efficiently on different hardware and Umpire for portable memory allocations and deallocations that can be shared among different physics libraries. We discuss these in Section 3.7 in the context of a cross platform performance scaling study that scales up to half of the Sierra supercomputer cluster and to all of the Astra cluster (Section 3.8). In Section 3.9 we examine our code's power usage, which we tracked as we ported the code to Sierra. We also compare the energy requirements for running the same simulation on different compute

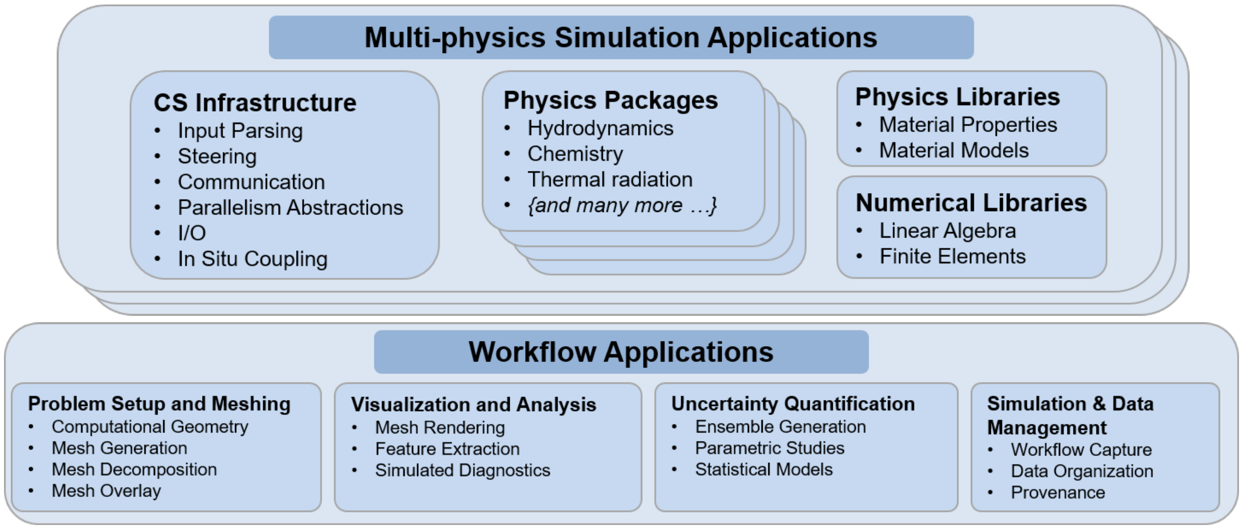


Figure 3.1: There are many capabilities in the HPC ecosystem that users expect from a multiphysics code. Although many of these currently only support low order codes, our computer science infrastructure allows us to plug into this rich HPC ecosystem.

platforms.

We have devoted considerable effort within our project into supporting efficient workflows to help our users with setting up, running and analyzing the results of their simulations. Specifically, we discuss workflows for setting up problems setup including our Intermediary Representation for input decks (IREP), high order mesh setup via PMesh and high order tracer particles in Section 3.10. In Section 3.11 we discuss how our code can be steered at the lua prompt and interactively via Jupyter notebooks. We then discuss analyzing outputs in Section 3.12.

We conclude this chapter by assessing how well we are delivering on our developer workflow goals with the results from a survey to 12 developers outside our core team that have integrated modular computer science, math and physics packages into our codebase over the past five years (Section 3.13).

3.2 Modular CS

The LLNL ATDM effort began in 2014 with an extensive requirements gathering effort that would form the basis for the LLNL next-generation code development strategy. Three semi-autonomous working groups were dispatched to gather input and seek recommendations from code developers and users in the areas of physics, user work flow, and computer science. The goal was to define recommendations for code development that emphasize flexibility to continuously maintain the ability to react to changes in mission drivers, code capabilities, and computer architectures. The *Computer Science Recommendations for LLNL ASC Next-Gen Code* report [1] that resulted from this effort contains a collection of 50 computer science recommendations that we follow in our development of modular CS infrastructure software that supports MAPP. The recommendations cover the following areas:

- Software Design Goals
  - Software Architecture
  - Software Components
- Productivity
  - Software Development and Processes

  - Project Management
  - User Workflow and Experience

- Performance and Architecture Portability
  - Programming Models and Parallelism
  - Co-design, External Interactions, and Research Areas

3.2.1 Modular CS Infrastructure Goals

The modular CS infrastructure discussed in this chapter addresses four high-level goals that meet the recommendations in the aforementioned report:

1. Make applications high-performance and portable across a wide range of computing platforms.
2. Simplify management and sharing of mesh-based data across codes.
3. Employ common development tools and practices across our software ecosystem.
4. Collaboratively develop and share modular software building blocks across applications.

High-performance and portability is important because:

- Application performance significantly impacts user productivity.
- Porting applications to new platform architectures consumes a lot of developer resources.
- Developing and maintaining platform-specific versions of codes is untenable.
- Portable, single-source applications enable running on all relevant platforms and reduces effort required to port to new platforms.

To meet these demands, we collaboratively develop software abstractions that enable portable execution and memory management with application project teams. Contributors include: CS researchers, application developers, academics, and vendors. Applying shared abstractions across codes facilitates sharing of implementation patterns, performance tuning expertise, and portability best practices. Our approach to performance portability is described in Section 3.7.

Simplified sharing of mesh-based data across codes is important because:

- Mesh data sharing is fundamental to coupling physics packages in integrated applications and is a critical part of user workflows.
- Incompatible data representations impede data sharing between applications and the use of common visualization and analysis and other tools.

We have established conventions for describing mesh-based simulation data and are using these conventions to connect applications and tools with common interfaces. This approach has been fundamental to MAPP from its start. We are also evolving mature applications to use common interfaces rather than multiple diverse interfaces.

Using shared development tools and practices across our software ecosystem is important because:

- It lowers barriers to cross-project collaboration and enables a more sustainable, rapid development HPC software ecosystem.

- It reduces the costs of integrating new software and code maintenance on HPC platforms.
- Sharing effort and expertise reduces the burden on individual teams and frees up energy for other important development work.
- Shared build components and CI processes and tools allows for rapid bug fixes and software improvements.

We are building a shared software development environment with basic tools that projects commonly use "in the box". These tools contain expert knowledge in building code with different languages and programming models and are easily adopted across our diverse set of applications.

Collaborative development of shared modular software building blocks is important because:

- Applications have many foundational needs in common, such as: in-memory data management, file I/O, spatial queries and indexing schemes, visualization and analysis, etc.
- Replicating code-specific capabilities has high development costs for application teams and are harder to vet, maintain, and share.
- HPC platform diversity requires sharing CS expertise across projects given limited developer resources.

To broaden the base of contributors to our efforts beyond traditional application teams, we are building shared capabilities as open source projects. Our productized software toolkits provide foundational building blocks that all codes can leverage and customize for specific use cases. In the next several sections, we describe key modular software components we have developed and how we apply them in MAPP.

3.2.2 Axom CS Components

Axom [34] is an open source library of robust, flexible software components that provide a variety of building blocks for multi-physics applications and computational tools. A key objective of Axom is to facilitate integration of novel, forward-looking shared computer science capabilities into simulation codes. The initial focus of Axom development was to provide core infrastructure for MAPP. The success of this effort has been demonstrated and reported on in multiple ATDM Level 2 milestones in recent years. Axom has also been adopted by multiple other WSC applications at LLNL.

Although written in C++, Axom components provide native interfaces in C++, C, Fortran, and Python along with mechanisms to ensure inter-language data consistency. We developed a simple-to-use multi-language interface generation tool called *Shroud* [35] in Axom. Subsequently, Shroud has been made an open source project and is used in a variety of software projects.

Sidre: In-memory datastore.

The Axom *Sidre* (**Si**mulation **d**ata **re**pository) component provides capabilities to centralize data management in HPC applications, such as data description, allocation, access, and so forth. The goal of Sidre is efficient coordination and sharing of data across physics packages and other libraries in integrated applications, and between applications and tools that support file I/O, in situ visualization and analysis, etc. For MAPP, Sidre is the central point of access to mesh-based and other simulation data used in its component physics packages. It provides the application with checkpoint/restart functionality and a common interface for external tools to access data, see Figure 3.2. This is discussed in Section 3.2.4.

The design of Sidre is based on substantial experience with mature LLNL applications and requirements and the enablement of high performance on a diverse range of computer architectures. Applications must carefully manage data allocation and placement to run efficiently. Related capabilities in mature codes were typically developed independently for each code with little regard to

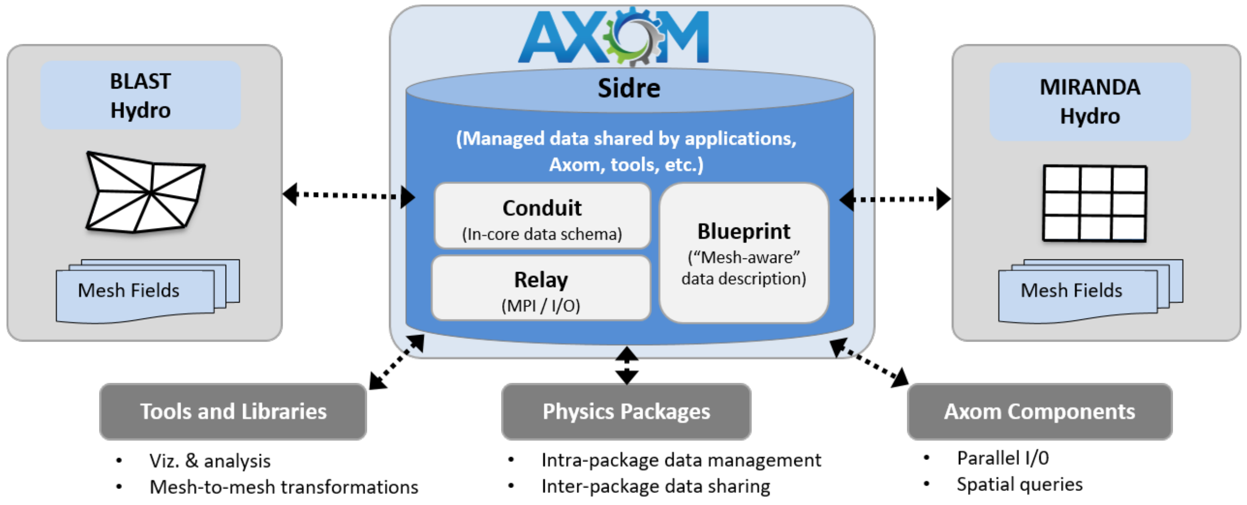


Figure 3.2: Mesh data sharing simplifies the coupling of applications, libraries, and tools.

sharing software or data with other applications. In contrast, Sidre is designed to be used in multiple applications and to facilitate data sharing amongst them.

Sidre provides a variety of data management capabilities, including:

- Tree-structured data hierarchies. Many mesh-based application codes organize data into hierarchies of contexts (e.g., domains, regions, blocks, mesh centerings, subsets of elements containing different materials, etc.). Sidre supports hierarchical, tree-based organizations in a simple, flexible way that aligns with the data organization used in production applications.
- Separate data description and allocation operations, which allow applications to describe their data and then place the data in memory based on performance or algorithmic needs.
- Multiple different *views* into a chunk of (shared) data. A Sidre view includes description semantics to define data type, number of elements, offset, stride, etc. Thus, a chunk of data in memory can be interpreted conceptually in different ways via different views into it.
- Externally-owned *opaque* or *described* data. Sidre can accept a pointer to externally-allocated data and provide access to it by a name identifier. When external data is described to Sidre, it can be processed in the same ways as data that Sidre owns. When data is not described (i.e., opaque), Sidre can provide access to the data via a pointer, but the consumer of the pointer must know type information to do anything substantial with the data. These capabilities are exercised in the MAPP application.
- Attributes or metadata associated with a Sidre view. This metadata is available to user code to facilitate program logic, such as selective writing subsets of data to files.

Internally, Sidre uses the Conduit library to manage data description details, which is described in Section 3.2.3.

Quest: Spatial queries on meshes

The Axom *Quest* (**Que**ries over **s**urfaces **t**ool) component provides tools to perform queries on points in space relative to a surface, a mesh cell, or spatial regions. Such queries include:

Point containment Given an arbitrary point in space and a surface mesh bounding a region of space, determine whether the point lie inside or outside the enclosed volume. This query, also referred to as an In/Out query, forms the basis of our "shaping" algorithm to initialize volume fraction fields for multimaterial simulations, as we describe in Section 3.2.5.

Signed distance Given a point in space and a surface mesh bounding a region of space, return the distance between the point and the closest point on the surface mesh. The sign of the returned value indicates whether the point is inside the enclosed volume (negative) or outside (positive). This query is used by Miranda in the context of immersed boundary applications.

Point-In-Cell Given a point in space and a volumetric mesh (low order or high order), return the index of the cell that contains this point, as well as the parametric coordinates of the point within that cell. This query is at the core of MARBL's tracer particle package, which we discuss in Section 3.10.3.

All Nearest Neighbor Given a list of points, each with an assigned regions label, find the closest neighbor point in each of the other regions.

Quest also provides other capabilities for querying and manipulating surface meshes, such as welding vertices on a surface mesh that are not at the exact same point but are within some specified distance of each other, locating intersecting or degenerate triangles on a surface mesh, and asking whether a surface mesh is a watertight manifold. Such operations and queries are essential for shaping in material regions on a simulation mesh and getting diagnostic information about simulation state.

Quest capabilities are built on top of building blocks that are contained in other Axom components: *Primal*, which contains geometric primitives, such as points, rays, etc., as well as computational geometry operations on pairs of primitives, such as finding the intersection point between a ray and a triangle; and *Spin*, which contains spatial index structures that can be used to accelerate spatial searches. These other components can be used by applications to build custom spatial searches or other computation geometry algorithms.

Slic: Common logging across application components.

The Axom *Slic* (Simple logging interface) component provides a simple, lightweight, and extensible infrastructure for logging application messages. Slic focuses on logging interoperability across constituent libraries in an application. Messages logged by an integrating application and any of its libraries using Slic can have a uniform format and be routed to a common output destination. Slic has built-in *log streams* that support common use cases, such as logging to a file or console output. It allows easy customization of message formats, message handling and filtering, and log streams and message levels, such as error, warning, notice, etc. Slic integrates with the Axom *Lumberjack* component, which supports message logging and filtering at scale via MPI. Similar to other Axom components, Slic provides native interfaces for C++, C, and Fortran applications.

3.2.3 Conduit

Data description and I/O capabilities in Sidre are built on the open source *Conduit* library [36]. Conduit provides an intuitive model for describing scientific data in C++, C, Fortran, and Python. It provides Sidre and the MAPP application support for in-memory data coupling between packages, data serialization, and I/O tasks. Two of the key features that Conduit provides applications is its *data model* and the *Mesh Blueprint*.

Data Model

At the core of Conduit is a data model and an API for building and accessing hierarchical data. The flexible data model is inspired by JSON [37] and provides an intuitive way to describe hierarchical in-memory scientific data. The Conduit API provides a dynamic way to construct and consume hierarchical data objects.

Conduit is built around the concept that an intuitive in-memory data description capability simplifies many other common tasks in the HPC simulation ecosystem. Conduit's core provides a runtime focused in-memory data description API that does not require repacking or code generation. It supports a mix of externally-owned and Conduit allocated memory semantics. As mentioned in Section 3.2.2, these capabilities are fundamental to sharing data amongst packages in the MAPP application.

The primary object in Conduit is a *node*, which supports construction of and access to hierarchical data objects. Each node instance manages a collection of memory chunks. Data can be copied into a node and a node can be extended with new data without changing existing allocations. Following JSON, collections of named nodes are called *objects* and collections of unnamed nodes are called *lists*, all other types are leaves that represent concrete data. Once constructed, one can easily traverse a Conduit node hierarchy (or tree) via straightforward UNIX directory-like relative path syntax; i.e., "path/to/data". Conduit provides generator methods to parse JSON and YAML [38] described data into a node hierarchy. It also provides support to save and restore Conduit nodes to HDF5 files. Sidre uses Conduit's HDF5 support to implement HDF5-based checkpoint restart files for MAPP.

Understanding the precision of underlying data types is essential when sharing data in scientific codes. Thus, Conduit uses well-defined bitwidth-style data types (inspired by NumPy [39]) for leaf nodes. Leaf node data types can be numeric scalars or arrays.

Mesh Blueprint

The flexibility of a Conduit node allows it to be used to represent a wide range of scientific data. However, if used in an unconstrained manner, this flexibility can lead to applications making specific choices about representing their meshes and data that could preclude sharing data between them. The Conduit *Mesh Blueprint* provides high-level conventions for describing simulation meshes and associated data using Conduit nodes so that simulation codes or physics packages that need to share mesh data will understand each other's data. The Mesh Blueprint provides methods to verify whether a node instance conforms to known conventions, called *protocols*, and methods to transform data between protocols as needed.

Currently, Mesh Blueprint supports: uniform and rectilinear structured meshes and unstructured meshes containing elements of triangles, quadrilaterals, polygons in two-dimensions, and tetrahedra, hexahedra, and polyhedra in three-dimensions. It also supports locally-refined and adaptive meshes. Key concepts in the Mesh Blueprint include: coordinate sets, topologies, material sets, fields, species, nesting and adjacency sets, and state.

Figure 3.3(a) illustrates a Mesh Blueprint-conformant Sidre hierarchy for 'tiny mesh', a two-element unstructured hexahedral mesh containing a node-centered field 'nodefield' and an element-centered field 'eltfield'. This dataset can be loaded into VisIt, using the native Mesh Blueprint reader. Figure 3.3(b) shows a pseudocolor plot rendering of the node-centered 'nodefield'.

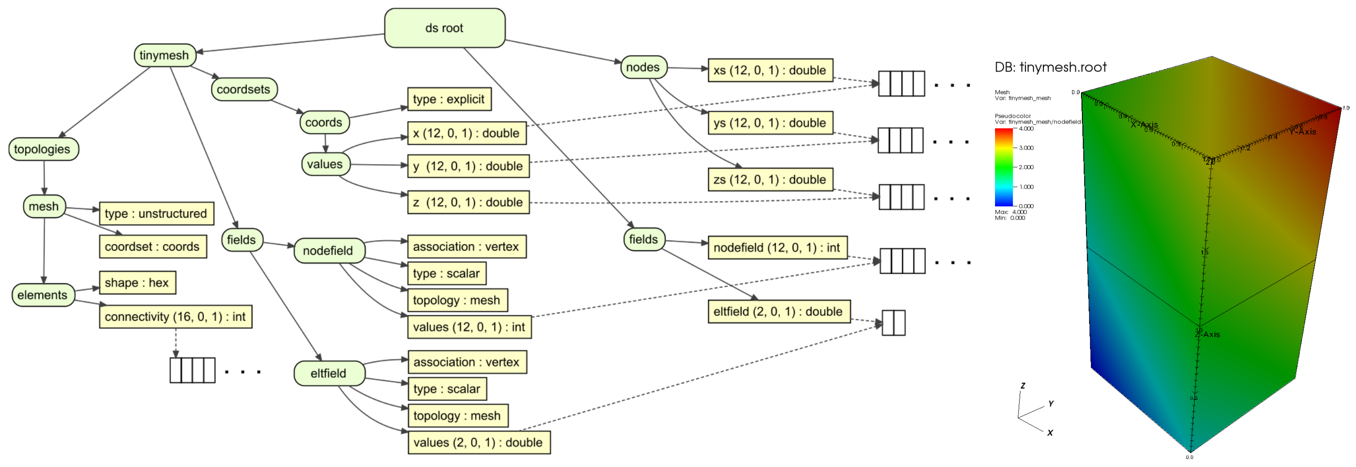
3.2.4 Support for checkpoints and restarts

For our FY16 L2 milestone, we integrated Axom into MARBL by centralizing our mesh data and fields in a Sidre data store to support checkpoint and restart capabilities in the code.

The differences in our two hydro packages led to two very different ways of using Sidre. For our C++ Lagrangian package, Blast, we opted to let Sidre own and manage our mesh and field data. Being high order, Blast's field data is based on mfem's GridFunction abstraction, a class that wraps an array and controls access to the underlying data. We developed a class `SidreDataCollection` derived from mfem's `DataCollection` and several utility function to manage the calls to Sidre to allocate the data, construct the class and set the underlying pointer appropriately. From the developer's perspective, we require only a single line of code to allocate a field and add it to the restart state:

Listing 3.1: Allocating and registering a restart field in a single line of code.

```
DataCollectionUtility::AllocateGridFunc(gf,     // pointer to grid function wrapping field data
                                         fes,    // pointer to finite element space
                                         dc,     // pointer to data collection for the Datastore
```



(a) Sidre hierarchy for 'tiny mesh', a simple 2-element hexahedral mesh (b) Rendering 'tiny mesh' in VisIt

Figure 3.3: (a) Sidre description of a simple Blueprint conformant hexahedral mesh, 'tiny mesh'. (b) Pseudocolor rendering of a nodal field 'nodefield' from 'tiny mesh' using VisIt's native Mesh Blueprint reader.

```
                                    name); // name for this field in Sidre
```

In contrast, for our Fortran Eulerian hydro package, Miranda, we opted to have Miranda own the data and only register the mesh and field arrays with Sidre. This allowed us to use Fortran's `Allocatable` arrays, for potentially improved performance via Fortran compiler optimizations. This implementation leveraged Sidre's *Shroud*-generated Fortran bindings to describe and register the fields with Sidre.

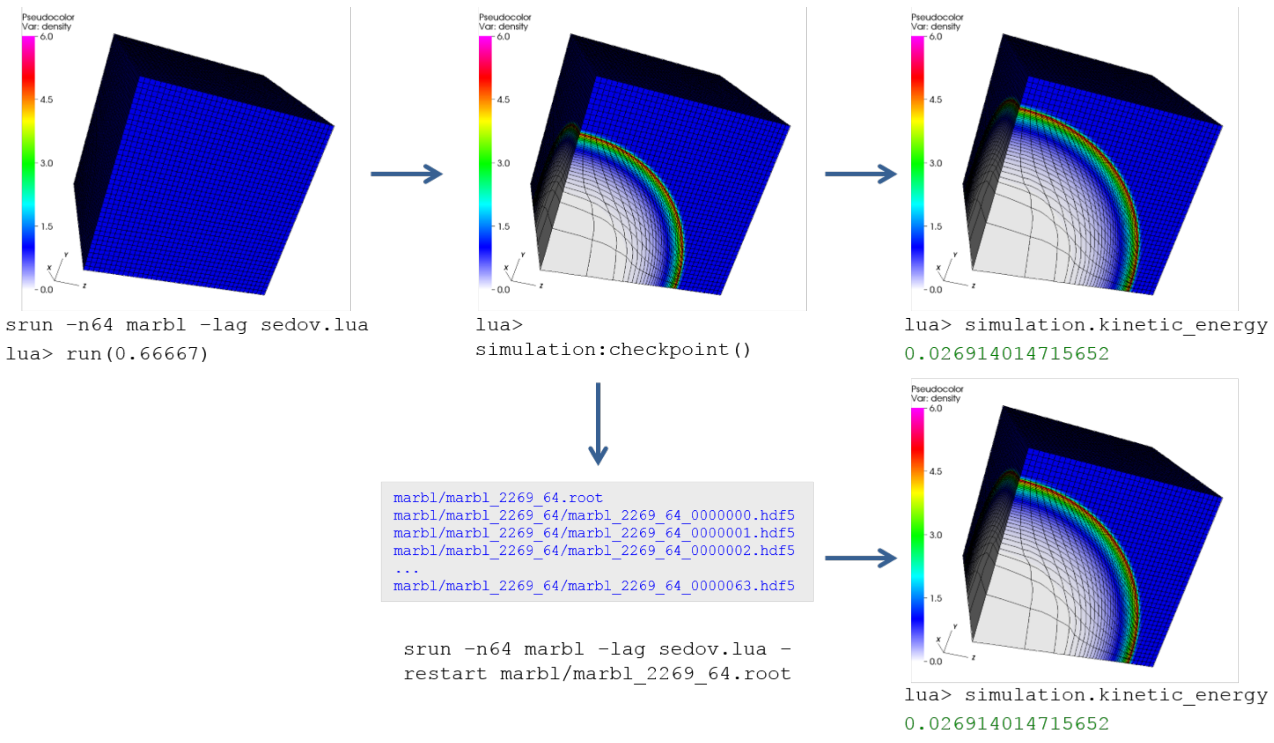
As demonstrated in our FY16 milestone final report, the overhead for using Sidre to centralize our mesh and field data was negligible. Figures 3.4 and 3.5 illustrate checkpoints and restarts when running MARBL in Lagrangian and Eulerian mode, respectively. As can be seen in the figures, our restart runs generate identical results as the non-restart runs. Note that our checkpoint files are output in the Mesh Blueprint format, and these figures were generated from checkpoint files using the VisIt Mesh Blueprint plugin.

3.2.5 Shaping complex regions into multimaterial simulation

Another early success of our shared computer science infrastructure came in the form of an accelerated query for "shaping" materials with arbitrarily complex boundaries into simulation meshes. Recall from Section 2.2.4 that our multimaterial model utilizes material indicator functions to encode the material decomposition within an element. This section discusses how we initialize high order material indicator functions (i.e. volume fractions) in multimaterial simulations. Specifically, we discuss how we shape an arbitrary triangulated surface meshes bounding a volume of space into a multimaterial simulation [40].

We build an In/Out quadtree over the mesh triangles. This is a spatial index that allows us to determine if a point is inside or outside. It uses three types of octree blocks: White: determined to be outside when generating the tree; Black: determined to be inside when generating the tree; Gray: this block contains a limited amount of geometry; we need to run some additional tests. Figure 3.7(b) shows an octree built over the 'plane.stl' triangle mesh model. In MARBL, we replicate each shape over each rank. Then, on each rank, we query the tree using points from the elements on that rank.

We use a sampling approach followed by L2 projection to initialize the volume fraction fields. For each quadrature point within an element, we check if the physical coordinates of the quadrature point are inside or outside. We then use L2 projection to solve for the DOFs. Figure 3.7(a) illustrates sampling at quadrature points (blue 'x's) mapped into physical space via map $\Phi(\hat{x})$. Points



```
srun -n64 marbl -lag sedov.lua
lua> run(0.66667)
```

```
lua>
simulation:checkpoint()
```

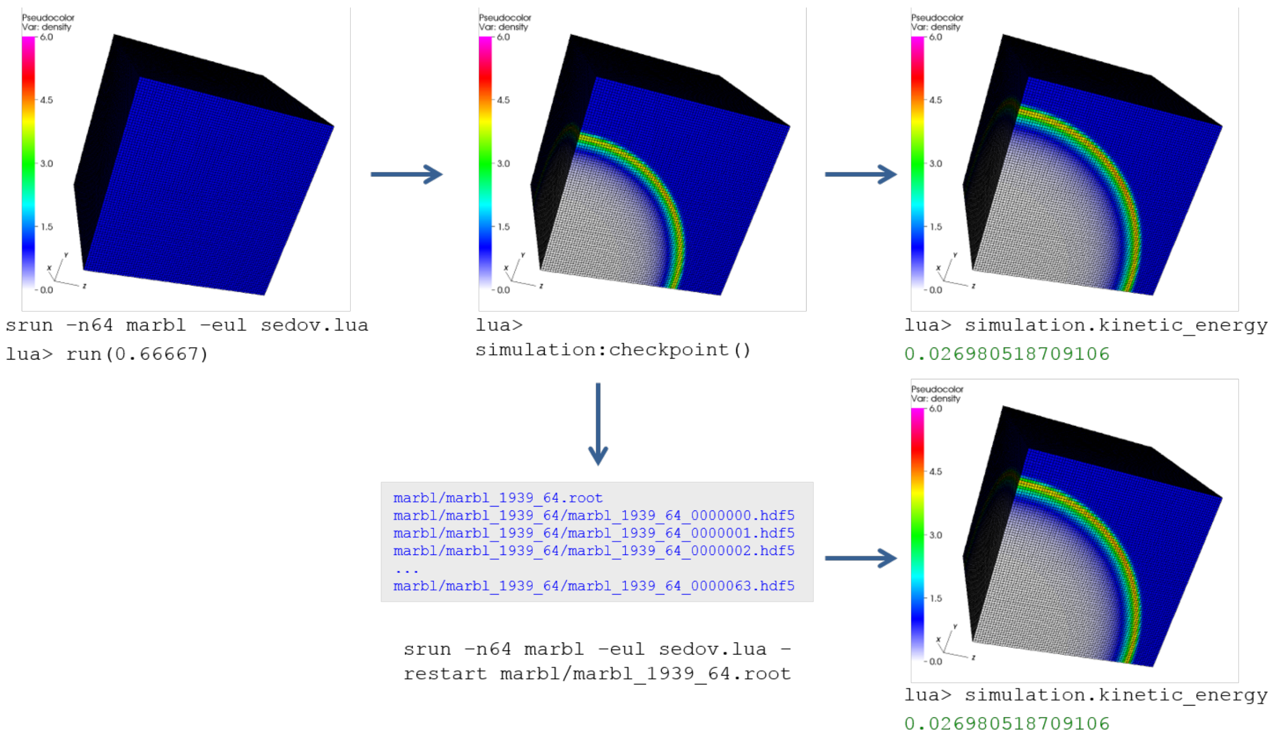
```
lua> simulation.kinetic_energy
0.026914014715652
```

```
marbl/marbl_2269_64.root
marbl/marbl_2269_64/marbl_2269_64_0000000.hdf5
marbl/marbl_2269_64/marbl_2269_64_0000001.hdf5
marbl/marbl_2269_64/marbl_2269_64_0000002.hdf5
...
marbl/marbl_2269_64/marbl_2269_64_0000063.hdf5
```

```
srun -n64 marbl -lag sedov.lua -
restart marbl/marbl_2269_64.root
```

```
lua> simulation.kinetic_energy
0.026914014715652
```

Figure 3.4: Checkpoints and restart on the Sedov problem in `MARBL -lag`.



```
srun -n64 marbl -eul sedov.lua
lua> run(0.66667)
```

```
lua>
simulation:checkpoint()
```

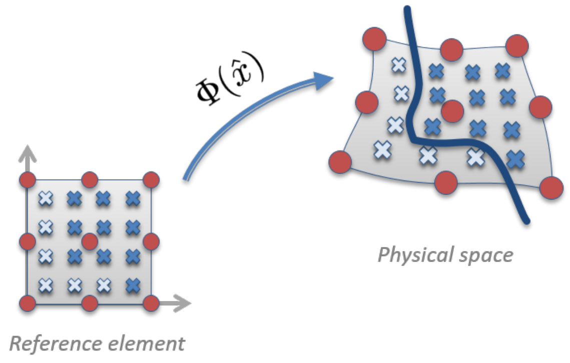
```
lua> simulation.kinetic_energy
0.026980518709106
```

```
marbl/marbl_1939_64.root
marbl/marbl_1939_64/marbl_1939_64_0000000.hdf5
marbl/marbl_1939_64/marbl_1939_64_0000001.hdf5
marbl/marbl_1939_64/marbl_1939_64_0000002.hdf5
...
marbl/marbl_1939_64/marbl_1939_64_0000063.hdf5
```

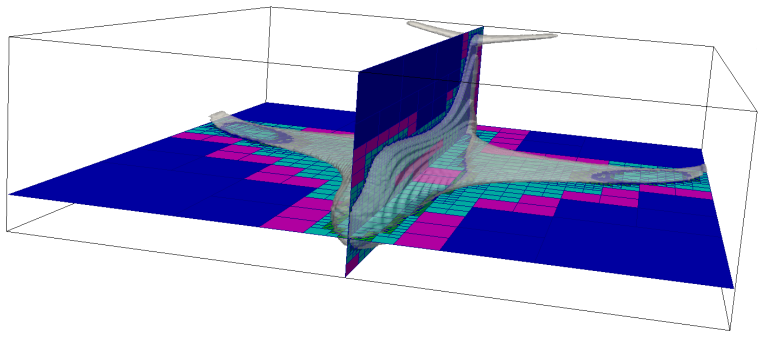
```
srun -n64 marbl -eul sedov.lua -
restart marbl/marbl_1939_64.root
```

```
lua> simulation.kinetic_energy
0.026980518709106
```

Figure 3.5: Checkpoints and restart on the Sedov problem in `MARBL -eul`.



(a) Sampling the In/Out field at quadrature points within an element



(b) A `quest::InOutOctree` built over a triangle mesh

Figure 3.6: (a) We initialize our material volume fractions over each element by checking if its quadrature points (blue 'x's) in physical space lie inside or outside the specified interface (blue curve) and projecting onto its high order degrees of freedom (red circles). (b) We accelerate our In/Out queries over a triangulated mesh using a `quest::InOutOctree`.

within the interior of the interface (dark blue 'x's) get a value of 1, while those outside the interface (light blue 'x's) get a value of 0. We use L2 projection to compute the value of the element's degrees of freedom (DOFs) (red circles) from the indicator values.

3.3 Exo

3.3.1 Exosim

The Exo library is shared infrastructure for all MAPP codes and includes the `Simulation` and `Package` object definitions, the Package Registry, and the shared `main`.

The `Simulation`

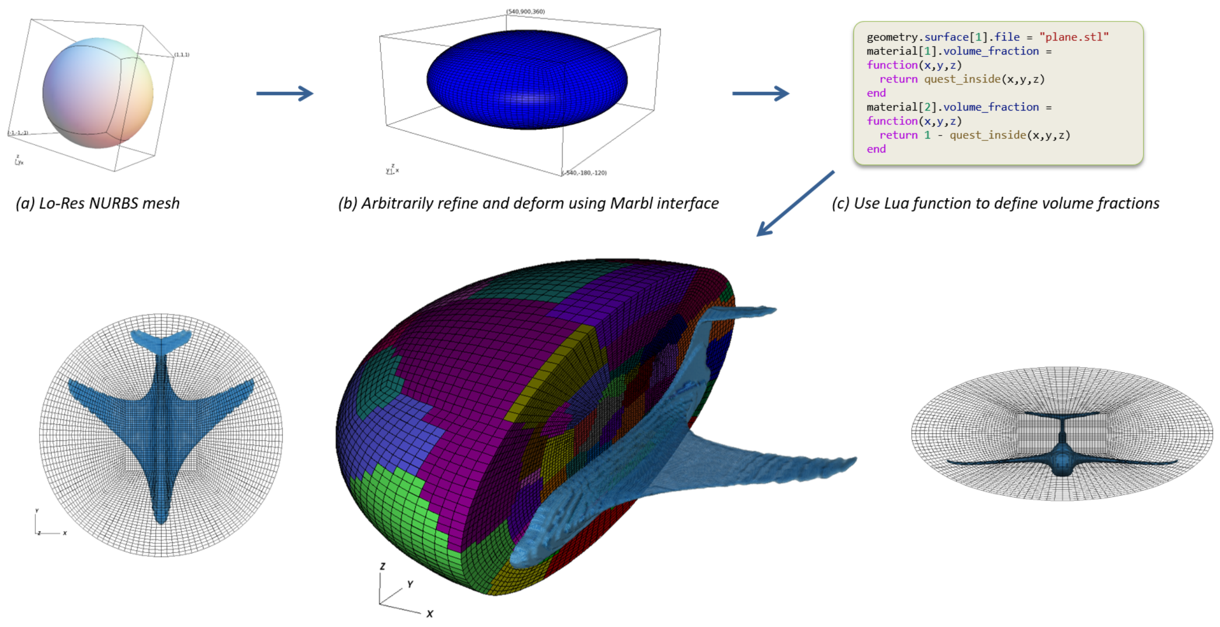
The `Simulation` object encapsulates all the state associated with a given code run and handles control-flow, including among other responsibilities, initializing a new run or restarting, event evaluation, handling stopping criterion, and `Package` tracking and cleanup.

The `Package`

The `Package` defines the abstract set of methods each derived `Package` can choose to implement, including checkpointing, restarting, time-stepping and Lua interface setup, among others. The `Package` also handles creation of Sidre groups and the Lua UI table. Almost all the main components of MARBL are instances of `Packages`, including the MARBL-lag and MARBL-eul hydro packages (Blast and Miranda), and interfaces to Ascent, Carter, and the Tracer library.

The Package Registry

The separation of codes under MAPP into multiple repositories exposes a more flexible permission model and a better encapsulation of code components but increases dependency complexity. Exo, the central driver, needs to know about each package, while


(1,1,1)
(-1,-1,-1)
(a) Lo-Res NURBS mesh
(540,900,360)
(-540,-180,-120)
(b) Arbitrarily refine and deform using Marbl interface
(c) Use Lua function to define volume fractions
(d) Shaped in volume fractions over domain decomposed quadratic mesh

```
geometry.surface[1].file = "plane.stl"
material[1].volume_fraction =
function(x,y,z)
  return quest_inside(x,y,z)
end
material[2].volume_fraction =
function(x,y,z)
  return 1 - quest_inside(x,y,z)
end
```

Figure 3.7: Workflow for using quest to "shape" a triangulated airplane model (`plane.stl`) into a high order MARBL simulation. Starting with an input computational mesh, in this case, a 7-element MFEM NURBS mesh over a spherical domain (a), we use MARBL's Lua interface (not shown), to refine and deform the mesh onto an ellipsoidal domain with 229,376 quadratic (Q2) elements distributed over 128 subdomains (b). (c) We also use the MARBL Lua interface to specify the per-material volume fractions using `quest`. (d) Three views of the resultant shaped-in volume fractions over our computational mesh. The elements in the middle figure are colored by subdomain. Each element has 27 quadrature points

Blast, for example, needs to know about some parts of Exo, creating a circular dependency. Additionally, since Miranda can be built by itself, any knowledge Exo has of Blast needs to be conditionally toggleable so that Blast is not required to build Miranda; explicit inclusion of Blast in Exo would necessitate additional build components, adding complexity.

To realize the benefits of multiple repositories while reducing complexity a Package Registry was introduced. The Package Registry lets classes derived from `Package` register themselves with Exo at runtime.

For example, to register a normal package:

```
class TracerPackage : public exo::Package {...};
...
EXO_REGISTER_PACKAGE("tracer", TracerPackage);
```

And for ”uber” packages, that can be toggled from the commandline:

```
class BlastPackage : public exo::Package {...};
...
EXO_REGISTER_UBER_PACKAGE("blast", "-lag", BlastPackage);
```

With the Package Registry, Exo has no explicit knowledge of Packages that exist outside of the Exo repository; whichever packages are linked in (or dynamically loaded) will be available to the user after the Packages register themselves, which happens before the application main program is entered.

Shared *main.cpp*

When the MARBL project started, each code in MAPP (MARBL, Miranda, and Blast) had their own entry point (`main`), which was a maintenance burden; each time a new feature was added, or bug was fixed, the changes had to be applied in 3 places. Not only did multiple source files need to be updated but since Miranda's `main` was written in Fortran a source code translation was also involved. After the introduction of the `Simulation`, `Package`, and Package Registry a single shared `main` was developed for Exo. This shared `main` is now used for all codes under MAPP, simplifying maintenance and providing a uniform experience for each code.

3.3.2 Physics Package Abstraction in Blast

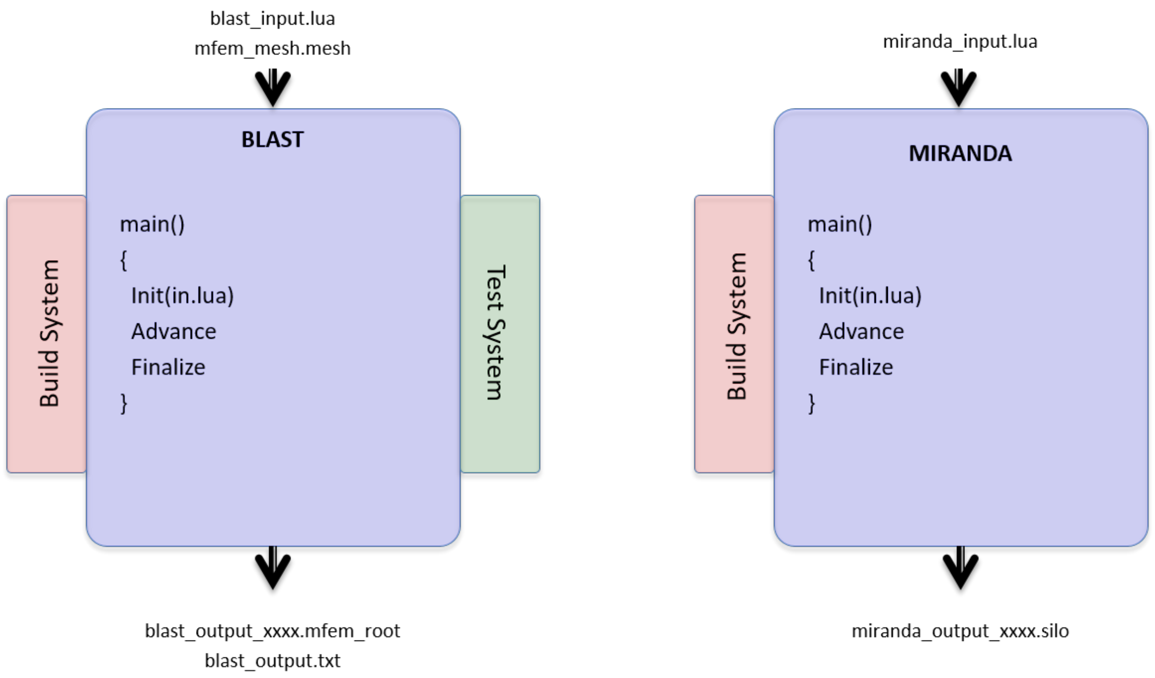
Beyond the abstractions for the `Simulation` and `Package`, MARBL-lag (Blast) uses its own set of abstractions to make adding and handling new physics simpler. These are broken down into a Problem, Method, State, and Processing Unit.

Problem

The Problem object is responsible for parsing the Well Known Tables (WKTs) that are used for setting up **what** MARBL is solving based on the user's input file. For example, the geometry, initial conditions, materials, boundary conditions, and physics options would be parsed in the Problem. Attributes of the Problem are derived from the IREP structure but do not modify them. Besides the base `HydroProblem` each additional physics may also have a Problem, which helps to keep the base object uncrowded and avoids unnecessary setup.

Method

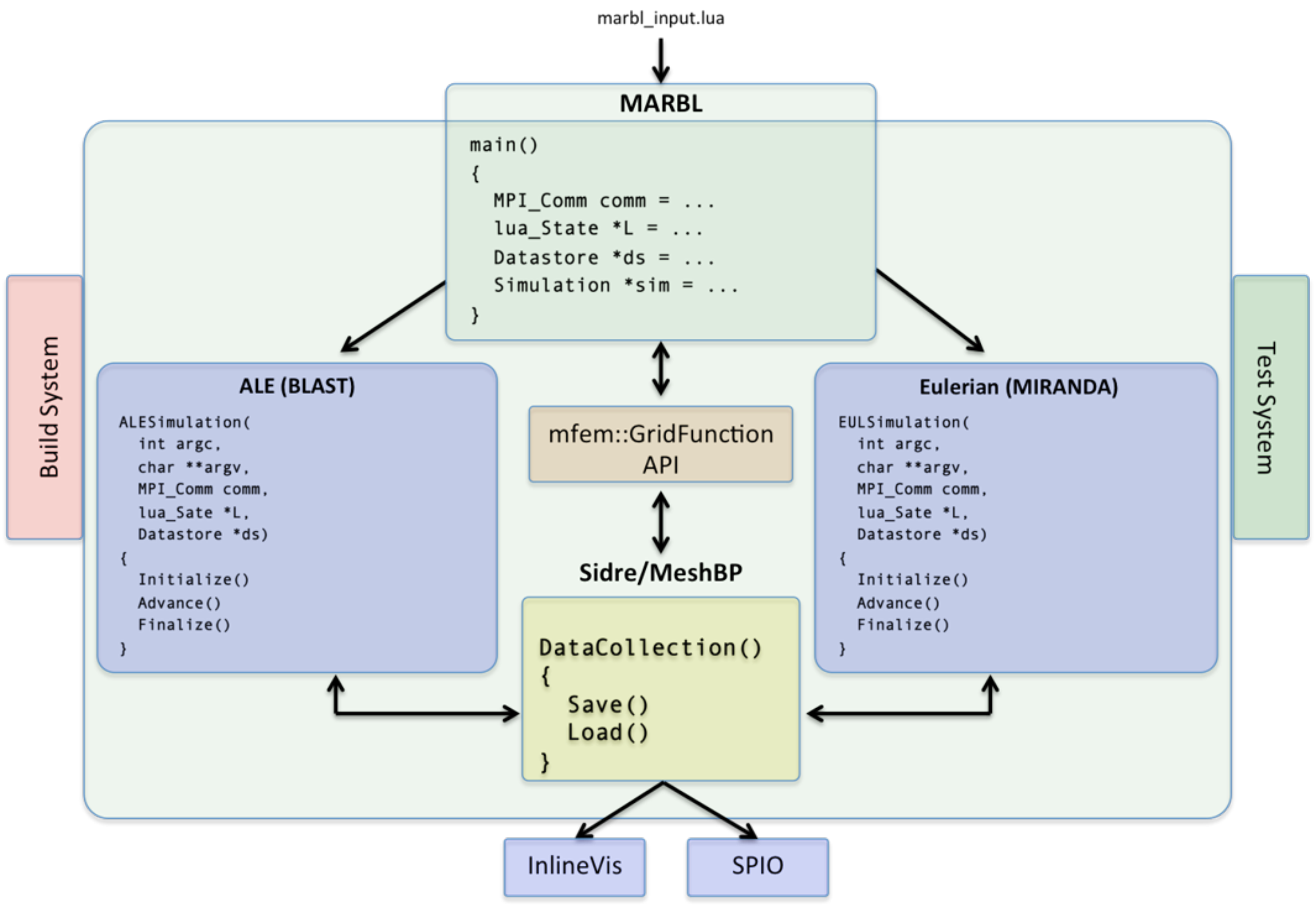
The Method object, like the Problem object, is responsible for parsing WKTs but this time for attributes on **how** MARBL should solve the problem in the user's input file. Input like the numerical method options and discretization parameters are handled by



```
main()
{
  Init(in.lua)
  Advance
  Finalize
}
```

```
main()
{
  Init(in.lua)
  Advance
  Finalize
}
```

(a) Separate codes in Summer 2015



```
main()
{
  MPI_Comm comm = ...
  lua_State *L = ...
  Datastore *ds = ...
  Simulation *sim = ...
}
```

```
ALESimulation(
  int argc,
  char **argv,
  MPI_Comm comm,
  lua_Sate *L,
  Datastore *ds)
{
  Initialize()
  Advance()
  Finalize()
}
```

```
EULSimulation(
  int argc,
  char **argv,
  MPI_Comm comm,
  lua_Sate *L,
  Datastore *ds)
{
  Initialize()
  Advance()
  Finalize()
}
```

```
DataCollection()
{
  Save()
  Load()
}
```

(b) Combined MARBL with abstract Main

Figure 3.8: Illustration of separate Blast and Miranda at start of project (a), and combined MARBL using an abstract main (b). *Note: This is an outdated figure from the FY16 L2. If we want to keep it, we need to update the class and function names.*

the Method. Similar to the Problem object, individual physics can optionally have their own Method object beyond the shared base one.

State

A State object is implemented for each physics and is used to hold the current (you guessed it) state. Some physics also include a Init(ial)State object to help separate the state that is updated from that which is static after initialization.

Processing Unit (PU)

Each physics has its own PU, which it used to perform computation on its associated State.

MasterPU The MasterPU holds a handle to each PU and provides methods for operating on the ordered list of PUs and for getting the ordered list of PUs. Similar to the Package Registry in Exo, the MasterPU includes functionality for registering PUs; the `REGISTER_PROCESSING_UNIT(name, type)` macro. Since developers can get the list of PUs without having to know each individual type, regions of the code that operate on the PUs can be written in an abstract way, which reduces the maintenance required when new physics are added.

Time Integrator Abstraction

The MasterPU breaks down PUs further by including a list of Implicit physics. This list enables the use of a multirate time integrator, for details see the IMEX section (2.7.5).

3.4 HO HPC

Several paths to providing features for an HO Code; each has benefits and risks

0D Works directly in HO via quadrature points instead of cell centers. Examples: EOS, Material properties

HO to LOR Convert from HO to low order (linear) approximation and run via existing tools / libraries. Works when we only need outputs / post-processing. Examples: Visualization, Transfer from Blast to Miranda

HO to LOR to HO Run using linear library, but we need to incorporate results back into simulation. Examples: Contact, Transfer from Miranda to Blast

Native HO Develop native high order algorithms and libraries. This option has the highest risk, but can have a lot of benefits. Examples: Visualization (Devil Ray), Tracer particles

3.4.1 HO/LOR data interoperability

Algorithms and infrastructure dedicated to transforming data between high-order and low-order meshes are central pieces to enabling mixed-order physics pipelines in vast multi-physics software ecosystems like MAPP. Broadly speaking, there are three types of algorithms that buttress these pipelines: mesh order transformations (i.e. transforms between high-order meshes and low-order approximations), mesh structural transformations (i.e. transforms between unstructured quad/hex meshes and structured rectilinear meshes), and data formatting transformations (e.g. bitwidths transforms, contiguous/interleaved transforms, etc.). Mesh order and structural transformations enable abstract compatibility between different physics packages and data formatting transforms sort out the practical concerns associated with passing these abstractly compatible meshes through real software interfaces.

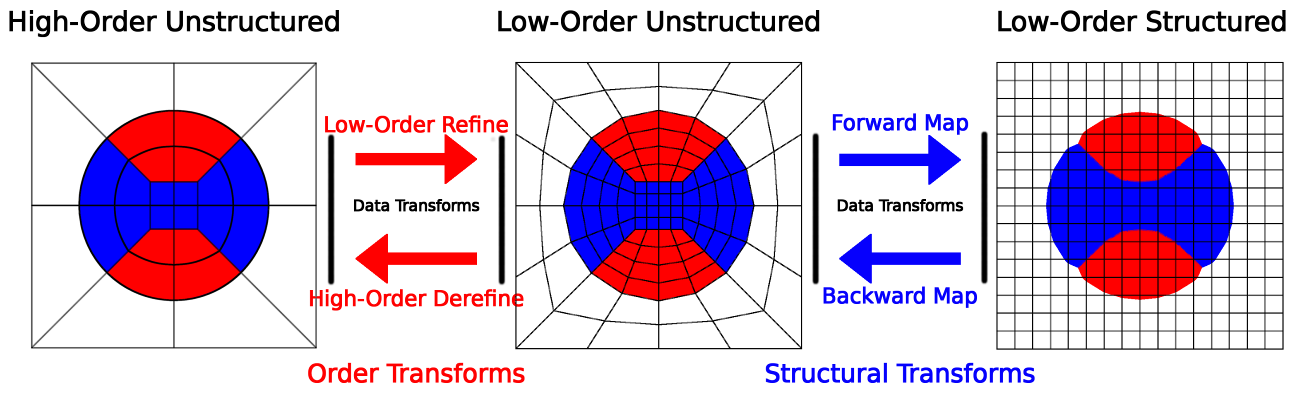


Figure 3.9: An abstract view of the data pipeline underpinning MAPP code communication.

Order Transforms

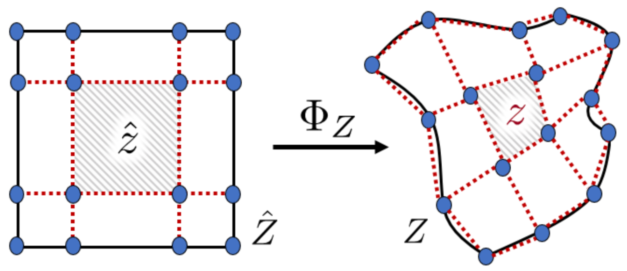
By their very definition, high-order elements contain a level of information that cannot be exactly capture by any finite number of low-order elements, so any sort of transformation between these two representations will necessarily involve some form of approximation. Low-order refinement $R : (m_{HO}, r_f, r_b) \rightarrow m_{LOR}$ is one such approximation technique that intakes a high-order mesh m_{HO}, a refinement factor r_f, and a refinement basis r_b, and produces a low-order approximation mesh m_{LOR} with r_f per-dimension low-order elements per high-order source element derived using basis r_b. Geometrically speaking, this operation effectively generates elements by connecting high-order basis sample points with straight lines and quantities by local interpolation in element centers (see Figure 3.10). More precisely, the low-order refinement operator R is an L^2 projection where:

$$(Rm_{HO}, m_{LOR}) = (m_{HO}, m_{LOR}) \,|\, m_{HO} \in HO,\ \forall m_{LOR} \in LOR \tag{3.1}$$

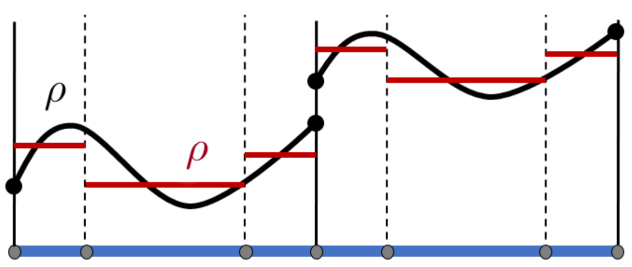
An important consideration in formulating the low-order refinement operator is in how it's approximations can impact conservative physics quantities (e.g. mass). Fortunately, this can be readily accounted for by modifying the low-order approximation equation for such quantities to average over volume instead of over samples. Put in mathematical terms, given a physical field $\rho : \mathbb{R}^D \rightarrow \mathbb{R}$ and a physical region C_z (corresponding to logical region $C_{\hat{z}}$) with volume V_z and Jacobian J_z, the approximation field $\rho' : \mathbb{R}^D \rightarrow \mathbb{R}$ can be made conservative by defining it as:

$$\rho'(C_z) = \frac{1}{V_z} \int_{\hat{z}} \hat{\rho} |J_z| \tag{3.2}$$

In its most general form, the inverse of the low-order refinement process is a difficult procedure to formulate. Given an arbitrary low-order mesh, derefinement factor, and high-order basis, it isn't obvious how to construct a high-order mesh with its associated quantities to best represent the data from this coarser form (consider: the high-order mesh can interpret low-order vertices as hard constraints or as targets for error minimization when defining their associated high-order functions). Fortunately, the context of this algorithm within a data pipeline where data is being transferred from a well-defined low-order refined mesh back onto its source high-order mesh greatly reduces the complexity of this problem. With this assumption in mind, we define high-order derefinement $P : m_{LOR} \rightarrow m_{HO}$ as the inverse process of low-order refinement R, which can be written mathematically as the left inverse of R, i.e.:



(a)



(b)

Figure 3.10: Low-order Refinement Visualizations. The mesh graphic presents how a basis (Gauss-Legendre) and a refinement factor (3) translate from logical space to physical space and how a low-order mesh is derived from this translation (red overlays). The graph presents how high-order quantities ρ over elements (black) are averaged to generate low-order approximations (red).

$$P = (R^T M_{LOR} R)^{-1} R^T M_{LOR} \tag{3.3}$$

It's important to note that the low-order space needs to be sufficiently large in order to prevent system degeneracy; in particular, R should be full column rank.

A nice property of the above formulation of P is that it guarantees aggregate quantity conservation. Intuitively, this property follows from the fact that P is the left-inverse of the quantity-conserving operator R (i.e. since $RP = 1$ and R conserves, P must also conserve). To put this in more concrete terms, we present a high-level proof of this claim:

Theorem 1. *Given a high-order mesh m_{HO} and it's low-order approximation m_{LOR} defined by $Rm_{HO} = m_{LOR}$, $Pm_{LOR} = m_{HO}$ and is quantity conservative.*

Proof. Consider arbitrary meshes $n_{HO} \in HO$ and $n_{LOR} \in LOR$. To prove our theorem, we must show that $(Rn_{HO}, 1) = (n_{HO}, 1)$ (quantity conservation of HO $\rightarrow$ LOR) and that $(Rn_{LOR}, 1) = (n_{LOR}, 1)$ (quantity conservation of LOR $\rightarrow$ HO). Observe: $(Pn_{LOR}, Rn_{HO}) = (RPn_{LOR}, Rn_{HO}) = (n_{LOR}, Rn_{HO}) = (n_{LOR}, n_{HO})$ based on the definitions of R and P. Since constants are represented in both the HO and LOR spaces, we have $R1_{HO} = 1_{LOR}$, so LOR $\rightarrow$ HO quantity conservation follows from setting $n_{HO} = 1_{HO}$ in the previous statement. □

Structural Transforms

Now that the high-order/low-order gap has been bridged, the next data reconciliation chasm that needs to be crossed lies in potential mesh structural incompatibilities. This set of incompatibilities encompasses any difference between interfaces in how a mesh's coordinates/topology are structured and/or presented. One of the most common instantiations of this incompatibility pattern in hydrodynamics software ecosystems is between unstructured quad/hex meshes (i.e. quad/hex meshes wherein each point and each element is explicitly defined) and structured rectilinear/uniform grid meshes (i.e. quad/hex meshes that follow an implicit grid for point/element definitions). Since they're the most relevant to enabling the various MAPP data pipelines, we direct our focus on this subset of structural transformations for the remainder of this section, providing an overview of how both directions of this transform are implemented within the code Carter (first presented in a more general form in [41]).

As its implementation informs its counterpart, we first outline the unstructured-to-structured transform, known in Carter terminology the as the *forward map* from the *donor* unstructured mesh to the *target* structured mesh. The first step in this process is to calculate intersection volumes between the arbitrary quad/hex elements of the *donor* and the axis-aligned elements of the *target*. Mathematically, we present this calculation as the function $V_x : d_i, t_j \rightarrow \mathbb{R}^+$, where d_i represents the ith *donor* zone, t_j represents the jth *target* zone, and the result is the intersection volume. To improve the efficiency of this calculation and to make it generalizable to polytopal elements, this intersection is framed as the intersection of each triangle in the triangular decomposition of

d_i (24 surface triangles for a hexahedron) with the rectangular surfaces of t_j (see Figure 3.11). In this frame, the computation is done by intersecting each d_i triangle with each of the six t_j planes to obtain intermediate in-*target* polygons and then calculating the resultant combined polyhedron to find the intersectional volume and centroid (using the standard formula for the volume/centroid of a solid). Volume-dependent zonal quantities tz_j are mapped by simply scaling their contributions by volume fraction and summing these contributions, i.e. $tz_j = \sum_i dz_i V_x(d_i, t_j)$.

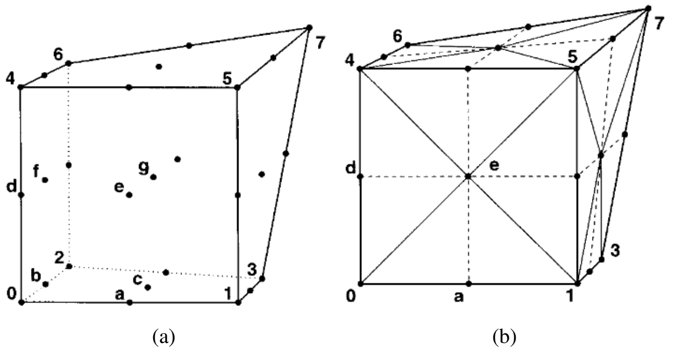


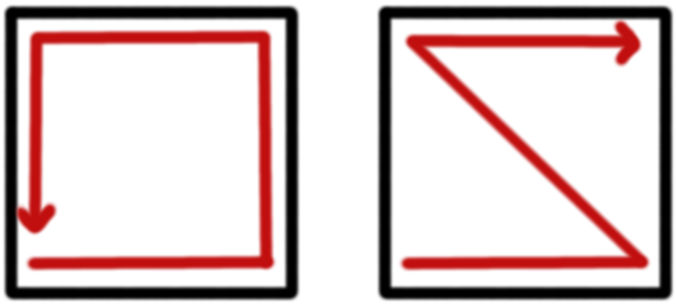
Figure 3.11: Donor Triangular Tesselation. These two images demonstrate how the unstructured-to-structured algorithm performs triangular tesselation on a hexahedral element, which is effectively just a mesh of the centroids of all related topological entities.

Now that we've covered the basics of the *forward map*, we can turn our attention to the mechanics of the *backward map*. As with the high-order/low-order transformations, the needs within the MAPP data pipeline allow the mesh to remain constant between conversions, which drastically reduces the scope of the problem and (consequently) its complexity. Since there are no changes to the geometry, we can use all of the $V_x(d_i, t_j)$ volumes calculated during the forward map to calculate how quantities will be projected from each *target* zone t_j (quantity tz_j) onto each *donor* zone d_i (quantity dz_i). For greater accuracy, second order interpolation enabled via the *target* gradient is used within the same base calculation as the *forward map*, i.e. $dz_i = \sum_j \nabla tz_j V_x(d_i, t_j)$.

Data Formatting Transforms

While the combination of order and structural transforms solve high-order/ low-order data interoperability issues in the abstract, they do little in the way of addressing the practical problems that arise when the realizations of these abstract procedures come into contact with physics codes and with one another. Though the physics software ecosystem tends to be fairly limited in its data concerns (i.e. most codes just use some form of mesh with a set of spanning quantities), there's still a great deal of freedom within this space, each dimension of which creates the potential for an incompatibility. The MAPP project adopted two solutions to combat these issues: the introduction of a library for annotating hierarchical data and transforming between the various realizations of these annotations (i.e. Conduit) and the development of shared hierarchical schemas for representing mesh data presented using these annotations (i.e. Blueprint) [42]. The project's uniform implementation of these solutions enabled fairly seamless internal cross-talk; however, external components needed to be adapted to interpret these new solutions. The remainder of this section will present the process for one such code, the structural transform code Carter, and demonstrate how Conduit, Blueprint, and their contained data formatting techniques were utilized in concert to create an adapted tool, Conduit-Carter (specifically, this tool created a Conduit/Blueprint interface for the Carter library's API).

The most rudimentary of the types of data formatting transforms leveraged in Conduit-Carter are Conduit's data layout transformations. Broadly speaking, this class of transformations spans the set of operations that manipulate data without changing its meaning or content (e.g. `float` to `double`, etc.). These transforms were employed mainly to solve two types of incompatibilities: contiguous/interleaved incompatibilities and topological ordering incompatibilities. In the case of the former, the incompatibility most often arises because one code chooses to orient its coordinate data to keep position data together (i.e. interleaved, e.g. $[x_1, y_1, z_1, ..., x_N, y_N, z_N]$) and the other chooses to keep dimensional data together (i.e. contiguous, e.g.



(a) MARBL Ordering (b) Overlink Ordering

Figure 3.12: Element Orientation Visualizations. The above diagrams represent two quad orderings used by two different codes in the MAPP data pipeline. To enable compatibility, per-element nodes need to be ordered between codes (in this case, by swapping the third and fourth vertices).

$[x_1,...,x_N,y_1,...y_N,z_1,...,z_N]$); both of these transforms are required for completely general compatibility, and the former is presented in Listing 3.2 (the algorithm for the latter is similar). In the latter case of topological ordering differences, simple per-element index re-arrangements are required; an illustration of the transformation that was required in the MAPP project pipeline can be found in Figure 3.12.

Listing 3.2: Conduit transforms from interleaved layout to contiguous layout

```
bool to_contiguous(const conduit::Node &src, conduit::Node &dest)
{
    Schema s_dest;
    NodeConstIterator itr = src.children();

    index_t curr_offset = 0;

    while(itr.has_next())
    {
        const Node &chld = itr.next();
        std::string name = itr.name();

        DataType curr_dt = chld.dtype();
        index_t elem_bytes = chld.dtype().element_bytes();
        curr_dt.set_stride(elem_bytes);
        curr_dt.set_offset(curr_offset);

        s_dest[name] = curr_dt;
        curr_offset += elem_bytes * curr_dt.number_of_elements();
    }

    dest.set(s_dest);
    dest.update(src);

    return true;
}
```

Another form of transformation implemented in the pipeline is material representation transforms. This set of transformations is necessarily more complex than the aforementioned set of formatting transforms as it encompasses methods that can fundamentally add, remove, or edit data in order to compress this data or to manipulate the relationship(s) presented therein. The former is more relevant in the case of the MAPP data pipeline as all the constituents happen to use the same relational model for materials, but have different strategies for presenting this data. More specifically, the MAPP hydrodynamics codes present materials as a list of uncompressed material-to-zone arrays (i.e. each material has an array specifying its fractional volume for each element in the mesh) and the structural overlay tool uses a custom compressed model that uses parallel indirection arrays to store mixed material cells separately [43]; Figure 3.13 presents a visual aid for comparing these two models. At a high level, the procedure for converting between this models is a matter of generating a zone-to-material mapping $M : z \rightarrow \vec{m}$ that associates each zone to its constituent materials (and their fractional volumes) and generating the custom layout zone-by-zone from this mapping.

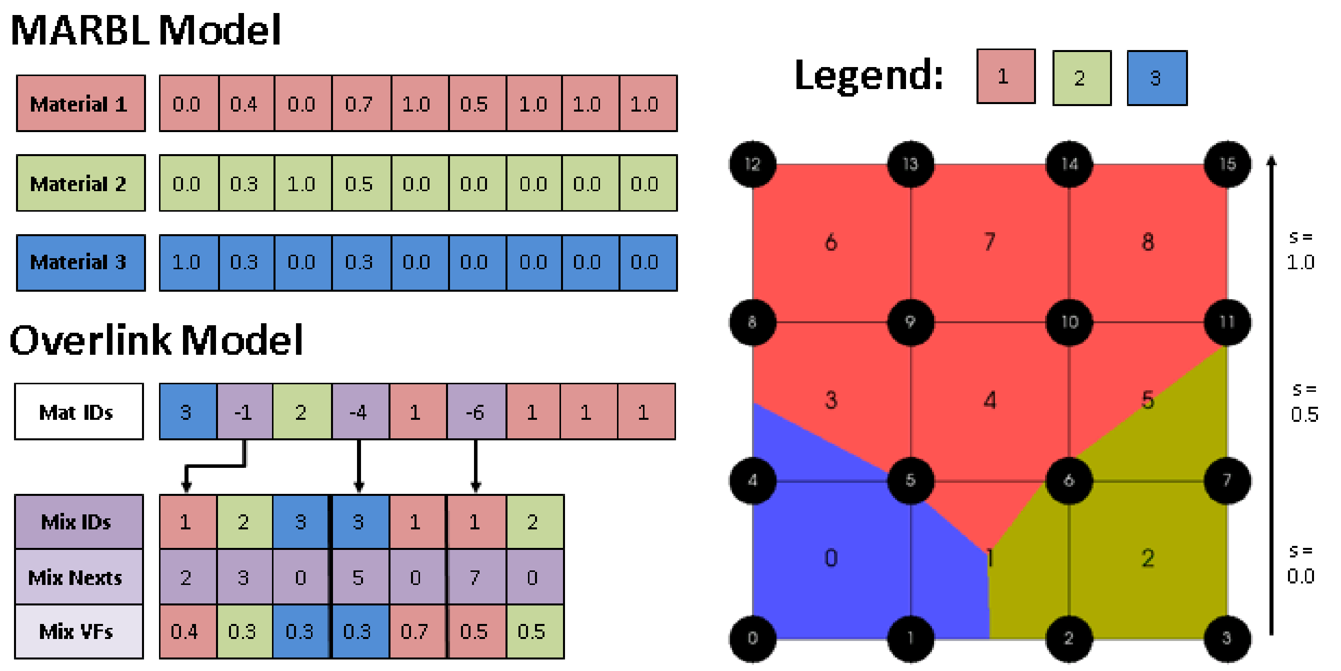


Figure 3.13: A visual comparison of the MARBL material model against the Overlink material model.

The third and final form of layout transformation employed by the MAPP pipeline is a slight twist on the second: the transformation of material-spanning quantities, specifically material species. This collection of transforms has similar nuance to the previous set, though adds an additional thin layer of complexity in the form of an extra layer of abstraction. This is unsurprising considering that MAPP and Overlink similarly diverge on their species models; fortunately, these models for both codes are logical extensions of their material models and thus the species transforms have a very similar flavor to their material counterparts. Figure 3.14 presents a visual for the MAPP hydrodynamics species model and Figure 3.15 presents a visual for Overlink's model.

3.4.2 Examples/Case studies

(The primary author of this subsection is ?)

- Contact
- TRT
- Visualization

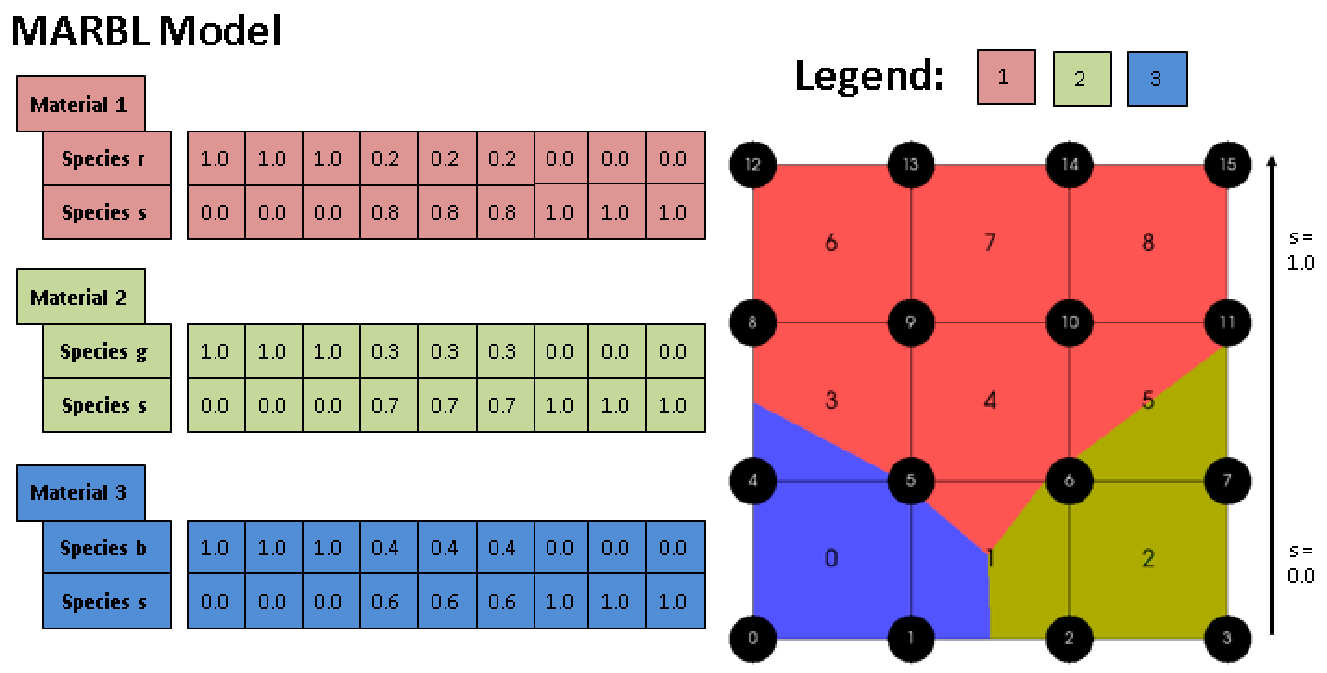


Figure 3.14: A visualization of the MARBL species model.

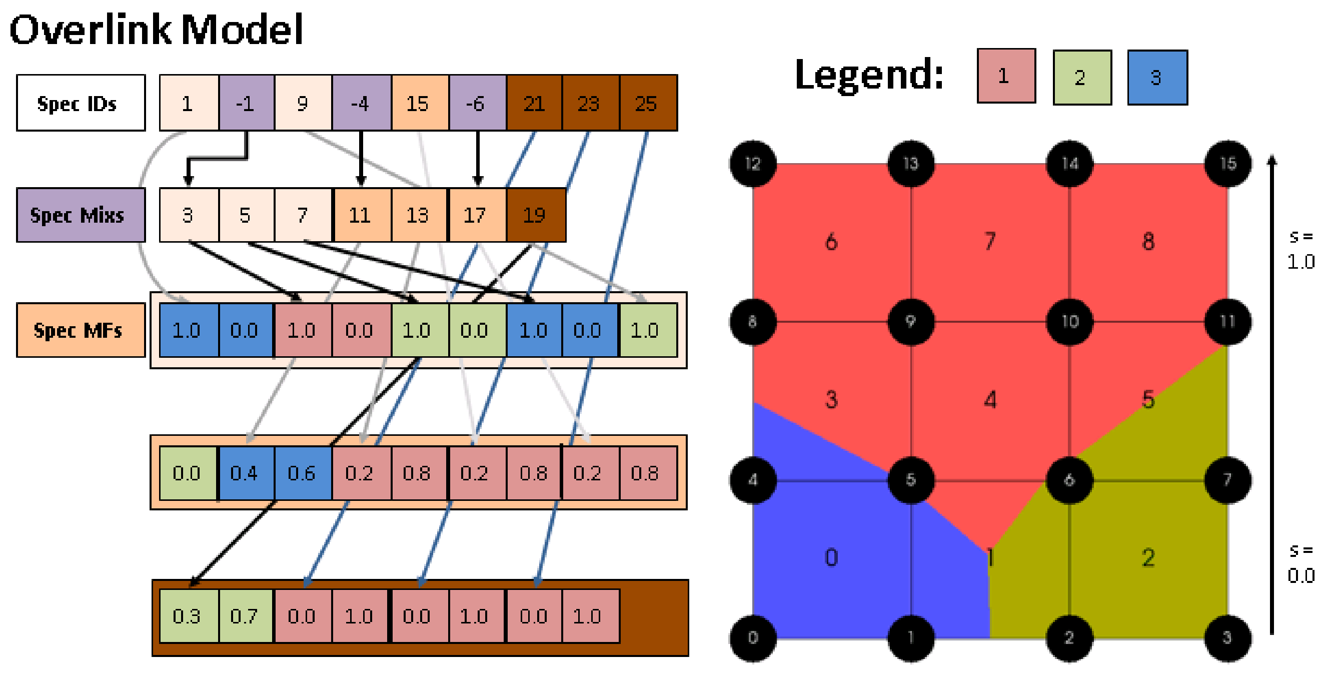


Figure 3.15: A visualization of the Overlink species model.

3.5 MARBL Build System

(The primary author of this section is L. Busby.)

This section describes the *MARBL Build System* (MBS). Specifically, this is the system that integrates compilation and linking of about 50 software packages into one of the several MAPP codes, including MARBL, Blast, and Miranda. [1]

3.5.1 History and Goals

The MBS began development in the fall of 2015. At that time, Blast and Miranda were separate codes, with separate development teams, separate third-party package systems, and each with its own GNU Make-based build system. The initial version of the MBS emerged in three steps over about 9–12 months:

1. The Miranda build system was redesigned and updated, borrowing some good techniques from the existing Blast system, along with some new ideas. One new idea was to adopt a file system layout such that `miranda` became a subdirectory in its own project. This allowed third-party packages to move up a level, thereby becoming available for use by other codes;
2. The Blast build system was updated, to incorporate the new layout and some other lessons as learned from the Miranda update;
3. A new directory for MARBL, similar to Blast and Miranda, was added. This step combined the three codes into one parent directory, adding a `bootstrap` script (and `mapp` repository containing it) to initialize a working directory from scratch. An `exo` repository in the parent directory contains the shared parts of the three codes, including most of the build system.

The system thus reached recognizable form in the autumn of 2016. The Build Monitor (BMR) was added in January, 2018. Improved support for builds on the LC Secure Computing Facility (SCF) required separation of all package download operations into the *bootstrap* scripts, added in June of 2018. Support for remote package farms was added in May of 2019. Most recently, in June, 2020, the Build Monitor was extended to add tracking of every individual file compilation for all packages.

The preeminent design goal for the MBS is simple: *Correct, minimal, fast, partial builds*. The edit-build-test cycle is the main activity for users of the MBS. It is critically important that the *build* part of that work should require minimal thought, start and finish quickly, and be entirely trustworthy. This goal is the main reason behind the overall architecture of the MBS: A single GNU Make process integrates the separate package builds and the final link of the *marbl* executable. This is possible and practical thanks to some new features that have been added to GNU Make over the past several years, as discussed below.

The layout of an MBS working directory is an important aspect of the build system design. Each MAPP code can access build system resources and third-party packages using identical path names. New packages and future codes can be added in a parallel fashion, without changing anything else in the layout. The shared parts of the system can be written without regard to which code uses them, so more of the build system code *is* shared: The overall size of the MBS, including the Build Monitor, is about 6000 lines of code (LOC), of which less than 300 are specific to one of Blast, Miranda, or MARBL.

3.5.2 Features of the MBS

After a brief review of GNU Make history, the remainder of this section discusses some features of the MBS that may be unusual or unique.

Stuart Feldman[44] began work on the first version of Unix Make in about 1976. There have been numerous reimplementations over the last 44 years.[45] The GNU version of Make [46] was written beginning in 1988, by Richard Stallman and Roland

[1] About 95% of the code in a MAPP executable belongs to others. Each package generally supplies its own build system, used by, but not otherwise part of, the MBS.

McGrath. The current maintainer of GNU Make is Paul Smith, who took over in 1997, with version 3.76. Over the next 10 years or so, beginning with version 3.78 in 1999, GNU Make added many new built-in functions and other capabilities that can, with appropriate usage, change what the program can do in fundamental ways. Specifically, the `eval` function[47] provides the foundation necessary to dynamically create *rules* (targets, their prerequisites, and recipes) inside a running `gmake` process.

In addition to `eval`, many other important new features were added, including text processing functions, user-defined functions, functions to query and interact with the filesystem, and job control for parallel processing. The new features stabilized about 13 years ago, at version 3.81, in April of 2006. In the last decade, GNU Make has become ubiquitous across nearly all Linux distributions, and version 3.81 or 3.82 tends to be the default. The MARBL Build System requires version 3.81 as a minimum, and it uses most of the available capabilities of that version. Development of GNU Make has continued since 2006, of course, to version 4.3 at present. The latest version has new features such as dynamically loadable modules, a builtin Guile interpreter, and others, but none of those features is currently required by the MBS.

Packages

To say that the MBS is composed of *packages* (and nothing else) would not be too far from the truth. The *package* abstraction is central. We use the term in several ways, and it often has the common sense meaning conveyed in an expression such as *the hdf5 package*. But it also has a formal definition inside the MBS makefiles, and here it is:

> An MBS package *p* is a collection of one or more GNU Make variables whose names are of the form *p.attribute*. One attribute in particular, the defining attribute for a package, is *status*.

That is, in the MBS, a GNU Make variable assignment `A.status=off` creates a package named *A*. For GNU Make, `A.status` is simply a global variable with a string value. (Variable names in GNU Make can include a broad set of characters [48].) For the human programmer, however, the variable is understood as the control switch for a package named *A*, a package that doesn't do anything at this moment, because it is turned off. This becomes interesting in the context of *computed variables* [49], when you also have introspection and string processing functions. For example, we can build a dynamic list of all the package names:

```
packagenames = $(subst .status,,$(filter %.status,$(.VARIABLES)))
```

GNU Make maintains a realtime list of all its current variables in *.VARIABLES*. The *filter* function then finds all those whose name ends with the suffix *.status*, and the *subst* function strips (substitutes an empty string for) that suffix. Note that *filter* and *subst* operate on whitespace-separated *lists* of of strings.

In addition to *status*, MBS packages share about a dozen other attributes. For example, *A.root* defines a pathname to the *root* directory for that package, which may vary depending on whether the MBS built the package locally, or found it in some external location.

The MBS can build most packages locally. Each such package defines its own so-called *DUCBI* attributes: *download*, *unpack*, *configure*, *build*, and *install*. The *download* attribute for a package defines the instructions for how to download the source code of the package, and so on. Usually, a simple shell command suffices, but it could be an arbitrary set of operations or program. For example, here is the shell code for how to unpack the *hdf5* package, expressed in MBS `package.attribute` form:

```
hdf5.unpack  = tar xfz $(hdf5.basepath)
```

The *basepath* is another standard attribute: It gives the location of the package source in your local working directory, after the *download* operation has completed. (This differs from the *root*, which gives the location of the package resources after installation.)

DUCBI is important to the MBS because it formalizes the operations of building packages. To say that another way, it attaches a specific, tunable, set of instructions to those five operations for every package, and makes them available by name, and makes those names systematically available to automated, introspective mechanisms within GNU Make. Thus, we can, and the MBS does, dynamically construct all the makefile rules to manage every package from download through installation. Another package attribute of particular interest is *prereq*:

```
mfem.prereq = metis netcdf hypre sidre lapack tls raja
```

That assignment means that the *mfem* package has seven prerequisite packages, as named. Before the MBS can build the *mfem* package, it first has to build and install *metis*, *netcdf*, ... Those packages may have their own prerequisites, of course, and so on. Since this information is expressible in a GNU Make variable, it is easy to include in the package construction rules. This dynamically defines the inter-package dependency graph for any set of currently downloaded local packages, to enable correct (parallel) package construction.

There are several other uses for package prerequisite information. One is to help construct linker commands. If a package *A* depends on a package *B*, and they build libraries *libA.a* and *libB.a*, respectively, we typically want to present those libraries to the linker in that order: *-lA -lB*. (This because *libB* defines a function referenced from *libA*, etc.) Taken over the set of all packages, you can satisfy this requirement by carrying out a topological sort of the the *.prereq* attributes for all packages. The MBS does this, using a gmake function, and thus automates library ordering in the final link commands.

You can also reverse this sort for any given package, to produce the list of packages that depend upon it. This facility can be used to automate the construction and maintenance of the MBS *Incremental Package Farm* (IPF).

One more use of prerequisites is to interactively construct and visualize the inter-package dependency graph, via a Graphviz [50] display (Figure 3.16).

The Build Monitor

The MBS *build monitor* (bmr) was created partly as an exercise to gain experience with Lua[51] programming, partly as a proof of principle, to demonstrate the utility of formalizing package construction via the DUCBI procedures discussed above, partly to help better understand parallel builds in the MBS, and partly just to experiment with a graphical interface for build system output. In operation, BMR provides a wrapper script *bmake*, which is substituted for *make* or *gmake*.

Running a build under *bmake* opens a Gnuplot[52] output display (Figure 3.17), which is updated once per second for the duration of the build. Each bar in the graph represents one package. They grow to the right, with configuration, build, and install phases tracked through color changes. (The unpack phase is normally too short to display.) At the end of a package build, the display shows the CPU time used to build that package, and a parallelization factor. For example, in the figure the *hypre* package required 340.49 cpu seconds to build, and this was done in about 148 seconds of wall clock time ($2.3 = 340.49 \div 148$) In the figure, four other packages – samrai, leos, axom, and overlink – are actively configuring or building at the time of the snapshot: $T = 192$.

The build monitor is effective at showing the interleaving of a parallel build, and at showing how inter-package dependencies control scheduling. We have recently added a compiler-wrapping facility to BMR that tracks individual compiler operations for nearly every package and file. This allows file-level analysis both inside and across packages, to discover which files are quick to compile, and which are not.

Internally, the build monitor combines several components, in several languages. Inside the running Make process, bmr replaces the value of the `SHELL` macro with a program that adds a tag before and after each command it runs. These tags have a standard form. Because of the *DUCBI* template, the output of all Make commands that unpack, configure, build, or install packages also have a standard form, one that can be easily recognized by Lua string pattern operators.

The *bmake* wrapper program is a classic *select(2)*-based event loop in C that also embeds a Lua interpreter. It watches the build output streams, and constructs in real time a Lua database of the event start and end tags that were inserted by the modified `SHELL`. Once per second, the program wakes up, converts changes in the database into Gnuplot commands, and sends them to Gnuplot, updating the display. The Gnuplot commands are also logged for later use in reviewing the build, producing movies, and so forth.

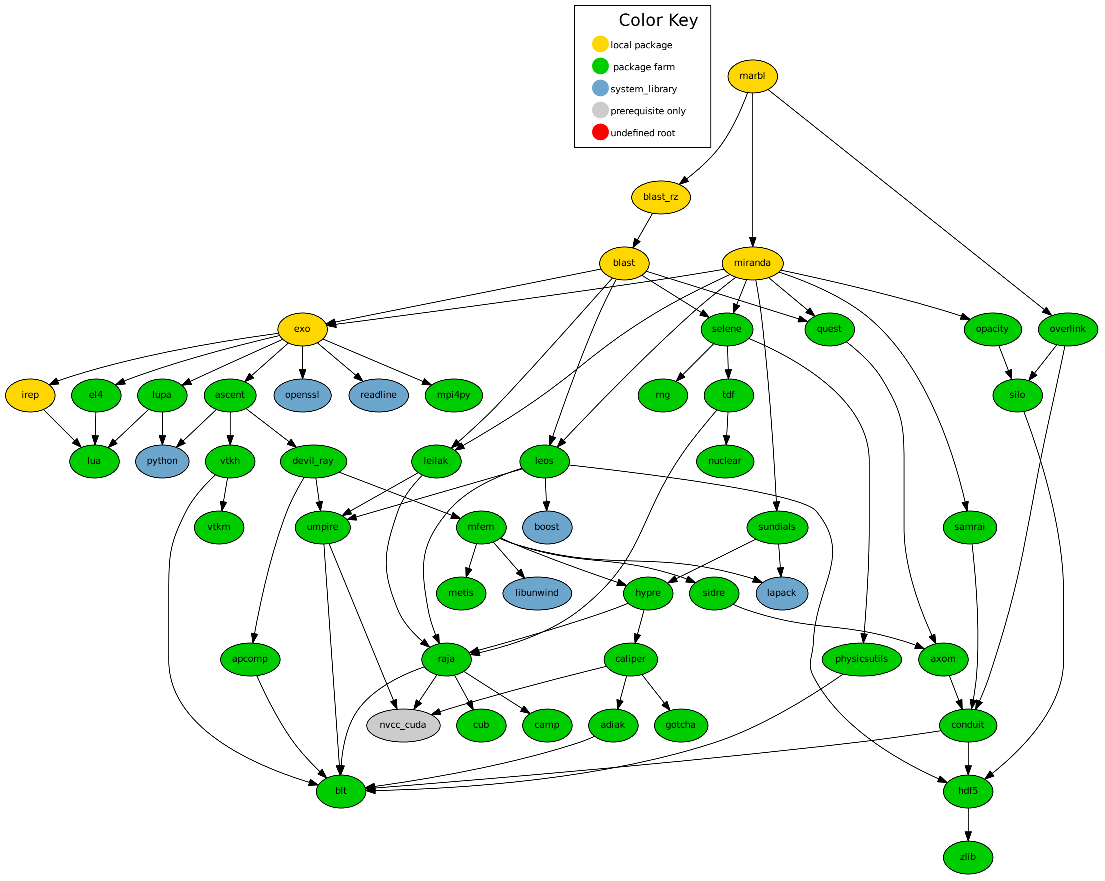


Figure 3.16: The MBS inter-package dependency graph (`make ginfo`)

Fortran Dependency Analysis and Compilation

`Modules`[53] add significant complexity to the task of compiling Fortran. The language definition allows a Fortran source code file `abc.f` to define an arbitrary number of modules, with arbitrary names $m1, m2, \ldots$. Fortran compilers produce an output file `abc.o`, containing the object code, along with a separate file corresponding to each module, typically `m1.mod`, `m2.mod`, etc. All the output files *depend on* the original source file in the usual sense: If the source file is changed, the build system must reconstruct all the output files. For GNU Make, this is an example of a *rule with multiple outputs*[54], a well-known and widely discussed problem.

In the general case of compiling Fortran using GNU Make, this problem can only be solved by using the *dummy target* approach discussed by Melski in the reference above. And the makefile rules for the several Fortran source files must explicitly contain the module names inside each source file. In a practical sense, this requires a separate set of makefile targets, one set for each source file and its modules, written by some other (external) program that analyzes each source code file.

However, if you slightly constrain the style of the Fortran source code, the problem can be solved using a GNU Make pattern rule (Melski's 'right way', ibid). Here is the constraint:

> Each Fortran source code file `abc.f` is expected to define exactly one module, and the name of the module must match the source file name in a regular way.

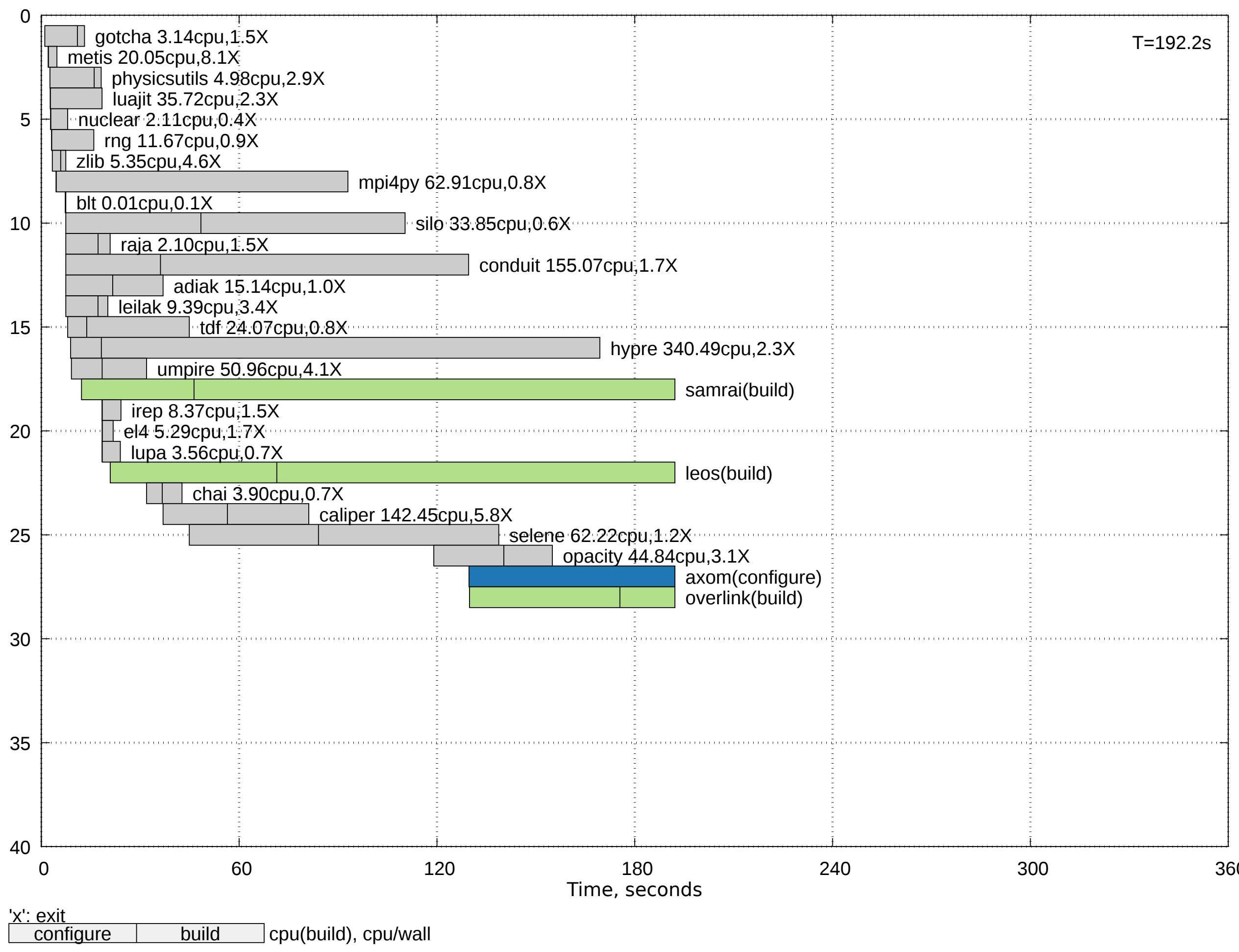


Figure 3.17: BMR snapshot at 3.2 minutes (192s) into a build

This constraint does not limit the expressiveness of Fortran, but it greatly simplifies the GNU Make rules that compile the code. In fact, the primary rule is now general and simple enough to show here in its essential form:

```
%.o les_%.mod : %.f
      $(COMPILE.f) $<
```

In words, we expect that compiling `abc.f` will produce `abc.o` and `les_abc.mod` as output. (Miranda applies a standard prefix “LES_” to all its module names, which the compiler converts to lower case in the file name.) That rule is supplemented by additional rules of the form

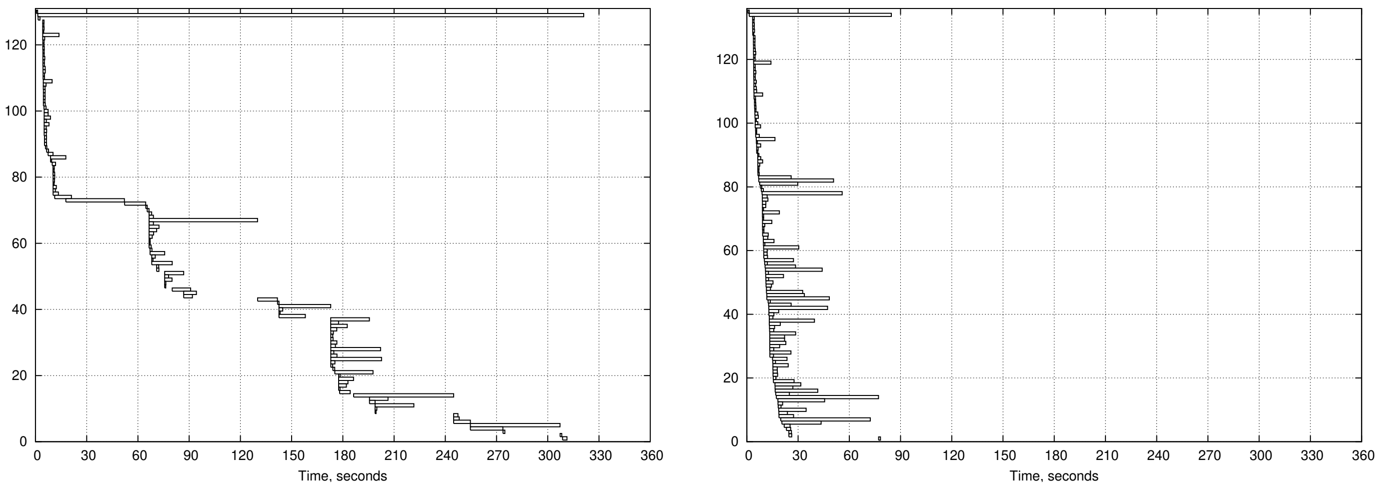
```
abc.o les_abc.mod : les_xyz.mod [...]
```

The additional rules supply inter-file dependency information. In the given case, we would find that `abc.f` contains a line of the form

```
USE LES_xyz
```

This effectively says that `abc` depends on `xyz`. If you change `xyz`, you have to recompile `abc`, and you have to compile `xyz` *before* `abc` in the first place. (Because compiling `abc` involves reading the module file `les_xyz.mod`, which didn't exist until you compiled `xyz.f`. Remember we started by saying that compiling Fortran with modules is complicated.) The additional rules are created by a 30-line shell script written specifically for the MARBL build system.

We could stop here, with a small, relatively efficient system for compiling Fortran code in the MBS. We can go farther, however. An early version of the build monitor produced these two pictures: (Figure 3.18).



(a) Onepass (b) Twopass

Figure 3.18: Miranda Twopass Compilation is 3.8× Faster than Onepass

These are taken from a longer paper [55] that discusses how to speed up Fortran compilation by using two passes over each source file. Each bar in the graphs represents one file; the horizontal axis is time in seconds. In the two pass method, the first pass (cheap) produces only the module (`.mod`) files. The second pass (expensive) produces the object (`.o`) files. With access to multiple cores, the second pass can be parallelized much more efficiently if all the module files are already present. This technique allowed us to compile Miranda about four times as fast as the usual one pass approach.

3.5.3 Assessment

Build systems tend to become Big Balls of Mud.[56] That outcome may be unavoidable, because utility is necessarily one of their main goals, and because they must be changeable, (even if not understandable), by developers whose primary interests are elsewhere. To try and put off that outcome for awhile, the MBS uses two or three strategies:

1. Simplicity, or failing that, brevity. The MBS is fairly small for what it does. Much conditional logic is condensed into its computed variables. For example:

   ```
   cmp_dir = $($(host_compiler).$(pltname).cmp_dir)
   ```

   sets a path name based on the current compiler and platform, replacing a two-level nested if-else construction that would otherwise have to be modified and extended for each new compiler or platform. Simplicity led us to the twopass Fortran method: It was easy enough to try, because the existing onepass method is short and simple, which is itself due to the one module-per-file Fortran style choice.[2]

[2] A. Cook gets all the credit for this; the MBS merely took advantage of his good taste.

2. Use of `$SYS_TYPE` for most platform discrimination purposes. More generally, the MBS hardwires (supplies pre-written configurations for) quite a bit of functionality from its host networks – the LLNL LC networks, extended slightly to the Sandia and perhaps LANL networks in the future. Giving up some carefully selected generality can save a lot of complexity.

3. The package machinery, as implemented, is both an architectural simplification (a repeated element with a standard form) and a small restoring force in the direction of order in the MBS. For the system to work at all, new names (*package*.`status`, etc.) have to be given in a form that meets the conventional syntax and semantics. *Ad hoc* variables won't work.

Unix Make is an old program, and many writers have considered its deficiencies at length: [57], [58], [59], [60]. One often-cited problem is that Make cannot easily scale up from small, one-directory projects to larger systems. This can be traced mainly to the static set of targets in a traditional Makefile, which leads to a static dependency graph. Prior Make-based build systems often try to address this problem via a metaprogrammed configuration step, that writes makefiles to match the varying build goals to a given platform. The choice *not* to adopt this approach was deliberate, and in retrospect, quite important in achieving not only the primary design goal, but the several other novel features as discussed earlier.

Using `eval` and other functions, GNU Make can generate a dynamic dependency graph, and the MBS demonstrates the effectiveness and flexibility of this solution. It has scaled from about 5 to about 50 packages so far, and it could probably scale to several hundred packages without much trouble.

The MBS imposes some constraints on source code file names. In addition to the Fortran module constraints mentioned earlier, it requires that source code file names for each of Blast, MARBL, or Miranda should be unique independent of subdirectories, and (for Fortran) lower case. This is done so that we can build a single library archive for each code without considering its internal subdirectories. These constraints do not limit the actual expressiveness of C++ or Fortran, but they might make it difficult to retarget the MBS to some other code with less stringent conventions.

Another serious problem for both Unix Make and GNU Make is debugging. One author has gone so far as to produce a version of GNU Make with enhanced tracing and debugging capabilities.[61] The MBS does not use `remake` (though it does look interesting.) The MBS incorporates a realtime verification script (`system_check`) that looks for some common errors such as trailing whitespace in variable definitions, and verifies the one-module-per-file rule in our Fortran source. This is a useful tool; it has found and prevented many actual blunders. However, it does not solve all problems that may arise. For the general case, nothing can replace a programmer who understands GNU Make, and who is knowledgeable in the design, conventions and idioms used in the MBS. Those skills are not difficult to gain, but they are not valued very highly. Thus, such people are rare.

3.5.4 Acknowledgements

The authors of the Blast build system as it existed in 2015 included at least M. Kumbera, V. Dobrev, T. Kolev, and R. Rieben. The prior Miranda build system was created by A. Cook, P. Amala, B. Cabot, and L. Chase. Subsequent work on the MBS is due to L. Busby, L. Taylor, T. Stitt, K. Weiss, B. Olson, and others on the MAPP development team. S. Dawson and M. Collette have maintained, curated, documented and published many of the third party packages also used by the MBS. We are grateful to all those individuals for their work, and to the users of the system for their thoughtful suggestions and comments.

3.5.5 Additional References

The GNU Make manual[46], mentioned earlier, continues to be an excellent reference. Another current book is [62], along with many articles by the same author: [63]. Perhaps the best information about some specific (newer) GNU Make features are several essays by P. Smith:[64]. Unfortunately, none of these provide extended examples that illustrate how GNU Make *computed variables* and functions can work together to effectively give the program new and powerful capabilities.

3.5.6 Some Statistics Regarding the MBS

The MBS is implemented using five main languages. The *build monitor* (BMR) and what we refer to as the *bootstrap* system are loosely coupled to the main build system, as illustrated in Table 3.1. At the time the MBS was first completed (fall of 2016), it

		Lines of code, per language and total LOC					
Component	Files	make	shell	C	python	lua	TOTAL
bootstrap	7	44	486	0	0	0	530
build monitor	17	40	190	387	0	319	936
compiler specs	9	431	0	0	0	0	431
platform specs	33	483	0	0	0	0	483
package defn	81	1851	0	0	0	0	1851
core	15	813	255	0	69	57	1194
TOTAL	162	3662	931	387	69	376	5425

Table 3.1: MBS Breakdown by component and language, August, 2020

contained an estimated 1500 LOC, with no build monitor and a minimal bootstrap system. Thus it has more than tripled in size over the subsequent three years. Most of the added code has been in compiler, platform, and package description files. There are 81 package definition files. Many of those are trivial in the sense that they define packages that are never built locally. There are currently 52 downloadable packages that can be built locally, with package description files containing a total of 1622 LOC. Therefore, the average file for a buildable package requires 31.2 LOC ($1622 \div 52$).

Of the 3662 LOC in GNU Make, about 300 lines are function definitions. Another 350 or so are used to define static targets or pattern rules. The remaining 3000 LOC mostly define variables. (The `make printvars` command shows about 1813 variables in the MBS, a slight underestimate.) Not shown here are statistics for most of the code designed to configure and run the MARBL Automated Testing System (ATS), and some other code analysis tools.

As of August 2020, a full parallel build of the Marbl code (42 local packages plus the *marbl* executable itself) on the LC toss_3 platforms requires about 1339 seconds of wall time (22.3 minutes), and about 17,173 seconds (4.77 hours) of cpu time, yielding a speedup of 12.8$\times$ ($17173 \div 1339$). After MARBL has been built once, and with the *noremake* option set for the blast, exo, irep, and miranda packages, the minimal rebuild time for MARBL is about 22 seconds wall time, of which 18 seconds is for the final link. A faster linker would be useful.

3.6 Software Quality Assurance (SQA)

3.6.1 MARBL Test System

At the heart of MARBL's software quality efforts is the MARBL Test System (MTS), which runs larger integrated tests (as opposed to smaller unit tests.) The MTS is built on LLNL's Automated Test System (ATS) and generates ATS files, which it runs through a vanilla ATS installation. ATS handles choosing and creating the scheduler commands and handling job submission. After that, control is handed back to the MTS for results parsing.

Test files

Tests in the MTS are defined as entries in a Python dictionary and are separated into task-specific python files based on their purpose; we currently have files for nightly, performance, and smoke tests. Each test file has an intended purpose but a test file can be used in any context the developer wants. We'll start by describing the main purpose of each file. Then, we describe details of other testing modes.

Nightly Tests The nightly test file describes the tests that developers regularly run to validate changes; the vast majority of our tests fall into this file. Tests are expected to run in less than 2 minutes and use only part of a compute node, which allows this suite to run in 5-10 minutes on 2 nodes even though there are about 200 tests that run. Each test generates a *.csv* file consisting of multiple time-history curves, which are compared to baselines; for a test to pass, the code must run successfully and match the baselines curves to a tolerance defined by the test developer.

Performance Tests The performance tests are larger and longer-running than the nightly tests. Caliper is used during the runs and SPOT data is output, allowing developers to use the SPOT web-service to inspect runtime and annotation information.

Smoke Tests The smoke tests are the shortest tests we define, running for a few seconds and on only a few processors. They are useful to quickly test that a non-default build configuration is able to run input files.

Testing Modes

In addition to the nightly, performance, and smoke tests described above, the MTS defines additional modes for code analysis: `asan`, `lsan`, `coverage`, `memcheck`, and `memset`.

asan The `asan` mode works alongside the address sanitizer [65] instrumented build to test for memory usage errors related to using a buffer after it has been released and for buffer overflows. The overhead of asan is small and we're able to run our nightly test suite with it.

lsan The `lsan` mode works with the leak sanitizer [66] instrumented build to test for memory leaks. The overhead of `lsan` is small and we're able to run our nightly suite with it.

coverage To understand how well our integration tests exercise MARBL we use clang's coverage feature [67]. Clang's coverage tool works more robustly than GCC's on multi-process applications and merges coverage files much faster.

memcheck In addition to the `lsan` mode we have a mode that uses Valgrind called `memcheck`. The `memcheck` mode was introduced before `lsan` and only runs on a subset of tests because of larger runtime overhead.

memset To help diagnose uninitialized heap memory usage, we added a test mode called `memset`. The `memset` mode captures calls to `malloc` (and by extension `new`, `calloc`, etc.) and sets the bytes of the allocation to `0xff` (or some other value if the user prefers); this way `floats` and `doubles` are initialized to `NaN`, which makes the usage of uninitialized buffers easier to find. In the future we want to use the Memory Sanitizer compiler instrumentation [68] but it isn't as easy to use as `asan` and `lsan` and we've been unsuccessful in getting it to work.

Interface to the MTS

The gmake interface for building was extended to control testing:

```
make {mode} [test_files=a/b/c/py,x/y/z.py]            \
            [test_list=]                              \
            [num_nodes=]                              \
            [test_dir=path/to/output]                 \
            [exclude="<python predicate expression>"] \
            [include="<python predicate expression>"] \
            [timeout=]
```

Filtering The `include` and `exclude` options are used for filtering tests. Any option in a test definition can be filtered, for example to only include Lagrangian (MARBL-lag) tests (`include="frame == 'lagrangian'"`) or only run GPU tests (`include="ngpu > 0"`). A special test attribute, `dependencies`, is compared to the build packages, the mode(s), the host and device compilers, and the system type making it easy to include tests that only run with certain platforms and build configurations.

3.6.2 Continuous Integration (CI) and Continuous Deployment (CD)

With the introduction of centralized version control and Continuous Integration and Deployment (CI/CD) tools through Livermore Computing's managed Atlassian Bitbucket and Bamboo tools, we are better able to quickly find bugs and verify that new code additions build and run as expected. Our current CI/CD plans are partitioned into regression testing, non-default build configuration, performance testing, and code analysis.

Regression Testing

Each night and with each change to the `develop` branch of our 1st party libraries, MARBL is built and run through the full nightly test suite 3.6.1 on Livermore Computing CTS and ATS platforms.

Updating the Development Installation The executable from the nightly regression test is saved and installed for users as `marbl-dev` automatically. The latest build artifacts from the nightly tests and the tests that run with commits to develop are made available in a package farm for cached builds (See 3.5.2 for more details).

(Non-Default) Build Tests

Each night, MARBL is built on a wider range of platform and compilers than those used for regression testing. Each build is run through the smoke test suite 3.6.1.

Performance Testing

Each night, MARBL is run through the performance test suite 3.6.1 and the SPOT results are saved to a well known location, making it easy to view and compare histories of multiple performance tests on multiple platforms.

Code Analysis

Each night, a few diagnostic configurations of MARBL are built including an `lsan`, `asan`, and `coverage` build along with a GPU debug build. These builds are run through the nightly test suite 3.6.1, but skip the regression testing aspect. These builds have not only found memory issues in our 1st party libraries but also in 3rd party libraries, which we're able to report to the library authors, improving the code quality for all consumers.

3.6.3 CI/CD Assessment

Regression Testing

After an initial period of introduction the MARBL team has gotten regression testing to a point where either no tests are failing or only one to a few fail intermittently. A new gmake target was added, `make bamboo`, which lets developers test their code changes with the Bamboo agent before committing, which has decreased the number of submissions that result in tolerance failures. `Make bamboo` additionally emails the submitter if tests fail and copies a set of updates for the failing baselines to the user.

Easing the burden of regression testing, especially when it must be done on multiple systems, is key to getting developer adoption.

One issue we've noticed is that our regression tests are not always enough. They are designed to run fast, since resources are generally limited and we don't want to hold up development, but we've have reproducibility issues pop up. Had we included longer regression tests initially we would have seen some of these reproducibility issues earlier and spent less time debugging. A solution to this would be weekly regression tests that run longer and with more resources.

Build Tests

Build tests have been helpful when non-standard language features are added to MARBL; the build tests include builds with compilers we don't normally use.

Because the build tests run on multiple machines, and the PASS/FAIL status is the aggregate result, we'll sometimes see failures when a specific machine is down for maintenance, which can hide new failures that come up.

Performance Testing

Performance testing creates Caliper files that can be used along with SPOT to view runtime histories for specific problems. SPOT was used heavily during the phase of development where we were rapidly adding changes to performance-critical code and it made it easy for developers to validate their changes. Developers could run the performance suite locally and grab results from the nightly dump to make sure they were not hurting performance.

Later in the development cycle we used SPOT less but from time-to-time it was used to attribute performance degradation to specific days, from which we could look at the merge history. It would have been better for developers to always make sure their changes didn't cause performance regressions but the performance tests are expensive to run and do not give you a yes/no answer, like that in the regression tests.

Adding a feature to ”fail” the performance tests if performance drops ”too much” would be beneficial.

Code Analysis

To recap, the code analysis tests include memory leak and illegal memory usage tests with Leak Sanitizer (lsan) and Address Sanitizer (asan), along with code coverage and debug GPU runs. Overall these tools have been extremely helpful and we plan to add more in the future, including Memory Sanitizer (msan) and cuda-memcheck or compute-sanitizer tests.

One issue we've seen is that there is always at least a few tests failing in the suite, resulting in a FAIL for the whole suite. These known errors are low priority bugs that can be time consuming to fix, so they may hang around for months at a time. Because we always see a FAIL, new failures can slip under the rug.

Missed code analysis failures have been discovered multiple times by users or developers who observe abnormal runtime behavior only to realize that the failure they're seeing is captured in an already-failing test.

Asan in particular has helped the MARBL team find bugs that would have been extremely difficult to investigate otherwise; it provides a backtrace with source file names and line numbers making it easy to find the region of error.

We are looking forward to having msan because we've suffered from multiple memory initialization bugs that have taken a non-trivial amount of time to fix. Unlike asan and lsan, msan build options need to be used in all packages, which is difficult because our TPLs include a variety of build systems that may or may not make it easy to enable msan or pass explicit compiler flags. Msan usage also requires that libcxx, libc, and libmpi be compiled with proper flags; most code projects take those as given and may not provide easy ways to use external versions, especially for libc and libcxx.

marbl-dev and the Package Farm

Our continuous deployment of a user-accessible `marbl-dev` installation and build artifacts for the default compiler options has been extremely helpful. Users get access to the latest version of the application, if they want, and developers get the option to do partial builds, which result in huge build time savings. MARBL's CD has also been almost exclusively hands off. Once it was setup it has just worked.

3.6.4 Code Formatting

The Clang project's `clang-format` [69] and `clang-tidy` [70] tools are used to enforce consistent code style (style configuration was chosen to make every developer equally unhappy.)

`clang-format`

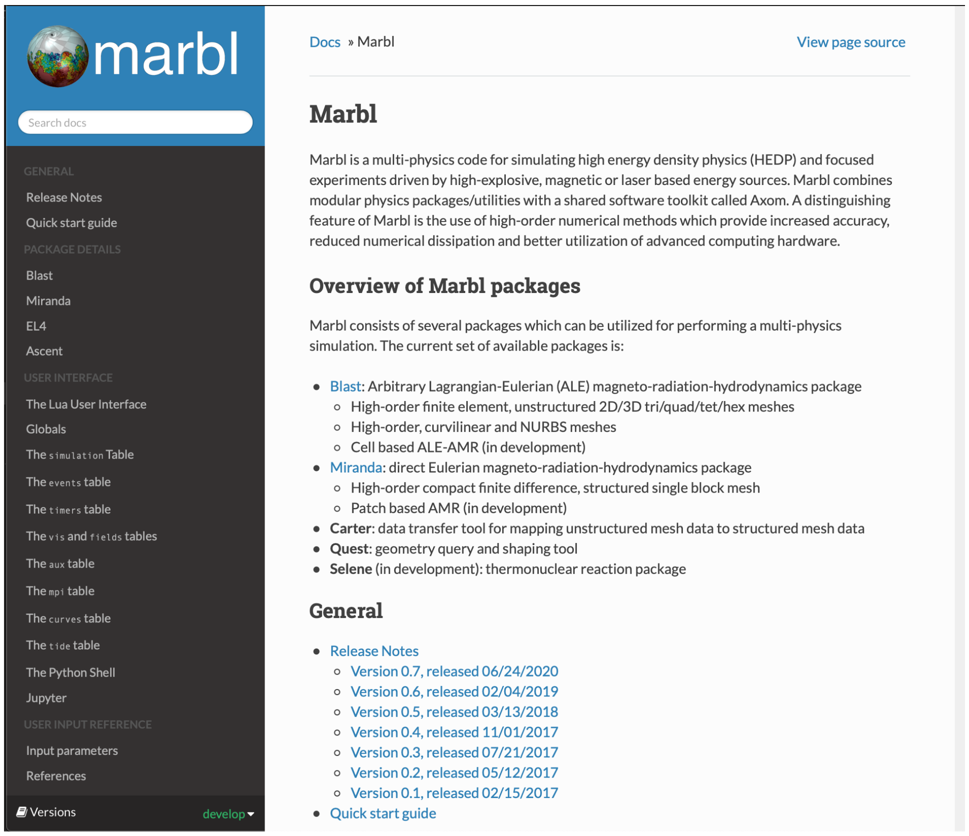
`clang-format` simply handles white-space and line breaks and operates on the source file, making it easy to apply. Another gmake target was added, `make format`, to make it easy for developers to run the formatter. With CI, we've also created a task that runs on new pull requests that verifies code changes are style compliant; if not, the PR page indicates there is an issue, which has made it easy to keep the develop branch formatting up-to-date.

`clang-tidy`

`clang-tidy` is a static-analysis and code modernization tool that operates on the code abstract syntax tree (AST), requiring a database of compilation commands for each source file being tidied. Since our default compilers are not clang and use vendor-specific compiler flags, we are unable to use `clang-tidy` in our default builds, which limits the times it is run. In addition, changes made by `clang-tidy` can break compilation so care must be taken when merging its changes. Nevertheless, `clang-tidy` has been helpful in highlighting various bad practices and modernizing our C++ usage; it has been exposed to run as `make tidy`.

3.6.5 Documentation

Code documentation is written in reStructuredText (rst) and Markdown (md) and built using Sphinx [71]. The documentation is published internally at *rzlc.llnl.gov/marbl* and includes the option to switch the documentation version much like, and inspired by, the web hosted readthedocs system. Figure 3.19 shows the documentation homepage.



Marbl

Marbl is a multi-physics code for simulating high energy density physics (HEDP) and focused experiments driven by high-explosive, magnetic or laser based energy sources. Marbl combines modular physics packages/utilities with a shared software toolkit called Axom. A distinguishing feature of Marbl is the use of high-order numerical methods which provide increased accuracy, reduced numerical dissipation and better utilization of advanced computing hardware.

Overview of Marbl packages

Marbl consists of several packages which can be utilized for performing a multi-physics simulation. The current set of available packages is:

- Blast: Arbitrary Lagrangian-Eulerian (ALE) magneto-radiation-hydrodynamics package
  - High-order finite element, unstructured 2D/3D tri/quad/tet/hex meshes
  - High-order, curvilinear and NURBS meshes
  - Cell based ALE-AMR (in development)
- Miranda: direct Eulerian magneto-radiation-hydrodynamics package
  - High-order compact finite difference, structured single block mesh
  - Patch based AMR (in development)
- **Carter**: data transfer tool for mapping unstructured mesh data to structured mesh data
- **Quest**: geometry query and shaping tool
- **Selene** (in development): thermonuclear reaction package

General

- Release Notes
  - Version 0.7, released 06/24/2020
  - Version 0.6, released 02/04/2019
  - Version 0.5, released 03/13/2018
  - Version 0.4, released 11/01/2017
  - Version 0.3, released 07/21/2017
  - Version 0.2, released 05/12/2017
  - Version 0.1, released 02/15/2017
- Quick start guide

Figure 3.19: Front page of MARBL's Sphinx-based web documentation

.

3.7 Performance Portability

Modern high performance computing platforms are diverse, with hardware designs ranging from homogeneous multicore CPUs to accelerated GPU systems. Achieving high performance on a given computer platform requires that application developers choose execution and memory management strategies that are well-suited to the given hardware architecture. Practically speaking, this means that computational algorithm structures and the programming model used to execute numerical kernels as well as data placement and access strategies must match hardware features and constraints. For large applications, used daily in production and which are under continual development, developing and maintaining architecture-specific versions is untenable. Maintainability requires single-source application code that is performance portable across a range of architectures and programming models.

For the MAPP application, we have chosen to adopt software abstractions, parallel programming models, and mathematical support libraries that have been developed to address large-scale multiphysics application portability challenges. These choices and how they are applied are described in this section.

3.7.1 RAJA: Portable Kernel Execution for C++

RAJA [72, 73] is an open source library of C++ abstractions, developed at LLNL, that enables developers to write portable (i.e., *single-source*) application codes. RAJA grew out of a need to support large scale production multiphysics applications in the LLNL ASC program on advanced technology supercomputing platforms, such as Sierra, as well as various other commodity systems, including laptops, workstations, and HPC clusters. RAJA also supports many other HPC applications at LLNL and elsewhere. It is part of the Exascale Computing Project (ECP) Software Technology ecosystem. RAJA is developed collaboratively by computer science researchers, application developers, and vendors.

Using RAJA, one writes numerical kernels that can be run on different platforms by applying different programming model back-ends. Typically, this is done by defining RAJA *execution policy types* in header files and then applying the desired policy types to application kernels and compiling the application for the associated computing platform. Since kernel execution is encapsulated in policy types located in header files, RAJA-enabled application source code does not require modification to run on different architectures.

There are four main encapsulation concepts in RAJA:

- Kernel launch C++ template method
- Execution policy type
- Kernel iteration space
- Kernel body C++ lambda expression

The launch template that executes a kernel is decoupled from the details of the kernel structure, the programming model used to execute the kernel, and how loop iterates are mapped to hardware resources. The way in which a kernel will execute and the computation it will perform are determined by a combination of C++ template specialization of the launch method and function arguments passed to it. Each RAJA launch method runs a kernel in a manner defined by the execution policy template type given to instantiate it. A kernel launch method is passed one or more loop iteration space objects, which define the size and iterates for each loop in a kernel. RAJA provides a basic set of iteration space objects, called *Segments*, that includes contiguous and strided index ranges, and “lists” of arbitrary indices. Segment types can be combined in arbitrary ways for a single kernel using the RAJA *IndexSet* abstraction. User-defined iteration spaces are also supported. Lastly, the computational operations performed in a kernel are defined in one or more C++ lambda expressions passed to the method.

RAJA provides kernel launch methods that support a virtually endless variety of kernel loop patterns from single C-style for-loops to complex nested loop structures. Applying RAJA in its simplest form yields architecture portability by enabling different programming model back-ends, such as OpenMP, CUDA, HIP, etc., by compiling an application with appropriate execution policies. RAJA execution policy constructs enable many fine-grained kernel optimizations, such as loop tiling, use of thread local and GPU shared memory, and detailed mapping of loop iterates to hardware resources, such as GPU blocks and threads in NVIDIA CUDA parlance, for example.

Recent RAJA development provides support for hierarchical parallelism and asynchronous execution whereby execution of a kernel can be overlapped with execution of other kernels (via CUDA streams, for instance) or overlapped with memory operations, such as host-device data copies. RAJA also supports fusion of many GPU kernels into a single CPU-initiated GPU launch operation. This has proven to be a performance benefit for MPI buffer packing and unpacking operations from/to GPU memory spaces in LLNL ASC production applications.

RAJA also provides portable support for other essential HPC application needs, such as reductions, scans, sorts, and atomic operations. The execution back-ends that RAJA supports include: sequential (multiple forms – useful for application debugging), OpenMP CPU multithreading and target offload, Intel Threading Building Blocks, CUDA (NVIDIA GPUs), HIP (AMD GPUs). A SYCL back-end to support Intel GPUs is under development via an ECP collaboration.

3.7.2 OpenMP and FEXL: Portable Kernel Execution for Fortran

The OpenMP pragma-based abstraction layer was chosen to provide performance and portability for MARBL's Fortran hydrodynamics code, Miranda. A solution like RAJA could not be implemented in Fortran due to the lack of Fortran language support for templates and lambda functions.

Recent updates to the OpenMP standard have included the "device offload" capability which enables the application programmer to specify regions of code to be executed on a selected device (GPUs, etc). These regions are specified through the OpenMP pragma syntax, which are interpreted simply as comments for standard CPU target builds. The OpenMP offload regions are most beneficial in leveraging the GPU when used around loop constructs in Fortran. A simple example of a standard Fortran loop available for GPU device offload is given in the Figure 3.3 below.

Listing 3.3: Example of simple Fortran loop with omp pragma for device offload

```
!$omp target teams distribute parallel do collapse(3)
do k=1,nz
   do j=1,ny
      do i=1,nx
         data(i,j,k) = rho(i,j,k) * u(i,j,k) * u(i,j,k)
      end do
   end do
end do
!$omp end target teams distribute parallel do
```

In this example, the trivial addition of the OpenMP pragma enables device execution of the loop. We are ignoring the data motion required for this kernel at the moment, but will return to this issue below.

Limitations in OpenMP array support in Fortran

In Miranda loops are rarely expressed as in the example of Figure 3.3. Miranda makes heavy utililization of Fortran array syntax for algebraic operations on its mesh dependent variables, which are expressed as three-dimensional Fortran arrays. The Fortran array syntax computes element-wise operations on equally shaped arrays which has the benefit of source code compactness and brevity, as well as performance with some compilers that more easily can vectorize these operations. Therefore, it is desirable to maintain Miranda's current usage of array syntax.

The previous example of Figure 3.3 in array syntax would be a single line.

Listing 3.4: Example of implicit Fortran loop using array syntax

```
data(:,:,:) = rho(:,:,:) * u(:,:,:) * u(:,:,:)
-or-
data = rho * u * u
```

The current OpenMP specification provides insufficient support for parallelizing array operations in Fortran written using array syntax. The extent of current support allows parallelizing over only one thread team via an OpenMP workshare construct.

Listing 3.5: Example of implicit Fortran loop using array syntax with omp pragma for device offload

```
!$omp target
!$omp parallel workshare
data(:,:,:) = rho(:,:,:) * u(:,:,:) * u(:,:,:)
!$omp end parallel workshare
!$omp end target
-or-
!$omp target
!$omp parallel workshare
data = rho * u * u
!$omp end parallel workshare
!$omp end target
```

To effectively utilize modern GPU's, multiple thread teams are required with a minimum of one thread team running on each of a GPU's streaming multiprocessor (SM) units. The NVIDIA Volta GPU, for example, has 80 SMs. Thus, the current OpenMP workshare construct limits performance of any array syntax kernels on a NVIDIA Volta to at best 1/80 of the GPU's potential.

FEXL: Language transformation for array syntax supported device offload

A solution for performance portability that keeps the array syntax for performance and compactness on the CPU, while enabling a device offload option for GPUs, was achieved by implementing a simple Domain Specific Language (DSL) called FEXL (Fortran EXplicit Loops). FEXL is a python based source code pre-processor that transforms array syntax blocks of code into explicit do loops that are amenable to performant OpenMP offload. The transformation process also inserts the OpenMP pragmas needed and diagnostic calls to Caliper to directly measure individual loop performance. The FEXL pre-processor requires the programmer to explicitly decorate the regions of code where the transformation will take place. These decorators are expressed similar to OpenMP pragmas, where the “!$omp” now becomes “!$FEXL” and the subsequent argument list has changed.

If we return to the simple example of Figure 3.3, we can express the kernel contents in array syntax with the appropriate FEXL pragma as:

Listing 3.6: Example of FEXL loop transformation for a simple loop

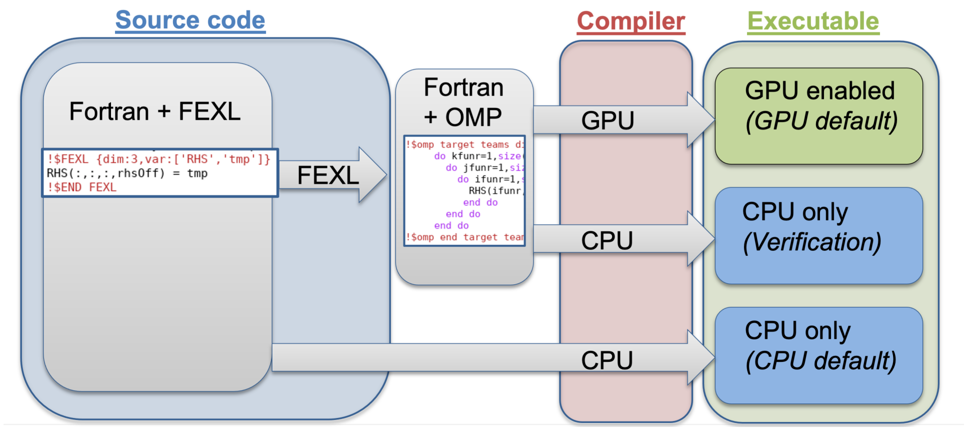
```
!$FEXL {dim:3,vars:[‘data’,‘rho’,‘u’]}         !$omp target teams distribute parallel do collapse(3)
                                               do k=1,nz
                                                  do j=1,ny
                                      ===>           do i=1,nx
data = rho*u*u                       (FEXL)             data(i,j,k)=rho(i,j,k)*u(i,j,k)*u(i,j,k)
                                                     end do
                                                  end do
                                               end do
!$END FEXL                                    !$omp end target teams distribute parallel do
```

More than one line of array syntax Fortran can be encapsulated within the FEXL pragma. Indeed, adding more concurrent computation inside a region helps to amortize the kernel launch overheard incurred by entering a offload region for the device. The programmer must take care to manage the notion of concurrency as loops are decorated for offload.

Note that all code changes are FEXL pragma statements and are therefore ignored when FEXL is not invoked as the pre-processor. The CPU code remains essentially untouched and performance and readability are automatically preserved. In addition, the resultant code produced by FEXL is valid Fortran which allows for code verification to take place on the CPU; that is, the pre-

and post-FEXL code is tested on a comprehensive suite of regression tests on the CPU (with OpenMP turned off) to ensure that the code transformation process has not introduced any changes to the loops or algorithms.

Figure 3.20 below shows the process for loop transformation of valid array syntax Fortran code, to expanded loop expression form with OpenMP pragmas ready for performant device offload. Once a loop is properly decorated with the "!$FEXL" pragma, at compile time (if FEXL is enabled) the loop contained by the pragma gets transformed to fully expanded "do loops". In addition, this loops get decorated with standard OpenMP syntax that directs the runtime to offload the loop to the device, in this case the GPU. The code is then sent to an OpenMP enabled compiler when a GPU enabled executable is desired. The transformed code can also be sent to a CPU compiler (with OpenMP disabled). This option serves as a code transform verification step as it can be compared to the default case, which is to bypass FEXL all together and simply build the code as per the normal pathway on CPU machines. The source-to-source translation process itself takes a few seconds to complete and currently transforms approximately 250 kernels over 20 files.



```
!$FEXL {dim:3,var:['RHS','tmp']}
RHS(:,:,:,rhsOff) = tmp
!$END FEXL
```

```
!$omp target teams di
    do kfunr=1,size
      do jfunr=1,siz
        do ifunr=1,s
          RHS(ifunr,
        end do
      end do
    end do
!$omp end target tean
```

Figure 3.20: Diagram of the code pre-processing step for performant offload of array syntax Fortran via OpenMP4.5. Explicit loops are generated as part of the build step and OpenMP pragmas are inserted. The FEXL DSL is described as valid Fortran and therefore code verification is trivial and porting the code can be done one loop at a time.

Memory management via OpenMP map

OpenMP provides a portable and performant mechanism for transferring data between the host (CPU) and the device (GPU) via its "map" construct. This feature of OpenMP allows the programmer to explicitly manage the movement of data between host and device. If the OpenMP target offload kernel needs to update the variable "data" (in our example from figure 3.3), prior to the kernel launch, the variables "rho" and "u" would need to be mapped from the host to the device, assuming they originated from the host. This is done by mapping the variable as:

```
!$omp target data enter map(to:u,rho)
```

After the update of "data", an exit map is invoked as

```
!$omp target data exit map(from:data)
```

For performance, its desirable to minimize the volume and frequency of data motion between the host and the device. Therefore, our strategy has been to create and entire copy of the hydro state variables on the GPU and only move data for IO, physics coupling, or MPI communication of boundary data. We found that adapting this approach gave an approximate 5x speed up in run time vs. simply mapping data at the subroutine level.

3.7.3 Umpire: Portable Memory Management

Similar to how RAJA and OpenMP provide abstractions for portable kernel execution, *Umpire* [74, 75] provides portable memory hierarchy programming for applications. Umpire is an open source C++ library, developed at LLNL, that emphasizes support for computer platforms with heterogeneous memory systems, such as distinct host-device memories for CPUs and GPUs. The goal of Umpire is to simplify coordination and sharing of limited memory resources for applications that must coordinate memory usage amongst multiple physics packages and libraries. To meet this need, Umpire presents a portable API (Application Programming Interface) that provides operations to query memory resources, provision and allocate memory, and to introspect memory allocation.

Modern computing systems, like Sierra at LLNL, present application developers with complex, hierarchical memory spaces, where each has different access and timing properties. Applications need to move data around the memory hierarchy while carefully managing resources. This complexity is amplified for integrated applications composed of multiple physics packages and libraries. Our experience shows that application components compete for space in GPU main memory, which is limited in size. Since packages need to share this resource and move data between host and device memory, they must carefully coordinate memory usage to efficiently execute problems of interest.

To access and manage complex memory hierarchies, application developers face a variety of vendor-specific APIs. Writing complex memory management logic using these APIs is not portable and requires substantial changes to port applications to a variety of platforms. Umpire's single API applies to memory operations on all platforms by decoupling an application's use of those operations from platform-specific memory spaces and vendor-specific software for using them.

There are four main API concepts in Umpire:

- Memory Resource
- Allocation Strategy
- Allocator
- Memory Operation

The interaction of these concepts is designed to be understood easily by application developers, yet sufficient to support complex allocation schemes and vendor hardware.

Umpire exposes underlying native, vendor-specific memory system calls on different platforms consistently via *memory resources*. Memory resources abstract away differences in vendor API calls for memory allocation and initialization. A memory resource represents a specific kind of memory provided in hardware on a system, distinguished by its location, performance characteristics, and accessibility. Resources are allowed to overlap and a memory resource may be nested inside a parent resource. On Sierra, for example, there are five resources: host memory, device memory, unified memory accessible from both host and device, pinned host memory, and constant device memory.

An *allocation strategy* references available resources, and attempts to limit the total allocated size or preallocate chunks of memory. An *allocator* is the high-level concept that an application uses to allocate and deallocate memory; allocators sit atop allocation strategies. By using a single type to represent all possible resources and allocation strategies, allocators can be easily passed between applications, physics packages, and libraries to share memory resources. Allocation strategies allow sophisticated memory allocation algorithms to be used with any allocators that Umpire provides. Strategies can be nested to allow complex combinations of properties and algorithms to be used when allocating data.

An *operation* is an abstraction of a procedure that modifies or transforms memory, such as copy or move between address ranges, or reallocation. By abstracting operations in this way, users can apply the same high-level operation to memory allocations residing on any resource. When an operation is invoked, the appropriate specialized version is selected based on the memory space location of its arguments, which are typically pointers representing memory addresses.

The potentially high overhead costs of run time memory allocations and deallocations, especially on GPUs, is obviated with Umpire through the use of *memory pools*. Umpire provides several pool types that are useful for HPC applications:

- Fixed pool: a memory pool that returns fixed-size allocations with size provided at construction time
- Dynamic pool: a memory pool that returns allocations of varying sizes on request
- Mixed pool: a replacement for dynamic pool that uses an internal set of memory pools to speed up allocation and deallocation calls at the cost of slightly higher overall memory usage
- Slot pool: a memory pool for a predefined number of allocations of any size
- Monotonic buffer: a very fast allocator that does not support deallocate

In addition, Umpire provides a *thread-safe adapter* that makes any allocator thread-safe and mechanisms to access memory allocation advice, when supported, such as the preferred location of unified memory (i.e., host or device memory space).

Umpire also provides tools to assist developers in debugging memory performance and correctness issues, including logging and *replay*. The logging option can be viewed as a high-level trace, which may be used to track down allocation issues. Replay output includes everything needed to reproduce an application memory state. The replay tool reads files generated by an application when it runs and literally replays allocations without the overhead of the whole application. We have found that by allowing Umpire developers to reproduce problems without the need to run a whole application, turnaround time to resolve issues is substantially reduced. This has proven critical during Umpire development not only to reproduce issues, but the team has also used it to determine whether employing alternative allocation strategies can improve application performance.

Using Umpire with OpenMP

OpenMP supports using memory allocated by external libraries, such as Umpire. Miranda plans to integrate Umpire into its use of OpenMP to take advantage of features that Umpire provides, such as device memory pools.

This code example maps some host memory to the device, with OpenMP allocating the device memory itself.

```
!$omp target map(alloc:h_ptr)
```

This code example maps host memory to the device, with Umpire allocating the device memory.

```
d_ptr = umpire_device_allocator%allocate_pointer(num_bytes)
! This code line will register the memory address pointers on host and device in the OpenMP runtime.
err_code = omp_target_associate_ptr( C_LOC(h_ptr), d_ptr, num_bytes, offset, omp_get_default_device() )
! Fortran requires a map operation to map over the array meta-data ( shape information, offset, etc ).
!$omp target enter data map(always, alloc: h_ptr) )
```

It is important to note that through the use of Shroud discussed in Section 3.2.2, Umpire provides native Fortran and C interfaces in addition to C++.

3.7.4 MFEM: Modular Finite Element Library

Since MFEM serves as the finite element toolkit for Blast, running the code on GPUs required MFEM data structures and routines to be GPU aware. With the release of MFEM 4.0, an intermediate layer was introduced between memory allocations and kernels. In particular, kernels may now be offloaded to a device and kernel execution may be done with number of programming models such as RAJA, OpenMP, CUDA, and OCCA. MFEM's memory manager enables memory allocations with different memory types and tracking modified host and device buffers. To enable fast alllocations and deallocations with device memory, MFEM's memory manager is enhanced by introducing an Umpire memory type. Using Umpire's memory pool capabilities, we are able to use the MFEM memory manager to quickly draw and release from a pre-allocated pool of memory overcoming expensive memory device memory allocations and deallocations costs. Using Umpire memory pools is particularly useful when temporary memory device buffers are needed in computational routines. Figure 3.21 illustrates the intermediate layer between memory allocations and kernel execution.

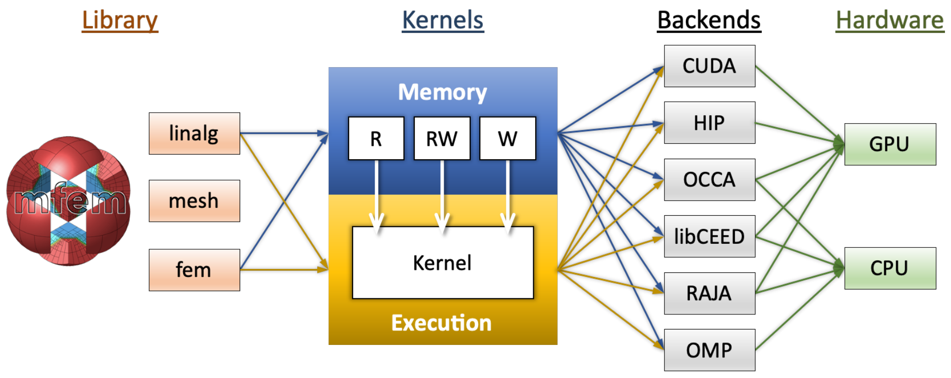


Figure 3.21: MFEM 4.0 introduced an intermediate layer between memory allocations and kernel execution. Memory allocation may be done with different memory types, the memory manager keeps track of modified data. Kernel execution may be done with a variety of different backends which can be used to target GPUs and CPUs.

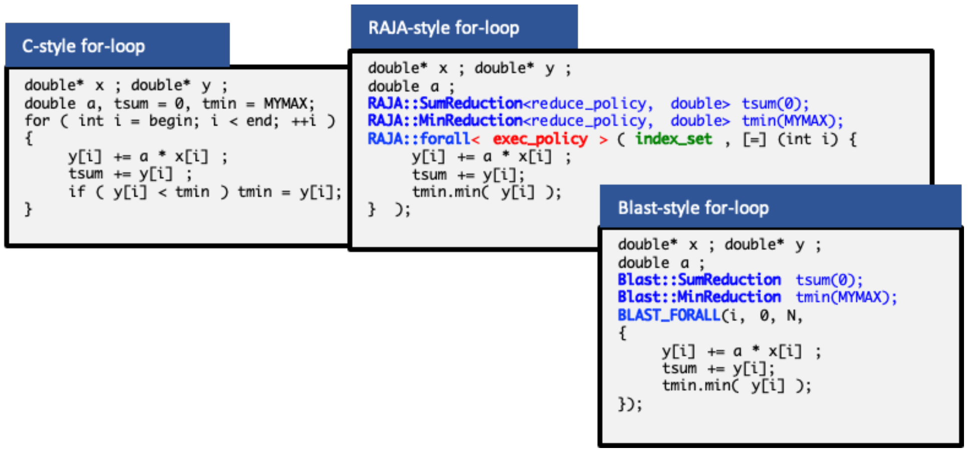
3.7.5 Development with RAJA, MFEM, and Runtime Execution Policy Selection

To simplify kernel porting and choosing execution policy types, kernels in Blast are expressed using one of two macros: BLAST_FORALL and BLAST_FORALL_TEAMS. The BLAST_FORALL macro serves as an abstraction layer over the RAJA forall method while BLAST_FORALL_TEAMS serves as an abstraction layer for kernels with nested loops. The BLAST_FORALL_TEAMS macro enabled users to explore new semantics for expressing kernels beyond what is was supported in the RAJA kernel interface. Additionally, it served as a tool for engaging with the RAJA team leading to the development of the RAJA teams API. The development of the RAJA teams API is discussed in Section 3.7.6.

Beyond a simplified kernel development layer, the macro layer is used to complement MFEM's run-time execution policy selection support. Through the MFEM abstraction layer, users may choose a back-end for kernel execution by specifying a string value with MFEM's device object. To compliment this flexibility, the Blast macro layer executes the appropriate back-end based on the MFEM device setting. Figure 3.22 compares a C-style for loop, RAJA style loop, and the Blast interface for RAJA. The Blast interface removes template specialization as execution is based on the MFEM device configuration. Additional details like specifying lambda capture types and defining RAJA iteration spaces are also simplified as all lambdas are set to be capture by value, which is required for GPU execution, and execution dispatch is based on the MFEM device configuration.

As a starting point for GPU porting, kernels were exclusively ported to use the BLAST_FORALL macro. Much of the early porting effort required refactoring kernels to use simple data structures and methods amenable to both host and device execution. Many kernels had to be refactored as existing data structures owned memory and capturing within a lambda would trigger copy operations to replicate objects and their data.

Refactoring kernels in Blast using BLAST_FORALL enabled the Blast computer science (CS) team to focus on correctness and ensuring kernels would be compatible with the RAJA portability layer A second component that played a vital role in refactoring Blast for GPUs was MFEM's memory manager. Memory movement through MFEM's memory manager is done explicitly through Read and Write methods that move buffers of data to the appropriate memory space. If the code tried to compute with data that had not been moved to the correct memory space, MFEM debugging tools would recognize data as invalid and emit an error


C-style for-loop
RAJA-style for-loop
Blast-style for-loop

```
double* x ; double* y ;
double a, tsum = 0, tmin = MYMAX;
for ( int i = begin; i < end; ++i )
{
     y[i] += a * x[i] ;
     tsum += y[i] ;
     if ( y[i] < tmin ) tmin = y[i];
}
```

```
double* x ; double* y ;
double a ;
RAJA::SumReduction<reduce_policy, double> tsum(0);
RAJA::MinReduction<reduce_policy, double> tmin(MYMAX);
RAJA::forall< exec_policy > ( index_set , [=] (int i) {
     y[i] += a * x[i] ;
     tsum += y[i];
     tmin.min( y[i] );
} );
```

```
double* x ; double* y ;
double a ;
Blast::SumReduction tsum(0);
Blast::MinReduction tmin(MYMAX);
BLAST_FORALL(i, 0, N,
{
     y[i] += a * x[i] ;
     tsum += y[i];
     tmin.min( y[i] );
});
```

Figure 3.22: MFEM 4.0 introduces an intermediate layer between memory allocations and kernel execution. Memory allocation may be done with different memory types, the memory manager keeps track of modified data. Kernel execution may be done with a variety of different back-end which can be used to target GPUs and CPUs.

message. The MFEM memory tracking capabilities enabled the Blast CS team to ensure data was moving correctly between host and device memory spaces.

RAJA forall methods served as good starting point. However, parallelism was limited at the element level and improved performance would require exposing hierarchical parallelism. Prior to the development of the RAJA teams API, a small prototype was developed which combined the *RAJA kernel* API and MFEM style macros. Listing 3.7 illustrates the teams prototype in Blast. In the context of a finite element code, teams are mapped to elements and TEAM_LOOP's expand to CUDA threads or regular C-style for loops depending on whether the code was compiled for the host or device. To maintain run-time selectivity, both host and device variants of the code are instantiated and the kernel is dispatched based on the MFEM device configuration.

Listing 3.7: The first prototype of RAJA teams consisted of macros.

```
//Teams are distributed directly to CUDA blocks or standard C loops
BLAST_FORALL_TEAMS(i, NoTeams, TeamSz_x, TeamSz_y, TeamSz_z,
{

  //Memory available to threads in a team
  TEAM_SHARED double A[][][];

  //Work at the team level is distributed to CUDA threads
  //in a thread block or executed via c-style for loops
  TEAM_LOOP_3D(tx, ty, tz, Nx, Ny, Nz,
  {
    //Load data into share memory
    A[tx][ty][tz] = ...
  });

  //Thread synchronization to ensure work within a team is complete
  TEAM_SYNC();
});
```

The first teams prototype enabled further performance improvements by exposing hierarchical parallelism and GPU shared memory. Although the RAJA kernel API has native capabilities to expose shared memory and hierarchical parallelism, it is accomplished through template specialization which can become verbose and require constructing specialized policies for each kernel. To simplify development of complex kernels, a co-design effort with RAJA developers led to the RAJA teams concept where an underlying goal was to develop a portable abstraction layer that maintains the look and feel of nested C-style for-loops.

3.7.6 Development of the RAJA Teams API

RAJA forall methods provide a simple interface for developing portable code, however; RAJA forall methods do not expose key GPU programming features such as hierarchical parallelism and shared memory which have been shown to improve performance for certain GPU kernels. Programming models such as CUDA, OpenCL, and HIP enable programming GPUs through the notion of threads, and thread blocks (or work items and work groups in OpenCL nomenclature). Under these programming models, computation on the GPU is performed using a specified grid of compute units. Following NVIDIA's nomenclature, each unit of the grid is referred to as a thread. Threads are grouped to form thread blocks. The hardware provides a similar hierarchy for memory. Threads are provided with a small amount of exclusive memory, threads in a thread block share block exclusive memory (shared memory), and lastly the entire compute grid shares global memory. Moving data between the CPU and GPU is accomplished through the use of global memory which is accessible from both host and device. An in depth description regarding GPU computing may be found in [76, 77].

To unify the computational hierarchies provided by the various GPU programming models, a cross project collaboration between the MARBL, RAJA, and MFEM teams lead to the co-design of the RAJA teams API. RAJA teams provides a kernel language which enables abstraction of a computational hierarchy and enables run-time host device kernel execution selection. Similar to other GPU programming models, RAJA teams enables developers to compute on a specified compute grid, where compute entities (threads) are grouped into teams, a scratch pad memory type which is accessible to threads in the same team is available through the "RAJA_TEAM_SHARED" macro exposing GPU shared memory (or CPU stack memory). Figure 3.23 illustrates the computational grid used as an abstraction for RAJA teams. The figure shows how threads are grouped together to form teams. Teams on the RAJA compute grid naturally map to GPU thread blocks, while threads in a team map naturally to GPU threads in a thread block. Execution of RAJA teams kernels on the host using RAJA sequential policies expand to C-style for-loops where work at the team and thread level is processed sequentially. The OpenMP programming model is also supported with RAJA teams and developers may choose to execute teams in parallel or threads in parallel. In future work, we plan to provide a direct mapping to OpenMP team and thread constructs.

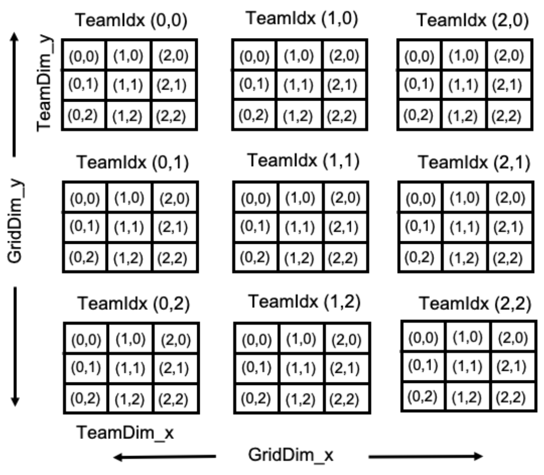


Figure 3.23: The RAJA teams API unifies GPU programming models which perform computation on a specified compute grid. RAJA teams enables a similar abstraction level found in many GPU programming languages by introducing the concept of teams into which threads are grouped. The figure above illustrates a two-dimensional compute grid with 9 teams (3×3) each with 9 threads (3×3). The RAJA Teams API enables developers to directly map threads and teams to GPU threads and thread blocks (following CUDA nomenclature). Host execution results in the work for each team to be done through standard for-loops, as well as the work at the thread level.

In the following section, we provide a brief overview of the RAJA teams API. There are three main API concepts in RAJA teams:

- **Launch policy**
- **RAJA loop policies (Threads and Teams)**
- **Launch method**

The **launch policy** is used to determine how the kernel will be executed. Supported host back-ends are sequential, and OpenMP, and device back-end support includes CUDA and HIP. To enable run-time selection, the RAJA launch policy is templated on both host and device policies as illustrated in Listing 3.8.

Listing 3.8: RAJA team launch types are templated on host/device policies enabling run-time selection

```
using launch_policy = RAJA::expt::LaunchPolicy<RAJA::expt::seq_launch_t
                                              ,RAJA::expt::cuda_launch_t<false>>;
```

Similarly, the **LoopPolicy** type is templated on both host and device execution policies to support run-time selectivity. Listing 3.9 illustrates choosing standard C-style loop execution for host and mapping team and thread loops to CUDA thread blocks and threads for device execution.

Listing 3.9: RAJA loop policy types are templated on both a host/device enabling run-time selection

```
using teams_x = RAJA::expt::LoopPolicy<RAJA::loop_exec
                                      ,RAJA::cuda_block_x_direct>;

using threads_x = RAJA::expt::LoopPolicy<RAJA::loop_exec
                                        ,RAJA::cuda_thread_x_direct>;
```

Once the launch policy and team and thread policies have been defined, developers may reuse the policies with multiple kernels. Listing 3.10 illustrates the interplay of the launch and loop types with the launch method in the RAJA teams API. Additional parameters are passed to the launch method in a resources object that holds the number of teams and threads in the RAJA compute grid. Additionally, the figure illustrates how kernels may be expressed as nested for-loops.

Listing 3.10: RAJA::Teams nested loop

```
RAJA::expt::launch<launch_policy>(select_cpu_or_gpu,
RAJA::expt::Resources(RAJA::expt::Teams(N_tri), RAJA::expt::Threads(N_tri)),
[=] RAJA_HOST_DEVICE(RAJA::expt::LaunchContext ctx) {

  RAJA::expt::loop<teams_x>(ctx, RAJA::RangeSegment(0, N_tri), [&](int r) {

    // Array shared within threads of the same team
    RAJA_TEAM_SHARED int s_A[N_tri];

    RAJA::expt::loop<threads_x>(ctx, RAJA::RangeSegment(0, innerRange), [&](int c) {
      //Load data into shared memory
    });  // loop c

    ctx.teamSync();

    RAJA::expt::loop<threads_x>(ctx, RAJA::RangeSegment(0, innerRange), [&](int c) {
      //Use data in shared memory
    });  // loop c

  });  // loop r
});  // outer lambda
```

To demonstrate the performance improvements the RAJA teams API enabled, we begin by considering the kernel for applying the action of the mass matrix via partial assembly. The action of the matrix is used in iterative solvers for both the Lagrange and Remap phase. As a starting point, we express the kernel using a RAJA forall method, and assess performance using the CEED benchmark suite for bake-off problem 1 (BP1), a linear solve with the mass matrix. Figure 3.24 presents the achieved throughput performance for varying orders. Notably, the figure shows that peak performance performance is roughly at 4.4e8 (dofs $\times$ CG iterations) / seconds, in these experiments we limit the number of compute nodes to 1 and experiments are carried out on Lassen. Lassen is equipped with four NVIDIA V100's. Figures 3.25a and 3.25b illustrate performance improvements when hierarchical parallelism and shared memory is introduced. Figure 3.25a presents performance when using a macro layer for the CUDA programming model, while Figure 3.25b shows performance when using RAJA teams. We find that RAJA teams achieves a comparable performance to the CUDA macro layer, and using shared memory and hierarchical parallelism improves performance (by a factor at least 2$\times$) over only exposing parallelism at the element level. For certain kernels in Blast, we observed 5$\times$ improvements over forall.

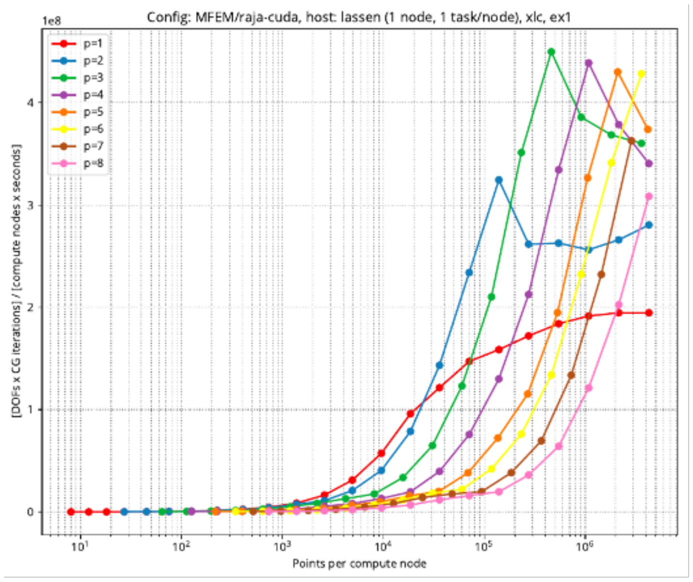


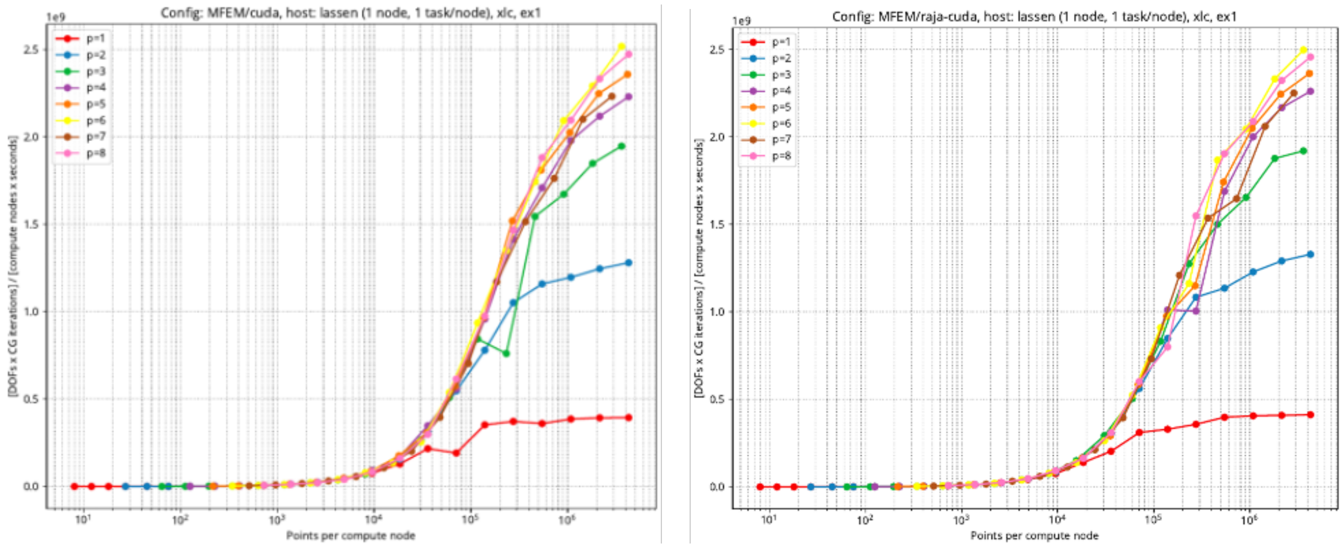
Figure 3.24: Conjugate gradient solver performance for solving a linear system with the mass matrix. The action of the mass matrix is applied through a matrix-free partial assembly algorithm, and RAJA forall is used to execute the kernel via CUDA. Parallelism through RAJA forall is limited at the element level.

3.7.7 Memory Interface Abstraction

A memory abstraction was added in Exo to provide access to a variety of Umpire allocators in a concise manner. This abstraction establishes a centralized location for initializing and storing access to multiple Umpire memory allocators and pools. Additionally, the abstraction provides an interface of useful functions for managing memory allocations. This abstraction was created in Exo to provide a centralized location for each package in MARBL to have access to a memory management capability.

The memory abstraction allocators that are used are:

- Host - An interface to Umpire default HOST allocator.
- Pinned - An interface to HostPinned memory.
- Temporary - An interface to a Dynamic Memory Pool tied to the DEVICE or HOST allocator (runtime decision tied to RAJA device). Temporary memory is used for memory which is only needed for a single cycle or less.
- Permanent - An interface to a Dynamic Memory Pool tied to the DEVICE or HOST allocator (runtime decision tied to RAJA device). Permanent memory is used for memory which is needed for the lifetime of the run.



(a) CUDA implementation of conjugate gradient solve

(b) RAJA teams implementation of conjugate gradient solve

Figure 3.25: Conjugate gradient performance, at different polynomial orders (colored curves), for solving a linear system with the mass matrix. We observed similar performance when the shared memory and hierarchical parallelism were expressed with (a) a CUDA macro layer and with (b) RAJA teams. Note that, compared to Figure 3.24, the performance is an order of magnitude better.

- Managed - An interface to a Dynamic Memory Pool tied to the UM Umpire allocator.

The memory abstraction interface functions are:

- Allocate(Size, AllocatorName) : Allocates memory using the given allocator
- Deallocate(Pointer, AllocatorName) : Deallocates memory from the given allocator
- Move(Pointer, AllocatorName) : Moves the memory associated with the pointer to the given allocator (introspection required)
- GetAllocator(AllocatorName) : Returns the Umpire allocator
- Coalesce(AllocatorName) : Defragment memory in the given allocator (pooled allocator and introspection required)
- Release(AllocatorName) : Release all unused memory associated with a memory pool (pooled allocator required).

Blast uses this memory abstraction interface for all of its GPU data allocations. The temporary memory abstraction is used for variables needed for only a kernel or subset of kernels; this helps reduce the memory high water mark while eliminating allocation and deallocation overhead. The permanent memory abstraction is used for variables which need to exist for the majority of the application run duration. Additionally, Blast integrated these memory abstractions into third party libraries by passing the allocators into their interface functions. This enables LEOS, Leilak, and MFEM access to these memory pools, which can significantly increase memory performance of the libraries when used in conjunction with Blast.

A PooledBuffer class was added to Blast to enable a clean and scoped interface to memory allocations. The PooledBuffer uses the memory abstraction under the covers by providing memory into the appropriate memory pool, through a templated interface which resembles the familiar std::vector. PooledBuffer provides a shorthand definition to each of the allocation policies which is available in the memory abstraction interface.

Listing 3.11: Memory Policy Definitions

```
#define HOSTMEM exo::memory_interface::AllocationPolicy::Host
#define PINDMEM exo::memory_interface::AllocationPolicy::HostPinned
#define PERMMEM exo::memory_interface::AllocationPolicy::Permanent
#define TEMPMEM exo::memory_interface::AllocationPolicy::Temporary
#define MANGMEM exo::memory_interface::AllocationPolicy::Managed
```

Given these definitions and the PooledBuffer class, creating a GPU memory buffer and accessing the pointer is clear and concise. In addition the scope of the buffer provides automatic cleaning up of the buffers. The following example shows the process of allocating memory in the temporary pool and getting a pointer to that memory to send into a kernel when using the PooledBuffer class.

Listing 3.12: PooledBuffer Temporary Pool example

```
PooledBuffer<double> Buffer(length, TEMPMEM);
double *buffer = Buffer.GetBuffer();
```

3.7.8 LC tools: Caliper, SPOT, Hatchet

To aid the development and porting of MARBL to GPUs, we heavily leveraged Livermore Computing tools like Caliper, Spot, and Hatchet.

We annotate our source code with Caliper [78], a generic context annotation tool developed at LLNL. Specifically, we annotate the phases of the execution, functions, and kernels. At runtime, Caliper provides the capability to measure execution time, MPI performance, and a selection of architecture-specific hardware counters. Caliper attributes these measurements to the annotated code regions. Caliper efficiently handles per-process measurements, and the data aggregation and reduction to ensure the low overhead of the measurements in a massively parallel code.

In addition to time measurement, Caliper provides many services. Two of the services were particularly useful for MARBL development:

1. The NVProf service takes Caliper annotations and turns them into NVTX ranges on NVIDIA GPUs, allowing the code developers to see those annotations in NVIDIA tools such as NVProf, NVVP, and the Nsight tool suite.

2. The annotation counting service (a config using the aggregate, event, and report services) counts how many times a specific annotation was invoked in the course of the run. If a specific annotation is invoked very frequently (e.g., from inside of the loop), it can cause performance overhead. Using this service, we were able to verify that we are not overly instrumenting our code and not causing performance overhead due to instrumenting inner most functions.

In the default setting, Caliper collects the runtime of the application across processes, writing a single file per run. A web-based tool called Spot [79] was developed by Livermore Computing to enable the application developers to view this collected performance data in a browser. If performance data is collected repeatedly (e.g., during nightly tests), Spot will display the performance over time, enabling the application developers to quickly “spot” the changes in performance. Spot also provides summaries of systems the data was collected on, along with other metadata about the runs.

Figures 3.26 and 3.27 show the Spot view of the performance of the TriplePoint3D problem on rzansel and rzgenie. The figures shown display the runtime breakdown by function for the months of January, February, March, April, and May. Each figure shows the performance during the month, with dates on the x-axis. The figures make it easy to “spot” the dates on which the performance of certain functions improved, which can be correlated to the code changes committed to the repo on that date.

Figure 3.28 and 3.29 similarly show the Spot view of the performance of the brl81 problem on rzansel and rzgenie January 2020 through May 2020. Again, we observe the development points which resulted in performance improvements of certain routines.

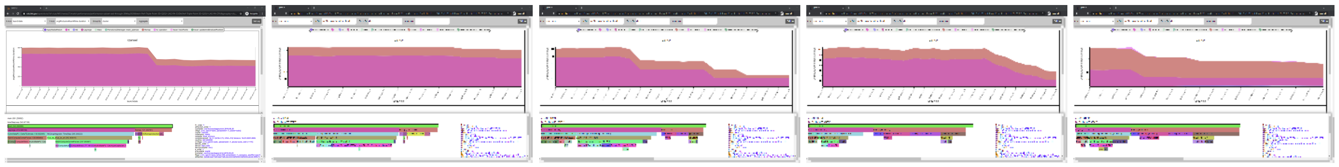


Figure 3.26: Spot view of TriplePoint3D performance over time (rzansel)

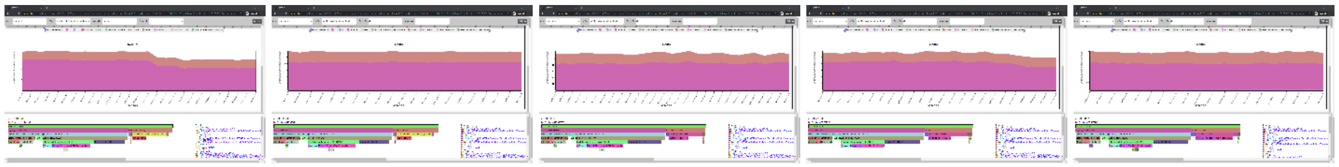


Figure 3.27: Spot view of TriplePoint3D performance over time (rzgenie)

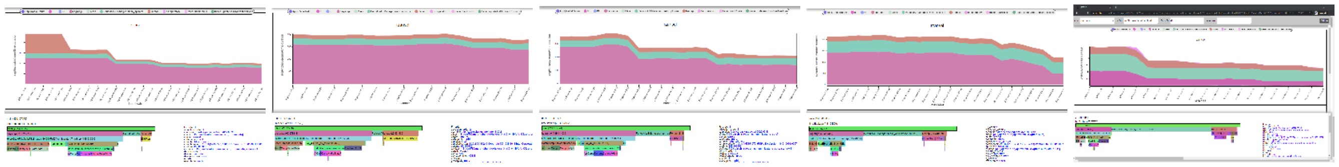


Figure 3.28: Spot view of brl81 performance over time (rzansel)

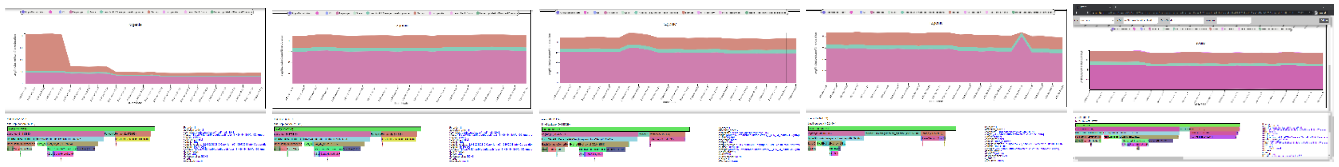


Figure 3.29: Spot view of brl81 performance over time (rzgenie)

The data also contains the “bumps” along the way which correspond to changes in the software stack on the machine or I/O problems, etc. These temporal changes are easy to distinguish from the more permanent effect of the code performance improvement.

To measure our weekly progress during porting to GPUs, we used Hatchet [80] to look at the performance in detail. Hatchet reads the same data Caliper generates for Spot, and allows for the user to analyze this performance data programmatically. A Jupyter notebook tool, Hatchet is a Python library with functionality to read, compare, and view the performance data. At the core of Hatchet is the functionality to view the call tree of the application run, and the times spent in each function. The call tree visualization includes the colormap of times in the functions, clearly showing the ones that that the most or the least time during the execution. The power of the call tree structure in Hatchet being a proper data structure is that the user can then compare calltrees of different runs of the application. For example, speedup can be calculated by “dividing” two trees, and the resulting call tree can be displayed, showing the speedup for each function in the call tree.

Figure 3.30 shows Hatchet visualization of the TriplePT problem call trees. The three-way chart shows:

1. Call tree of the GPU run, displaying the runtime for each of the functions while running MARBL on rzansel;
2. Speedup per function, computed with a single line in Hatchet as CPUtime/GPUtime;
3. Call tree on the CPU as the baseline for the GPU speedup work.

The legend coloring in Hatchet quickly and conveniently highlights the functions that take the most time in the “time call tree”, and functions that show the least speedup in the “speedup call tree.”

Hatchet also allows quick and easy filtering of the call trees to help the user look at the data they prefer to focus on. Figure 3.31 shows the GPU speedup regions where the speedup is greater than 1 (on the left), and the GPU slowdowns (on the right). This functionality enables us to quickly focus in on the portion of the code that needs additional work to perform well.

Additionally, once the data set is read into Hatchet, the performance data is stored in Pandas dataframes, and therefore fully accessible through the Python functionality. This has enabled us to build up custom scripts which we used to generate custom weekly reports of our performance progress, as described in the following section.

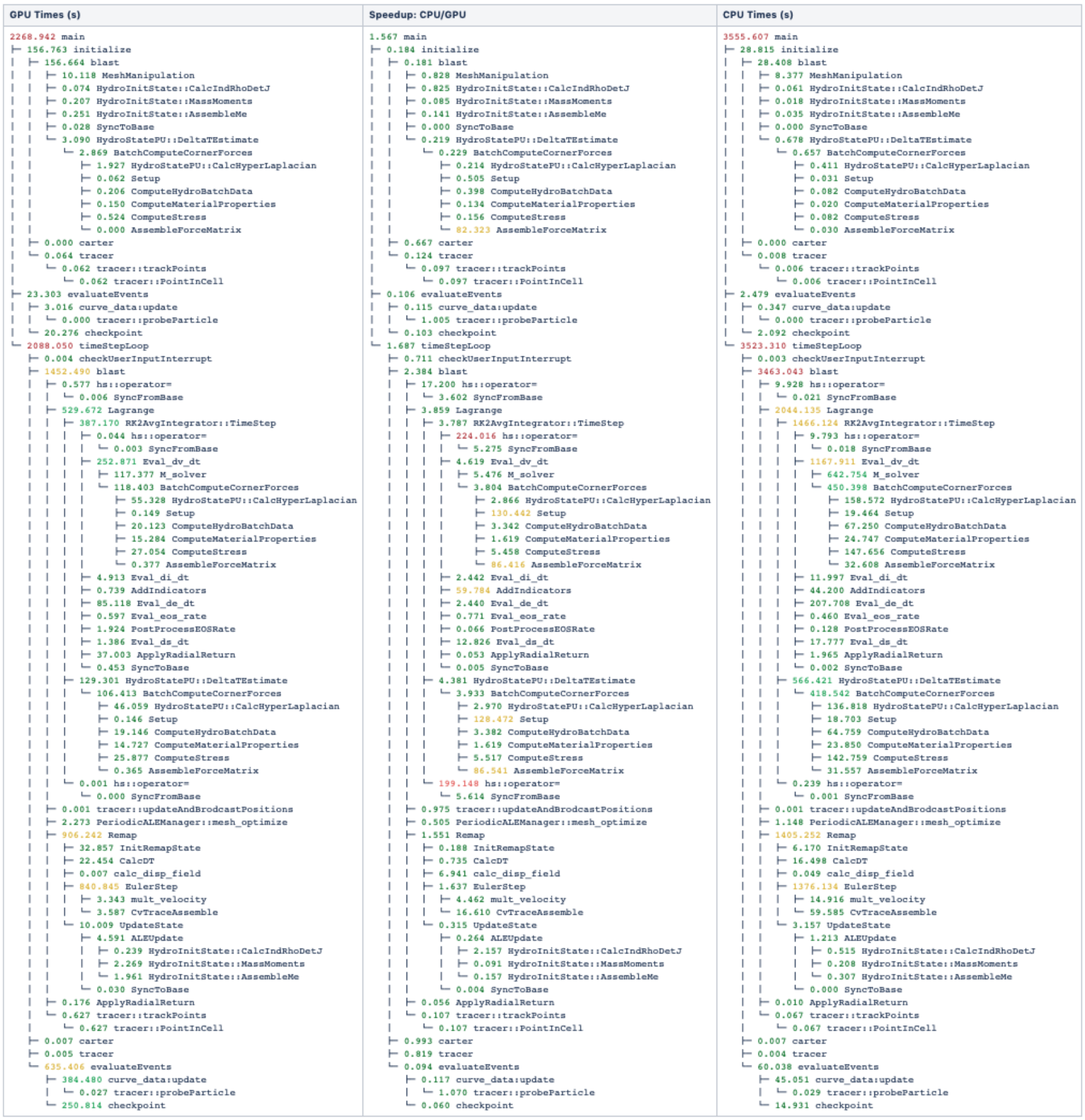
Overall, the Livermore Computing tools have been hugely beneficial to our project in that they enabled us to track the performance of our code over time, and to get at the details of that performance to guide where to allocate our development time to improve performance.

3.7.9 Porting history and timeline

MARBL-lag

As MFEM serves as the finite element toolkit for the MARBL lagrangian package, the portability strategy was heavily influenced by the GPU capabilities of MFEM 4.0. Prior to the release of MFEM 4.0, technology and capabilities were explored by members of both the MARBL and MFEM teams in the context of small examples relevant to MARBL. With the release of MFEM 4.0 (May 2019) a intermediate layer was introduced between basic MFEM data structures and kernels. Basic data structures were made aware of host and device memory spaces, and kernels were expressed in an abstraction layer to enable offloading to device. In the context of the MARBL, kernels were expressed using the RAJA abstraction layer.

As a starting point the RAJA forall method was used to port kernels in MARBL’s lagrangian package. To enable run-time selection of host/device policies, kernels are expressed using macros which forward the lambda body to the appropriate templated RAJA policy. Data transfers are handled using MFEM’s memory manager which requires developers to make explicit calls to transfer data to the the host or device memory space. Equipped with RAJA forall methods and MFEM’s memory manager, the MARBL team was able to focus on refactoring kernels to make them compatible with the RAJA abstraction layer and making sure



GPU Times (s)

```
2268.942 main
├─ 156.763 initialize
│   ├─ 156.664 blast
│   │   ├─ 10.118 MeshManipulation
│   │   ├─ 0.074 HydroInitState::CalcIndRhoDetJ
│   │   ├─ 0.207 HydroInitState::MassMoments
│   │   ├─ 0.251 HydroInitState::AssembleMe
│   │   ├─ 0.028 SyncToBase
│   │   └─ 3.090 HydroStatePU::DeltaTEstimate
│   │       └─ 2.869 BatchComputeCornerForces
│   │           ├─ 1.927 HydroStatePU::CalcHyperLaplacian
│   │           ├─ 0.062 Setup
│   │           ├─ 0.206 ComputeHydroBatchData
│   │           ├─ 0.150 ComputeMaterialProperties
│   │           ├─ 0.524 ComputeStress
│   │           └─ 0.000 AssembleForceMatrix
│   ├─ 0.000 carter
│   └─ 0.064 tracer
│       └─ 0.062 tracer::trackPoints
│           └─ 0.062 tracer::PointInCell
├─ 23.303 evaluateEvents
│   ├─ 3.016 curve_data:update
│   │   └─ 0.000 tracer::probeParticle
│   └─ 20.276 checkpoint
└─ 2088.050 timeStepLoop
    ├─ 0.004 checkUserInputInterrupt
    ├─ 1452.490 blast
    │   ├─ 0.577 hs::operator=
    │   │   └─ 0.006 SyncFromBase
    │   ├─ 529.672 Lagrange
    │   │   ├─ 387.170 RK2AvgIntegrator::TimeStep
    │   │   │   ├─ 0.044 hs::operator=
    │   │   │   │   └─ 0.003 SyncFromBase
    │   │   │   ├─ 252.871 Eval_dv_dt
    │   │   │   │   ├─ 117.377 M_solver
    │   │   │   │   └─ 118.403 BatchComputeCornerForces
    │   │   │   │       ├─ 55.328 HydroStatePU::CalcHyperLaplacian
    │   │   │   │       ├─ 0.149 Setup
    │   │   │   │       ├─ 20.123 ComputeHydroBatchData
    │   │   │   │       ├─ 15.284 ComputeMaterialProperties
    │   │   │   │       ├─ 27.054 ComputeStress
    │   │   │   │       └─ 0.377 AssembleForceMatrix
    │   │   │   ├─ 4.913 Eval_di_dt
    │   │   │   ├─ 0.739 AddIndicators
    │   │   │   ├─ 85.118 Eval_de_dt
    │   │   │   ├─ 0.597 Eval_eos_rate
    │   │   │   ├─ 1.924 PostProcessEOSRate
    │   │   │   ├─ 1.386 Eval_ds_dt
    │   │   │   ├─ 37.003 ApplyRadialReturn
    │   │   │   └─ 0.453 SyncToBase
    │   │   ├─ 129.301 HydroStatePU::DeltaTEstimate
    │   │   │   └─ 106.413 BatchComputeCornerForces
    │   │   │       ├─ 46.059 HydroStatePU::CalcHyperLaplacian
    │   │   │       ├─ 0.146 Setup
    │   │   │       ├─ 19.146 ComputeHydroBatchData
    │   │   │       ├─ 14.727 ComputeMaterialProperties
    │   │   │       ├─ 25.877 ComputeStress
    │   │   │       └─ 0.365 AssembleForceMatrix
    │   │   └─ 0.001 hs::operator=
    │   │       └─ 0.000 SyncFromBase
    │   ├─ 0.001 tracer::updateAndBrodcastPositions
    │   ├─ 2.273 PeriodicALEManager::mesh_optimize
    │   ├─ 906.242 Remap
    │   │   ├─ 32.857 InitRemapState
    │   │   ├─ 22.454 CalcDT
    │   │   ├─ 0.007 calc_disp_field
    │   │   ├─ 840.845 EulerStep
    │   │   │   ├─ 3.343 mult_velocity
    │   │   │   └─ 3.587 CvTraceAssemble
    │   │   └─ 10.009 UpdateState
    │   │       ├─ 4.591 ALEUpdate
    │   │       │   ├─ 0.239 HydroInitState::CalcIndRhoDetJ
    │   │       │   ├─ 2.269 HydroInitState::MassMoments
    │   │       │   └─ 1.961 HydroInitState::AssembleMe
    │   │       └─ 0.030 SyncToBase
    │   ├─ 0.176 ApplyRadialReturn
    │   └─ 0.627 tracer::trackPoints
    │       └─ 0.627 tracer::PointInCell
    ├─ 0.007 carter
    ├─ 0.005 tracer
    └─ 635.406 evaluateEvents
        ├─ 384.480 curve_data:update
        │   └─ 0.027 tracer::probeParticle
        └─ 250.814 checkpoint
```

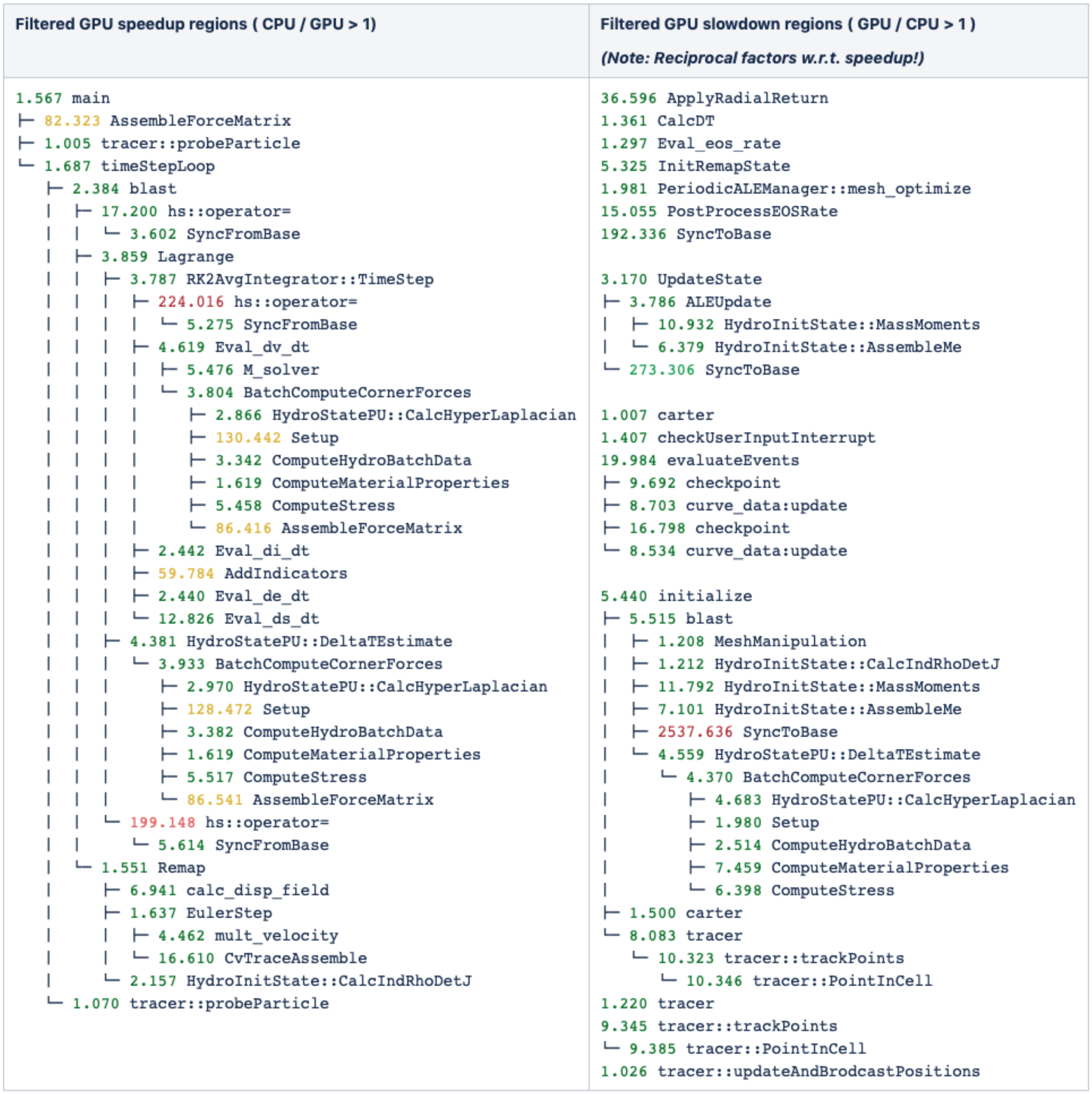
Speedup: CPU/GPU

```
1.567 main
├─ 0.184 initialize
│   ├─ 0.181 blast
│   │   ├─ 0.828 MeshManipulation
│   │   ├─ 0.825 HydroInitState::CalcIndRhoDetJ
│   │   ├─ 0.085 HydroInitState::MassMoments
│   │   ├─ 0.141 HydroInitState::AssembleMe
│   │   ├─ 0.000 SyncToBase
│   │   └─ 0.219 HydroStatePU::DeltaTEstimate
│   │       └─ 0.229 BatchComputeCornerForces
│   │           ├─ 0.214 HydroStatePU::CalcHyperLaplacian
│   │           ├─ 0.505 Setup
│   │           ├─ 0.398 ComputeHydroBatchData
│   │           ├─ 0.134 ComputeMaterialProperties
│   │           ├─ 0.156 ComputeStress
│   │           └─ 82.323 AssembleForceMatrix
│   ├─ 0.667 carter
│   └─ 0.124 tracer
│       └─ 0.097 tracer::trackPoints
│           └─ 0.097 tracer::PointInCell
├─ 0.106 evaluateEvents
│   ├─ 0.115 curve_data:update
│   │   └─ 1.005 tracer::probeParticle
│   └─ 0.103 checkpoint
└─ 1.687 timeStepLoop
    ├─ 0.711 checkUserInputInterrupt
    ├─ 2.384 blast
    │   ├─ 17.200 hs::operator=
    │   │   └─ 3.602 SyncFromBase
    │   ├─ 3.859 Lagrange
    │   │   ├─ 3.787 RK2AvgIntegrator::TimeStep
    │   │   │   ├─ 224.016 hs::operator=
    │   │   │   │   └─ 5.275 SyncFromBase
    │   │   │   ├─ 4.619 Eval_dv_dt
    │   │   │   │   ├─ 5.476 M_solver
    │   │   │   │   └─ 3.804 BatchComputeCornerForces
    │   │   │   │       ├─ 2.866 HydroStatePU::CalcHyperLaplacian
    │   │   │   │       ├─ 130.442 Setup
    │   │   │   │       ├─ 3.342 ComputeHydroBatchData
    │   │   │   │       ├─ 1.619 ComputeMaterialProperties
    │   │   │   │       ├─ 5.458 ComputeStress
    │   │   │   │       └─ 86.416 AssembleForceMatrix
    │   │   │   ├─ 2.442 Eval_di_dt
    │   │   │   ├─ 59.784 AddIndicators
    │   │   │   ├─ 2.440 Eval_de_dt
    │   │   │   ├─ 0.771 Eval_eos_rate
    │   │   │   ├─ 0.066 PostProcessEOSRate
    │   │   │   ├─ 12.826 Eval_ds_dt
    │   │   │   ├─ 0.053 ApplyRadialReturn
    │   │   │   └─ 0.005 SyncToBase
    │   │   ├─ 4.381 HydroStatePU::DeltaTEstimate
    │   │   │   └─ 3.933 BatchComputeCornerForces
    │   │   │       ├─ 2.970 HydroStatePU::CalcHyperLaplacian
    │   │   │       ├─ 128.472 Setup
    │   │   │       ├─ 3.382 ComputeHydroBatchData
    │   │   │       ├─ 1.619 ComputeMaterialProperties
    │   │   │       ├─ 5.517 ComputeStress
    │   │   │       └─ 86.541 AssembleForceMatrix
    │   │   └─ 199.148 hs::operator=
    │   │       └─ 5.614 SyncFromBase
    │   ├─ 0.975 tracer::updateAndBrodcastPositions
    │   ├─ 0.505 PeriodicALEManager::mesh_optimize
    │   ├─ 1.551 Remap
    │   │   ├─ 0.188 InitRemapState
    │   │   ├─ 0.735 CalcDT
    │   │   ├─ 6.941 calc_disp_field
    │   │   ├─ 1.637 EulerStep
    │   │   │   ├─ 4.462 mult_velocity
    │   │   │   └─ 16.610 CvTraceAssemble
    │   │   └─ 0.315 UpdateState
    │   │       ├─ 0.264 ALEUpdate
    │   │       │   ├─ 2.157 HydroInitState::CalcIndRhoDetJ
    │   │       │   ├─ 0.091 HydroInitState::MassMoments
    │   │       │   └─ 0.157 HydroInitState::AssembleMe
    │   │       └─ 0.004 SyncToBase
    │   ├─ 0.056 ApplyRadialReturn
    │   └─ 0.107 tracer::trackPoints
    │       └─ 0.107 tracer::PointInCell
    ├─ 0.993 carter
    ├─ 0.819 tracer
    └─ 0.094 evaluateEvents
        ├─ 0.117 curve_data:update
        │   └─ 1.070 tracer::probeParticle
        └─ 0.060 checkpoint
```

CPU Times (s)

```
3555.607 main
├─ 28.815 initialize
│   ├─ 28.408 blast
│   │   ├─ 8.377 MeshManipulation
│   │   ├─ 0.061 HydroInitState::CalcIndRhoDetJ
│   │   ├─ 0.018 HydroInitState::MassMoments
│   │   ├─ 0.035 HydroInitState::AssembleMe
│   │   ├─ 0.000 SyncToBase
│   │   └─ 0.678 HydroStatePU::DeltaTEstimate
│   │       └─ 0.657 BatchComputeCornerForces
│   │           ├─ 0.411 HydroStatePU::CalcHyperLaplacian
│   │           ├─ 0.031 Setup
│   │           ├─ 0.082 ComputeHydroBatchData
│   │           ├─ 0.020 ComputeMaterialProperties
│   │           ├─ 0.082 ComputeStress
│   │           └─ 0.030 AssembleForceMatrix
│   ├─ 0.000 carter
│   └─ 0.008 tracer
│       └─ 0.006 tracer::trackPoints
│           └─ 0.006 tracer::PointInCell
├─ 2.479 evaluateEvents
│   ├─ 0.347 curve_data:update
│   │   └─ 0.000 tracer::probeParticle
│   └─ 2.092 checkpoint
└─ 3523.310 timeStepLoop
    ├─ 0.003 checkUserInputInterrupt
    ├─ 3463.043 blast
    │   ├─ 9.928 hs::operator=
    │   │   └─ 0.021 SyncFromBase
    │   ├─ 2044.135 Lagrange
    │   │   ├─ 1466.124 RK2AvgIntegrator::TimeStep
    │   │   │   ├─ 9.793 hs::operator=
    │   │   │   │   └─ 0.018 SyncFromBase
    │   │   │   ├─ 1167.911 Eval_dv_dt
    │   │   │   │   ├─ 642.754 M_solver
    │   │   │   │   └─ 450.398 BatchComputeCornerForces
    │   │   │   │       ├─ 158.572 HydroStatePU::CalcHyperLaplacian
    │   │   │   │       ├─ 19.464 Setup
    │   │   │   │       ├─ 67.250 ComputeHydroBatchData
    │   │   │   │       ├─ 24.747 ComputeMaterialProperties
    │   │   │   │       ├─ 147.656 ComputeStress
    │   │   │   │       └─ 32.608 AssembleForceMatrix
    │   │   │   ├─ 11.997 Eval_di_dt
    │   │   │   ├─ 44.200 AddIndicators
    │   │   │   ├─ 207.708 Eval_de_dt
    │   │   │   ├─ 0.460 Eval_eos_rate
    │   │   │   ├─ 0.128 PostProcessEOSRate
    │   │   │   ├─ 17.777 Eval_ds_dt
    │   │   │   ├─ 1.965 ApplyRadialReturn
    │   │   │   └─ 0.002 SyncToBase
    │   │   ├─ 566.421 HydroStatePU::DeltaTEstimate
    │   │   │   └─ 418.542 BatchComputeCornerForces
    │   │   │       ├─ 136.818 HydroStatePU::CalcHyperLaplacian
    │   │   │       ├─ 18.703 Setup
    │   │   │       ├─ 64.759 ComputeHydroBatchData
    │   │   │       ├─ 23.850 ComputeMaterialProperties
    │   │   │       ├─ 142.759 ComputeStress
    │   │   │       └─ 31.557 AssembleForceMatrix
    │   │   └─ 0.239 hs::operator=
    │   │       └─ 0.001 SyncFromBase
    │   ├─ 0.001 tracer::updateAndBrodcastPositions
    │   ├─ 1.148 PeriodicALEManager::mesh_optimize
    │   ├─ 1405.252 Remap
    │   │   ├─ 6.170 InitRemapState
    │   │   ├─ 16.498 CalcDT
    │   │   ├─ 0.049 calc_disp_field
    │   │   ├─ 1376.134 EulerStep
    │   │   │   ├─ 14.916 mult_velocity
    │   │   │   └─ 59.585 CvTraceAssemble
    │   │   └─ 3.157 UpdateState
    │   │       ├─ 1.213 ALEUpdate
    │   │       │   ├─ 0.515 HydroInitState::CalcIndRhoDetJ
    │   │       │   ├─ 0.208 HydroInitState::MassMoments
    │   │       │   └─ 0.307 HydroInitState::AssembleMe
    │   │       └─ 0.000 SyncToBase
    │   ├─ 0.010 ApplyRadialReturn
    │   └─ 0.067 tracer::trackPoints
    │       └─ 0.067 tracer::PointInCell
    ├─ 0.007 carter
    ├─ 0.004 tracer
    └─ 60.038 evaluateEvents
        ├─ 45.051 curve_data:update
        │   └─ 0.029 tracer::probeParticle
        └─ 14.931 checkpoint
```

Figure 3.30: Hatchet: Three-way chart



Filtered GPU speedup regions (CPU / GPU > 1)

```
1.567 main
├─ 82.323 AssembleForceMatrix
├─ 1.005 tracer::probeParticle
└─ 1.687 timeStepLoop
   ├─ 2.384 blast
   │  ├─ 17.200 hs::operator=
   │  │  └─ 3.602 SyncFromBase
   │  ├─ 3.859 Lagrange
   │  │  ├─ 3.787 RK2AvgIntegrator::TimeStep
   │  │  │  ├─ 224.016 hs::operator=
   │  │  │  │  └─ 5.275 SyncFromBase
   │  │  │  ├─ 4.619 Eval_dv_dt
   │  │  │  │  ├─ 5.476 M_solver
   │  │  │  │  └─ 3.804 BatchComputeCornerForces
   │  │  │  │     ├─ 2.866 HydroStatePU::CalcHyperLaplacian
   │  │  │  │     ├─ 130.442 Setup
   │  │  │  │     ├─ 3.342 ComputeHydroBatchData
   │  │  │  │     ├─ 1.619 ComputeMaterialProperties
   │  │  │  │     ├─ 5.458 ComputeStress
   │  │  │  │     └─ 86.416 AssembleForceMatrix
   │  │  │  ├─ 2.442 Eval_di_dt
   │  │  │  ├─ 59.784 AddIndicators
   │  │  │  ├─ 2.440 Eval_de_dt
   │  │  │  └─ 12.826 Eval_ds_dt
   │  │  ├─ 4.381 HydroStatePU::DeltaTEstimate
   │  │  │  └─ 3.933 BatchComputeCornerForces
   │  │  │     ├─ 2.970 HydroStatePU::CalcHyperLaplacian
   │  │  │     ├─ 128.472 Setup
   │  │  │     ├─ 3.382 ComputeHydroBatchData
   │  │  │     ├─ 1.619 ComputeMaterialProperties
   │  │  │     ├─ 5.517 ComputeStress
   │  │  │     └─ 86.541 AssembleForceMatrix
   │  │  └─ 199.148 hs::operator=
   │  │     └─ 5.614 SyncFromBase
   │  └─ 1.551 Remap
   │     ├─ 6.941 calc_disp_field
   │     ├─ 1.637 EulerStep
   │     │  ├─ 4.462 mult_velocity
   │     │  └─ 16.610 CvTraceAssemble
   │     └─ 2.157 HydroInitState::CalcIndRhoDetJ
   └─ 1.070 tracer::probeParticle
```

Filtered GPU slowdown regions (GPU / CPU > 1)

(Note: Reciprocal factors w.r.t. speedup!)

```
36.596 ApplyRadialReturn
1.361 CalcDT
1.297 Eval_eos_rate
5.325 InitRemapState
1.981 PeriodicALEManager::mesh_optimize
15.055 PostProcessEOSRate
192.336 SyncToBase

3.170 UpdateState
├─ 3.786 ALEUpdate
│  ├─ 10.932 HydroInitState::MassMoments
│  └─ 6.379 HydroInitState::AssembleMe
└─ 273.306 SyncToBase

1.007 carter
1.407 checkUserInputInterrupt
19.984 evaluateEvents
├─ 9.692 checkpoint
├─ 8.703 curve_data:update
├─ 16.798 checkpoint
└─ 8.534 curve_data:update

5.440 initialize
├─ 5.515 blast
│  ├─ 1.208 MeshManipulation
│  ├─ 1.212 HydroInitState::CalcIndRhoDetJ
│  ├─ 11.792 HydroInitState::MassMoments
│  ├─ 7.101 HydroInitState::AssembleMe
│  ├─ 2537.636 SyncToBase
│  └─ 4.559 HydroStatePU::DeltaTEstimate
│     └─ 4.370 BatchComputeCornerForces
│        ├─ 4.683 HydroStatePU::CalcHyperLaplacian
│        ├─ 1.980 Setup
│        ├─ 2.514 ComputeHydroBatchData
│        ├─ 7.459 ComputeMaterialProperties
│        └─ 6.398 ComputeStress
├─ 1.500 carter
└─ 8.083 tracer
   └─ 10.323 tracer::trackPoints
      └─ 10.346 tracer::PointInCell
1.220 tracer
9.345 tracer::trackPoints
└─ 9.385 tracer::PointInCell
1.026 tracer::updateAndBrodcastPositions
```

Figure 3.31: Hatchet: Speedup filtered

memory was transferred to the appropriate space. The process was very much iterative and took roughly a year before a breakeven point was reached, here breakeven refers to running the same problem on a node of Sierra and a CTS-1 cluster and achieving the same run-time. Further performance gains would require advanced profiling tools, improved memory management strategies, and leveraging key programming features.

To further improve on performance, NVIDIA profiling tools were heavily used to track down hotspots. It was found that device memory allocation cost were fairly expensive compared to kernel execution time. To combat the high cost of memory allocation, MFEM's memory manager was enhanced with the Umpire resource manager enabling the construction of memory pools. Memory pools provide the capability to swiftly draw and release from a preallocated memory pool enabling different packages to share device memory. To improve kernel performance on GPUs it became necessary leverage the CUDA memory and threading hierarchy. Albeit, RAJA exposes more advanced threading capabilities through RAJA::kernel it was found that the native execution policies would become too complex for kernels found in MARBL. Through a co-design effort, between MFEM, RAJA, and MARBL teams, the RAJA teams API was introduced which simplified complex loop pattered as found in high-order finite element applications. Furthermore, RAJA teams enables run-time host/device policy selection, and kernel environment which resembles nested for loops.

With improved technology, the MARBL project started to realize greater than 10× run-time improvements when comparing to running on the same number of nodes of a CTS-1 cluster. Integration of additional GPU accelerated MFEM routines further simplified the amount of custom code in BLAST and further improved performance. Figure 3.32 highlights a number of notable events during the porting effort.

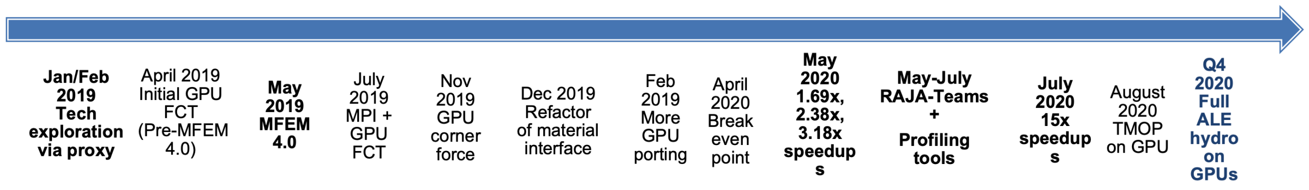

Jan/Feb 2019 Tech exploration via proxy
April 2019 Initial GPU FCT (Pre-MFEM 4.0)
May 2019 MFEM 4.0
July 2019 MPI + GPU FCT
Nov 2019 GPU corner force
Dec 2019 Refactor of material interface
Feb 2019 More GPU porting
April 2020 Break even point
May 2020 1.69x, 2.38x, 3.18x speedups
May-July RAJA-Teams + Profiling tools
July 2020 15x speedups
August 2020 TMOP on GPU
Q4 2020 Full ALE hydro on GPUs

Figure 3.32: Transformation timeline for MARBL-lag using RAJA.

MARBL-eul

The porting process for MARBL-eul (Miranda) began as an exploration study by a summer student in June of 2018. The primary options for computing on the GPU at that time were direct CUDA-Fortran and OpenMP with offload. The student focused on only porting the compute intensive portions of the hydro algorithm, namely the compact finite difference operator and its accompanying pentadiagonal linear solver. That work found that although the direct CUDA approach was slightly more performant, that the code would have to be forked and two versions of all the kernels would have to be created. OpenMP offered infinity better portability at the slight cost of performance.

However, adopting OpenMP was not quite as trivial as we had hoped, and early studies indicated that the entire code base might have to be rewritten in order to get performance. As was learned early on and as was discussed in Section 3.7.2, OpenMP is not performant on loops expressed using array syntax, or implicit Fortran loops. Exploration, therefore, was needed in creating and implementing a solution that enabled better developer productivity and performance on Sierra and on CTS1 like machines that use CPUs. Figure 3.33 shows the overall timeline for the past 2.5 years and the major stages of the exploration, development and porting, and evaluation process. Figure 3.34 shows specific details of the node:node speedup during the port, itself.

The majority of the node:node speed-up on the Fortran hydro (MARBL-eul) was achieved over a relatively short period of time. In part, this was enabled by small amount of code changes required by the FEXL DSL and the fact that the majority of the source code was not required to be changed. Furthermore, the code base is not massive with approximately 250 loops that were transformed in a matter of weeks.

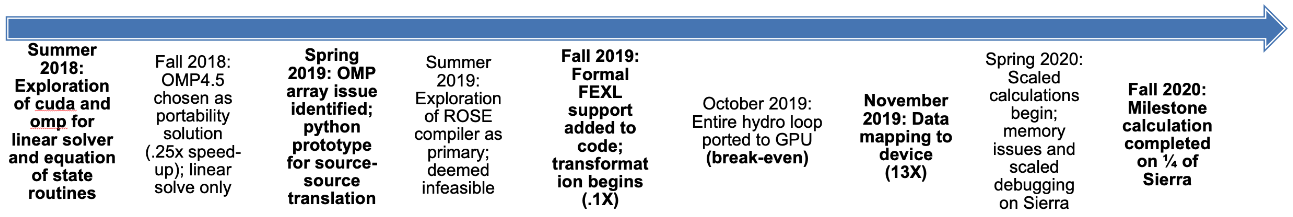

Summer 2018: Exploration of cuda and omp for linear solver and equation of state routines
Fall 2018: OMP4.5 chosen as portability solution (.25x speed-up); linear solve only
Spring 2019: OMP array issue identified; python prototype for source-source translation
Summer 2019: Exploration of ROSE compiler as primary; deemed infeasible
Fall 2019: Formal FEXL support added to code; transformation begins (.1X)
October 2019: Entire hydro loop ported to GPU (break-even)
November 2019: Data mapping to device (13X)
Spring 2020: Scaled calculations begin; memory issues and scaled debugging on Sierra
Fall 2020: Milestone calculation completed on ¼ of Sierra

Figure 3.33: Transformation timeline for MARBL-eul using FEXL. Preliminary exploration began over two years ago

Figure 3.34 shows the node:node speedup (Sierra:CTS1) using one node of each during the code porting process. This plots focuses on the actual implementation of the FEXL and OpenMP code into the core hydrodynamics solver. The stages of transformation that were noteworthy were:

- 0.1X speedup - Initial kernels decorated with FEXL for GPU offload
- 1.9X speedup - Explicit memory mapping of data to/from the GPU
- 6.0X speedup - Kernel launch overhear amortized by larger DOF/GPU
- 7.0X speedup - Optimized data motion, reduced call stack overhead and spurious memory motion of array meta-data
- 13.X speedup - Reduced the memory motion to/from the host by limiting to IO and buffer only

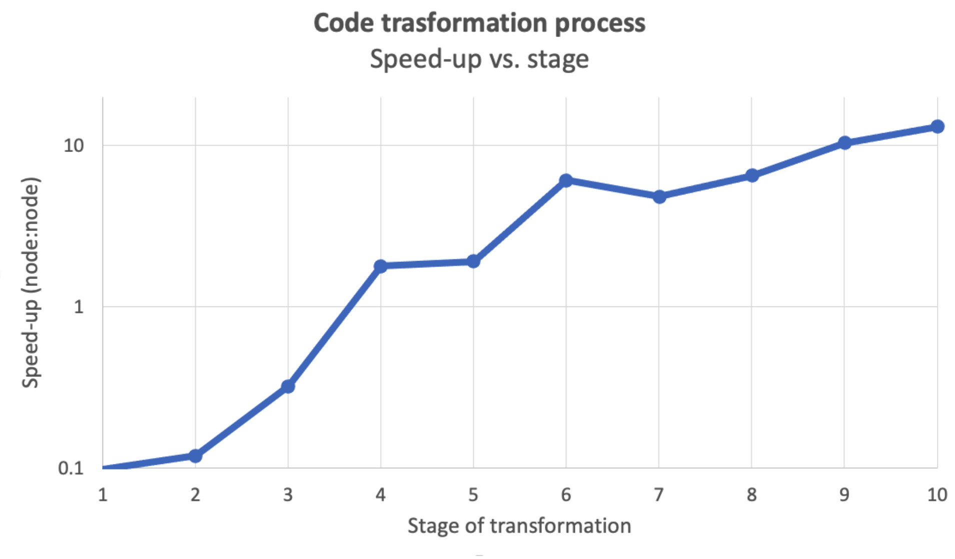


Figure 3.34: Node:node speed-up during the code porting process in MARBL-eul.

3.8 Node-to-node scaling study

This section describes a node-to-node scaling study comparing the code's performance on Commodity Technology Systems (CTS), which we refer to as CTS-1 and on Advanced Technology Systems (ATS). For ATS, our study uses Sierra, a GPU-based system at LLNL and Astra, an ARM based system at Sandia.

Sierra. Sierra is LLNL's 125 petaflop ATS system. It has a Mellanox EDR InfiniBand interconnect and 4,320 compute nodes, each of which has two sockets containing 20core POWER9 processors, 4 NVIDIA Volta V100 GPUs, and 256 GB of memory. Our scaling studies used up to 2,048 nodes, comprising half of the machine.

Astra. Astra is a 2.3 petaflop system deployed under the Sandia Vanguard program. It has a Mellanox EDR InfiniBand interconnect and is composed of 2,592 compute nodes, of which 2520 are user accessible. Each node contains two sockets containing 28-core Cavium ThunderX2 64-bit Arm-v8 processors and 128 GB of memory. Our scaling studies used up to the full machine: 2,496 nodes.

CTS-1. LLNL's CTS-1 clusters are commodity systems with Intel OmniPath interconnect, dual 18-core Intel Xeon E5-2695 2.1GHz processors and 128 gigabytes of memory per node. The system we tested on has 1,302 compute nodes and a peak of 3.2 petaflops. Our scaling studies used up to 256 nodes.

3.8.1 Node-to-node scaling

Since we are attempting to characterize and compare performance of our code across different computer architectures, the unit of comparison is an entire compute node of a given architecture. This differs from typical scaling studies which measure the baseline performance on a single processor. Node-to-node scaling studies allow natural comparison between a code's performance on different architectures and its granularity matches how our users typically view their allocations.

Specifically, for our studies, we utilize the following:

- **Sierra:** Each node has 4 GPUs. For our scaling studies, we ran with 4 ranks per node, with 1 GPU per rank.
- **Astra:** Each node has 56 cores. For our Lagrangian scaling studies, we are using OpenMP, with 2 ranks per node and 28 threads per rank. For our Eulerian scaling studies, we are using 32 ranks per node.
- **CTS-1:** Each node has 36 cores. We use 36 ranks per node for our Lagrangian scaling studies and 32 ranks per node for our Eulerian scaling studies. In both cases, we use 1 CPU core per rank on CTS-1.

Our study includes three types of comparisons:

Strong scaling Fixed *global* problem size (and polynomial order); vary the number of compute nodes.

Weak scaling Fixed *local* problem size (and polynomial order) per compute node; vary the number of compute nodes.

Throughput scaling For a fixed compute resource (1 compute node), we vary the problem size (and polynomial order).

Strong scaling demonstrates the ability of a code to solve a fixed problem faster as one adds additional compute resources, while weak scaling demonstrates the ability of a code to solve a larger problem faster as one adds more resources. In contrast, throughput scaling characterizes the machine performance and demonstrates the balance between parallelism and memory capacity; that is, it measures the work required to saturate memory bandwidth on the compute resources. Commodity systems have modest memory bandwidth and compute resources. As such, they can only accommodate a relatively small amount of parallelism. They are easy to saturate, yielding overall lower performance. On the other hand, GPU-based systems, like Sierra, have very high memory bandwidth and compute resources. As such, they require large amounts of parallel work to saturate, but can yield very high performance. Figure 3.35 illustrates what we are measuring with a throughput graph, and characterizes how performance improvements can be achiieved by alleviating performance bottlenecks and/or exposing additional parallelism.

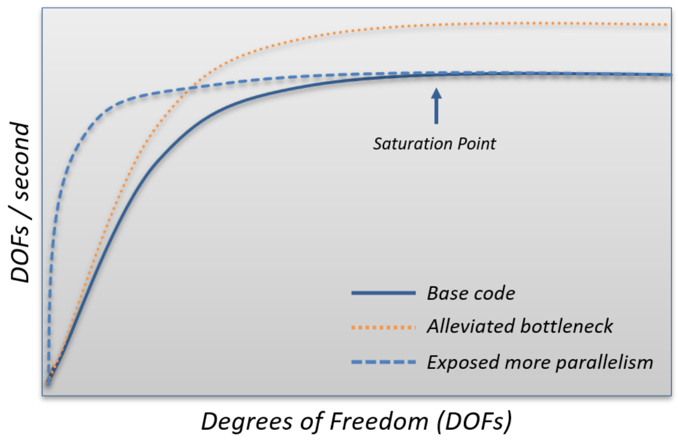


Figure 3.35: Understanding throughput scaling. The independent variable (x-axis) is measured in the number of Degrees of Freedom (DOFs) in the problem. The dependent variable is throughput: the number of DOFs processed per unit of time (higher is better). As the number of DOFs increases, performance improves until throughput has been saturated. One can increase throughput by alleviating a performance bottleneck (orange curve) or by exposing more parallelism (dashed blue curve).

3.8.2 Scaling study problem definition

We used the Triple-Point 3D problem for our scaling studies. In an effort to normalize the total amount of work, we ran 500 hydro cycles in each run. For our Lagrange runs (Blast), we applied an ALE remap every 50 cycles, ran the problem on a unstructured hexahedral NURBS mesh containing 1,764 elements in the base mesh. For strong and weak-scaling, we used uniform refinement to generate larger problem sizes. This axisymmetric mesh was generated by PMesh from a revolved 2D base mesh of 28×12 elements, see Figure 3.36(a). Each level of uniform refinement multiplies the number of elements by a factor of 8. For our Eulerian runs (Miranda), we ran on a $28 \times 12 \times 12$ Cartesian base mesh with 4,032 grid points.

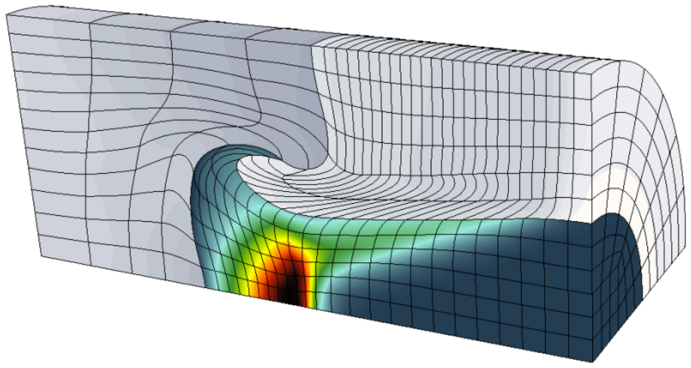
We note that, for high order Finite Element codes, the unit of work is at the quadrature point, rather than the element, as it would be in lower order codes (c.f. Section 2.2.1). For our runs, there were: 8 quadrature points (qpts) per linear element, 64 qpts per hex element and 216 qpts per cubic element. Figure 3.36(b) lists the number of elements and quadrature points in our mesh for quadratic runs at different uniform levels of refinement.

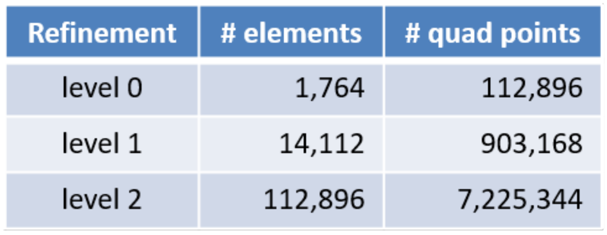
Study design (details)

3.8.3 Analyzing node-to-node Lagrangian scaling performance

Lagrangian strong scaling. Our strong scaling study was run at uniform refinement level 2, with a total of 112,896 elements and 7,225,344 quadrature points distributed among varying numbers of compute nodes. Our study starts with four compute nodes and iteratively doubles the node count in successive runs. For ideal strong scaling, the time per cycle would decrease linearly as we increase the node count. Specifically, as we double the node count, the time per cycle would reduce by a factor of two.

Figure 3.37 depicts the code's node-to-node strong scaling performance on Sierra (red), Astra (green) and CTS-1 (black). As can be seen in the figure, MARBL exhibits very good strong scaling on the CPU-based architectures up to 256 CTS-1 nodes and 512 astra nodes, when communication costs begin to dominate the runtime. As expected, the code exhibits worse strong scaling for higher node counts on the GPU, where as the node count increases, there is no longer sufficient work to feed each GPU. As such, for this problem, the strong scaling begins to level off when using more than 32 nodes (128 GPUs).


(a) Triple Pt 3D NURBS mesh for Lagrangian studies



Refinement	# elements	# quad points
level 0	1,764	112,896
level 1	14,112	903,168
level 2	112,896	7,225,344

(b) Triple Pt 3D problem sizes on quadratic meshes

Figure 3.36: Problem setup for Lagrangian scaling study. (a) Our Triple-Point-3D base mesh is defined by 1,764 NURBS elements. (b) A table listing the number of elements and quadrature points for quadratic meshes at different levels of uniform refinement.

Lagrangian weak scaling. Our weak scaling study was performed on the Triple-Pt-3D problem starting with 2 levels of uniform refinement for our first data point. Each subsequent run in the series maintained the same average number of quadrature points per rank by applying an extra level of uniform refinement (multiplying the global problem size by a factor of eight, see Figure 3.37(b)) and increasing the node count by a factor of 8. As such, we have up to four runs in each of our weak scaling series, using 4, 32, 256 and 2048 nodes. We note that 2048 nodes is half of Sierra and 80% of Astra.

Figure 3.38 depicts the code's weak scaling performance on Sierra (red), Astra (green) and CTS-1 (black). Since each rank has the same local problem size, ideal weak scaling would maintain the same compute time as we scale out to more nodes. This is not generally achievable due to the increased inter-node communication as we scale out to more nodes. In addition to plotting the compute time per cycle, we also annotate each data point with the weak scaling efficiency, measured as the time per cycle of the current data point divided by the time per cycle for the first data point the series.

As can be seen in the figure, our code achieves good weak scaling performance as we scale out to the maximum number of nodes in the study. Comparing Sierra performance to the other platforms, we are seeing about 15× speedup node-to-node for Sierra compared to CTS-1 systems, and about 30× or higher speedups compared to Astra.

Lagrangian throughput. Our throughput studies were run on a single compute node of each system. In order to increase the number of datapoints, we used a different type of uniform refinement. Rather than using successive refinement by powers of two, we refined each element in the mesh into an $n \times n \times n$ subgrid of elements. For example, our third datapoint refines each original element into 27 elements.

As can be seen in Figure 3.39, CPU-based platforms like CTS-1 and Astra saturate the throughput at relatively small problem sizes, where increasing the problem size scales out the time to solution linearly. They also achieve the same throughput for different polynomial orders. In contrast, throughput on GPU-based systems, like Sierra, increases with problem size and polynomial order. In fact, as we increased the problem size, we ran out of memory before we were able to saturate the throughput on a single node. We also observe that our throughput increases for higher orders with respect to the throughput for linear meshes.

Interestingly, we observed higher throughput for quadratic runs than for cubic runs on Sierra, We believe this is related with bottlenecks in our current implementation and are looking into ways to resolve this. In particular, parts of our ALE remap algorithm are still using full assembly (FA) over sparse matrices. This is evident, for example, in the relative throughput differences between quadratic and cubic runs in for the 'Lagrange' and 'Remap' phases of our simulation (compare Figures3.39(b) and 3.39(c)). We are investigating ways to convert this to a partial assembly (PA) algorithm.

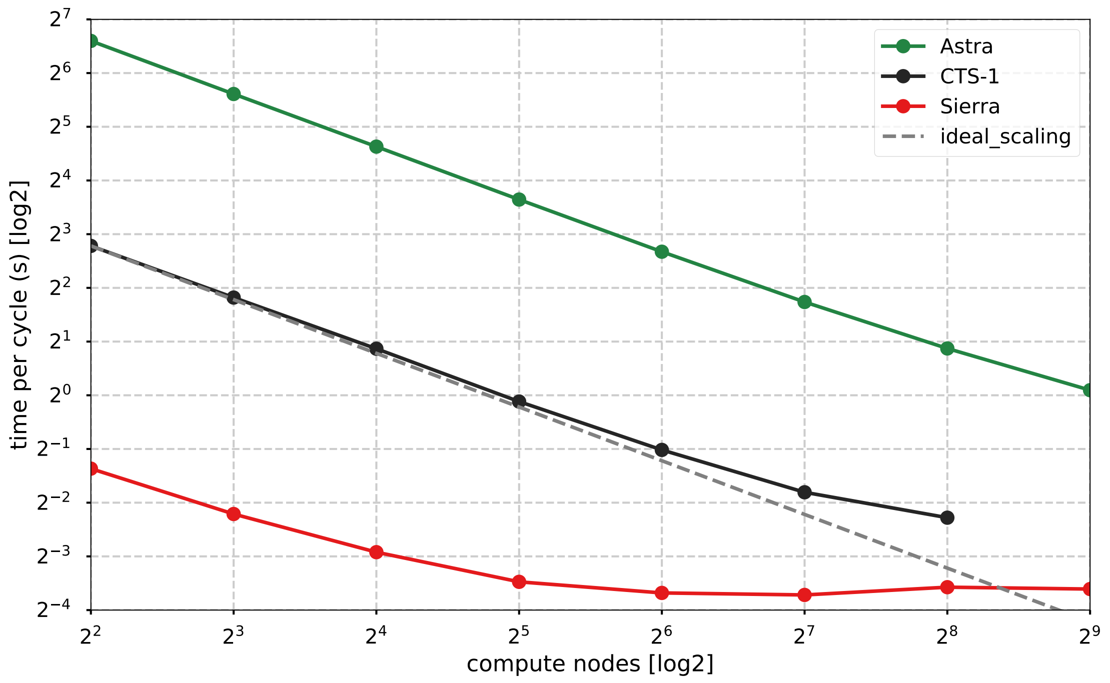


Figure 3.37: Node-to-node strong scaling for Lagrangian Triple-Pt-3D problem with two levels of uniform refinement. Each data point represents a 500 cycle Lagrange hydro simulation, with an ALE remap every 50 cycles. Data is plotted on a log2-log2 scale where the independent axis measure the number of compute nodes while the dependent axis measures the time per cycle. We also plot ideal strong scaling performance for the CTS-1 series: a line with slope -1. Data is from a strong scaling study performed on 26Oct2020.

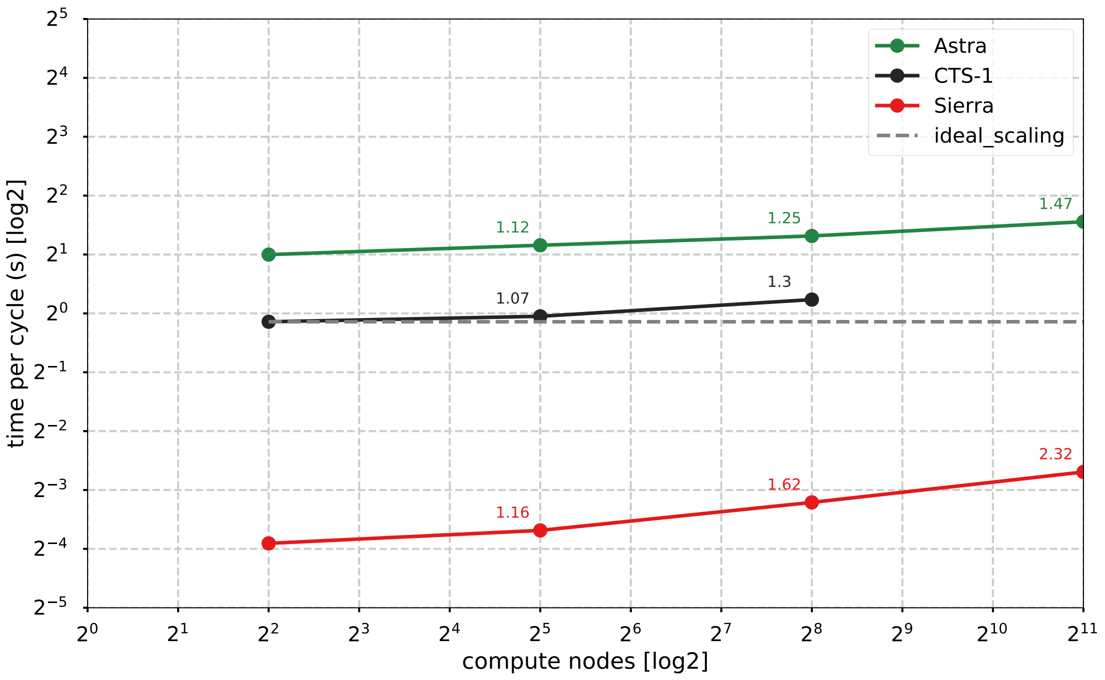
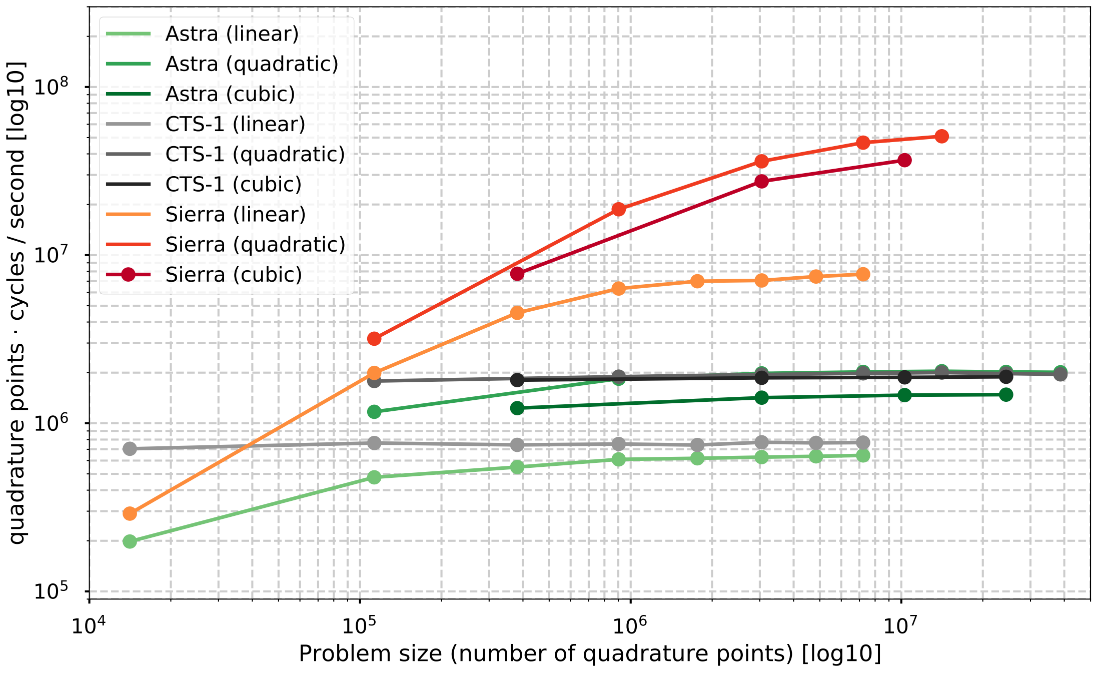
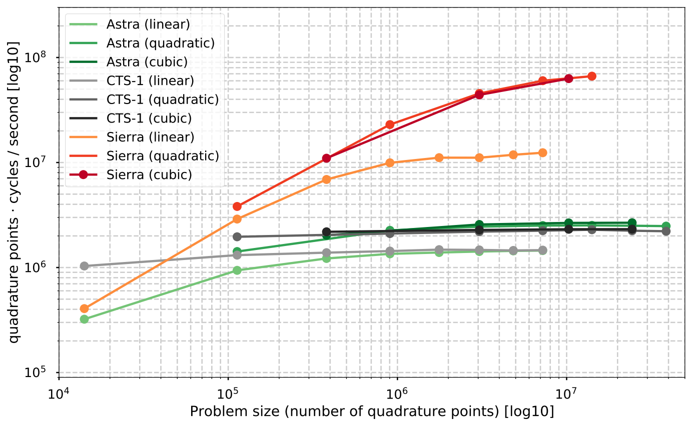


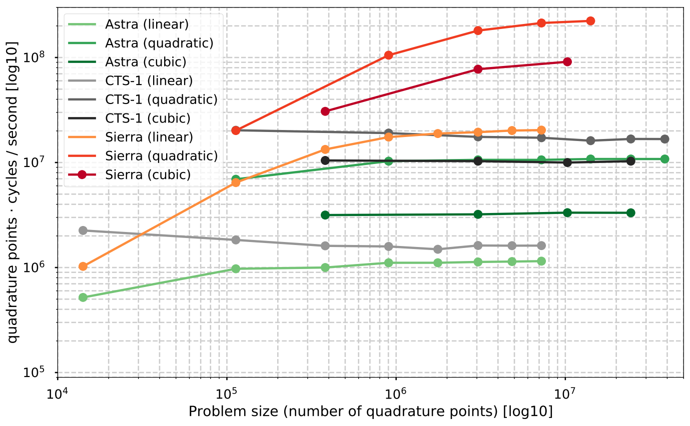
Figure 3.38: Node-to-node weak scaling for Lagrangian Triple-Pt-3D problem on 4, 32, 256 and 2048 compute nodes. Each data point represents a 500 cycle Lagrange hydro simulation with an ALE remap step every 50 cycles. Data is plotted on a log2-log2 scale where the independent axis is the number of compute nodes and the dependent axis is the time per cycle (in seconds). The annotations by each data point indicate the weak scaling efficiency with respect to the 4 node run on that platform. Ideal weak scaling would correspond to an efficiency of 1.0 and all data points in that series would lie on a horizontal line. Note that 2048 nodes is 50% of Sierra and 80% of Astra. Data is from a weak scaling study performed on 24Sep2020.



(a) Throughput for simulation 'timeStepLoop'



(b) Throughput for 'Lagrange' phase of simulation



(c) Throughput for 'Remap' phase of simulation

Figure 3.39: Node-to-node throughput study for Lagrangian Triple-Pt-3D problem at three different polynomial orders. (a) Overall throughput for simulation 'timeStepLoop' (ignoring initialization times) (b) throughput associated with the 'Lagrange' phase (c) throughput associated with the 'Remap' phases of our hydrodynamics cycles. We plot the problem size (total number of quadrature points) on the independent axis using a log10 scale and throughput, the number of quadrature points processed per unit of time, on the dependent axis using a log10 scale. Note that the CPU-based systems (Astra and CTS-1) reach their saturation point at relatively low problem sizes and maintain relatively stable throughput regardless of the polynomial order. In contrast, the GPU-based Sierra has a significantly higher throughput, especially for higher polynomial orders. Data is from a weak scaling study performed on 23Nov2020.

Scaling out to the whole system. Our final node-to-node study for ALE extends our analysis to the full Astra system (2,496 nodes out of the 2,520 user-accessible nodes) and to half of Sierra (2,048 compute nodes).

For this comparison, we devised a "strong-weak" analysis – a careful orchestration of runs that, when combined allow us to plot several strong-scaling and weak scaling studies on the same chart.

In particular, we have data points at all powers of two from 1 (2^0) to 2,048 (2^{11}), and for Astra, we also have runs with 39, 312 (i.e. $39 \cdot 8$) and 2,496 (i.e. $39 \cdot 8^2$) nodes. Figures 3.40(a) and 3.40(b) show the 'strong-weak' plots for Astra and Sierra, respectively. Note that the datapoint at the top right of Figures 3.40(a) uses all of Astra. In these plots, each vertical line with the same color represents a strong scaling series (they have the same global number of quadrature points in the problem) while the horizontal dashed lines represent weak scaling series (they each have the same average local number of quadrature points per rank). As in the earlier strong scaling plots, we are using a log2-log2 scale for the axes, so ideal strong scaling corresponds to a diagonal line of slope -1 while ideal weak scaling corresponds to a horizontal line.

Discussion. We note that we had trouble scaling out to all of Astra using just MPI due to difficulties distributing our mesh to more than 100,000 ranks using the Metis library. As such, we opted for an MPI+OpenMP configuration on Astra, with 2 MPI ranks per node and 28 OpenMP threads per rank. Our OpenMP configuration used a RAJA OpenMP execution policy, and we were able to adapt our RAJA-CUDA implementation to use RAJA-OpenMP in a single afternoon. Using MPI+OpenMP, as opposed to MPI-everywhere reduces our performance by about a factor of two in the initial 4-node run. However, (using low node codes) it weak scaled better than the MPI everywhere version.

3.8.4 Analyzing Eulerian node-to-node scaling performance

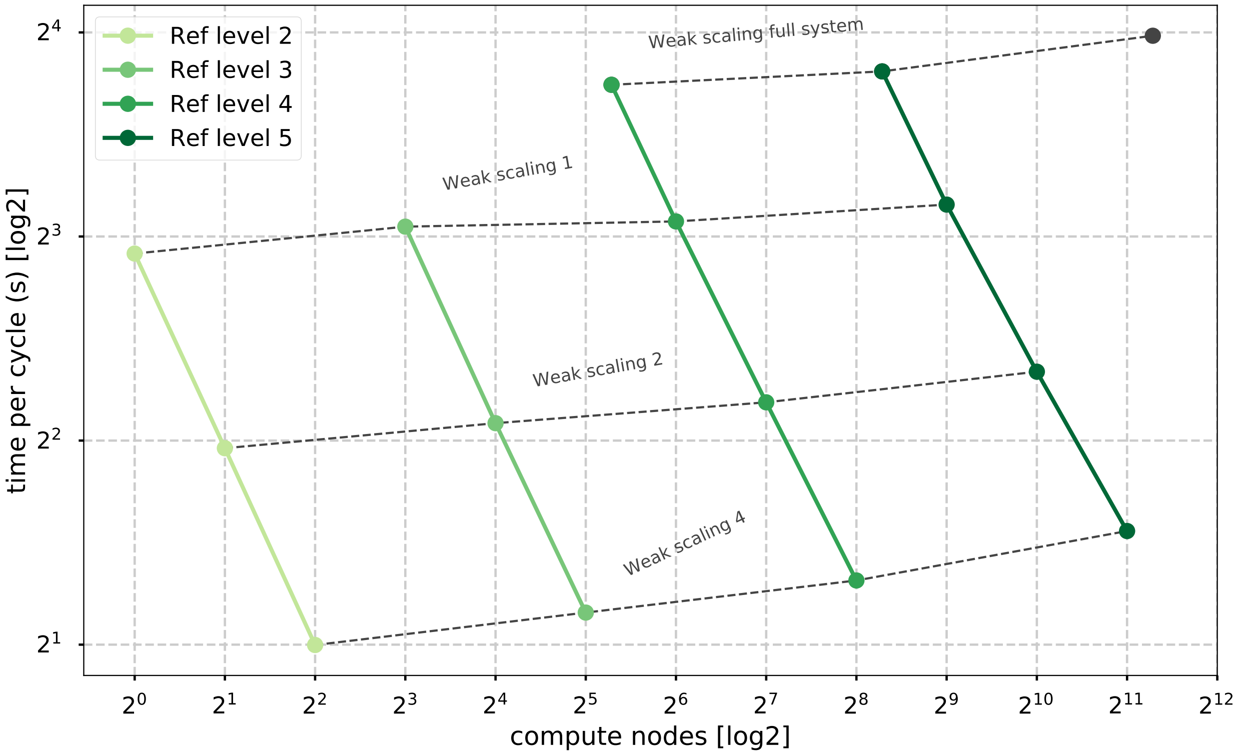
Eulerian strong scaling. Figure 3.41 shows the results of our Eulerian strong scaling study on Sierra (red), Astra (green) and CTS-1 (black). Similarly to the Lagrangian strong scaling, we see nice strong scaling performance for the CPU-based CTS-1 and Astra machines, with performance leveling off at around 128 nodes. The code also strong scales well up to 4 nodes, before leveling off at higher node counts due to insufficient degrees of freedom.

Eulerian weak scaling. Figure 3.42 shows the results of our Eulerian weak scaling study on Sierra (red), Astra (green) and CTS-1 (black) using up to 1,024 compute nodes. This comprises 25% of Sierra.

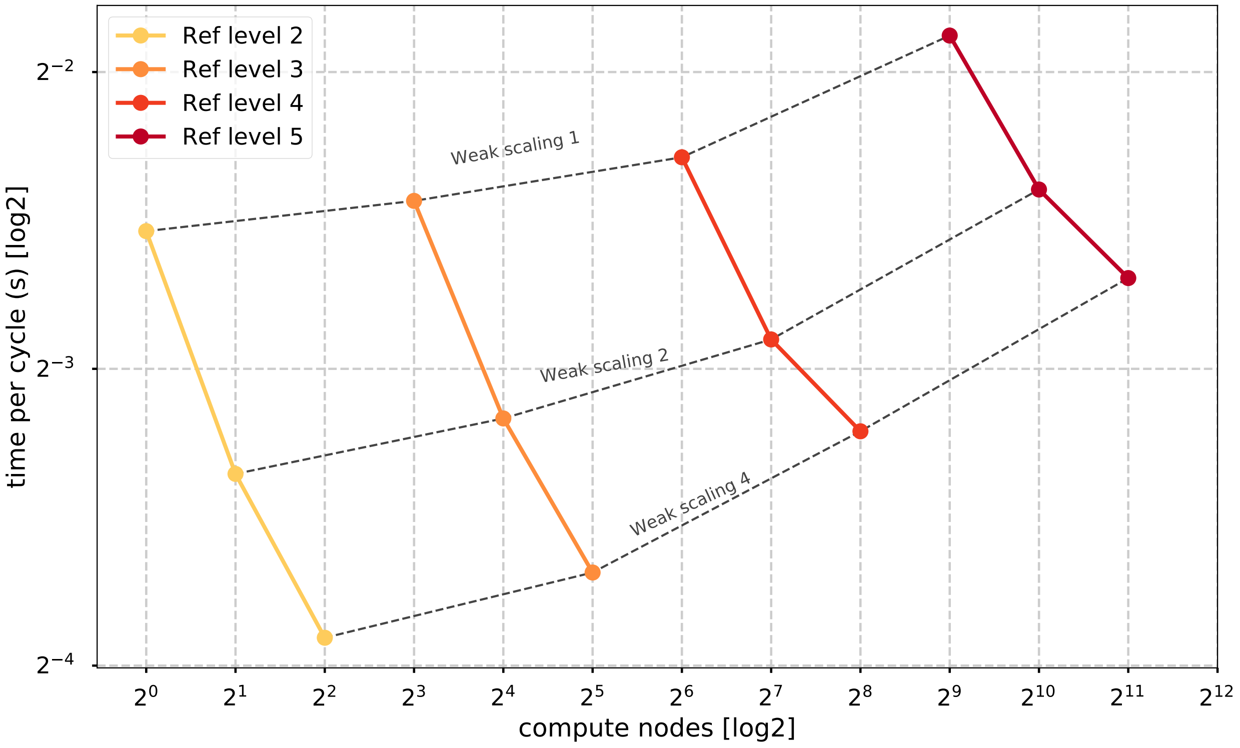
We add a brief discussion here of the poor weak scaling observed in the Sierra results of Figure 3.42. In section 4.3.1, we discuss the algorithmic costs associated with computing compact finite derivatives on a distributed system. Part of this algorithm requires the global communication via an `MPI_Allgather` to collect boundary data before the final linear system can be directly solved. The volume of this data exchange scales with the number of processors along a particular logical direction multiplied by the halo data width. As the over-all problem size grows and the number or ranks along a direction grows, once can see that the volume (and therefore costs) of this communication operator scales as $N^{1/3}$, where N is the total number of ranks. If we crudely approximate the time of one computational time step as the sum of the compute time and the data communication time we have:

$$T_{wall} = T_{comp} + T_{data}, \tag{3.4}$$

$$T_{data} = cN^{1/3}. \tag{3.5}$$



(a) Astra full system 'strong-weak' study



(b) Sierra half-system 'strong-weak' study

Figure 3.40: "Strong-weak" scaling plot for full system study on Astra (greens, top) and Sierra (reds, bottom) for the Lagrangian Triple-Pt-3D problem. This chart shows a combination of several scaling studies orchestrated to show strong and weak scaling for a given architecture on the same plot. Each colored diagonal lines is a strong scaling study, while the dotted lines represent weak scaling study. We note that the data point in the upper right corner of (a) utilizes all of Astra (2,496) nodes. Similarly, the strong scaling series (dark red) at refinement level 5 in (b) corresponds to 512, 1,024 and 2,048 compute nodes. That is, an eighth, a quarter and half of Sierra, respectively. The data is from a 'Strong-Weak' scaling study performed no 24Sep2020.

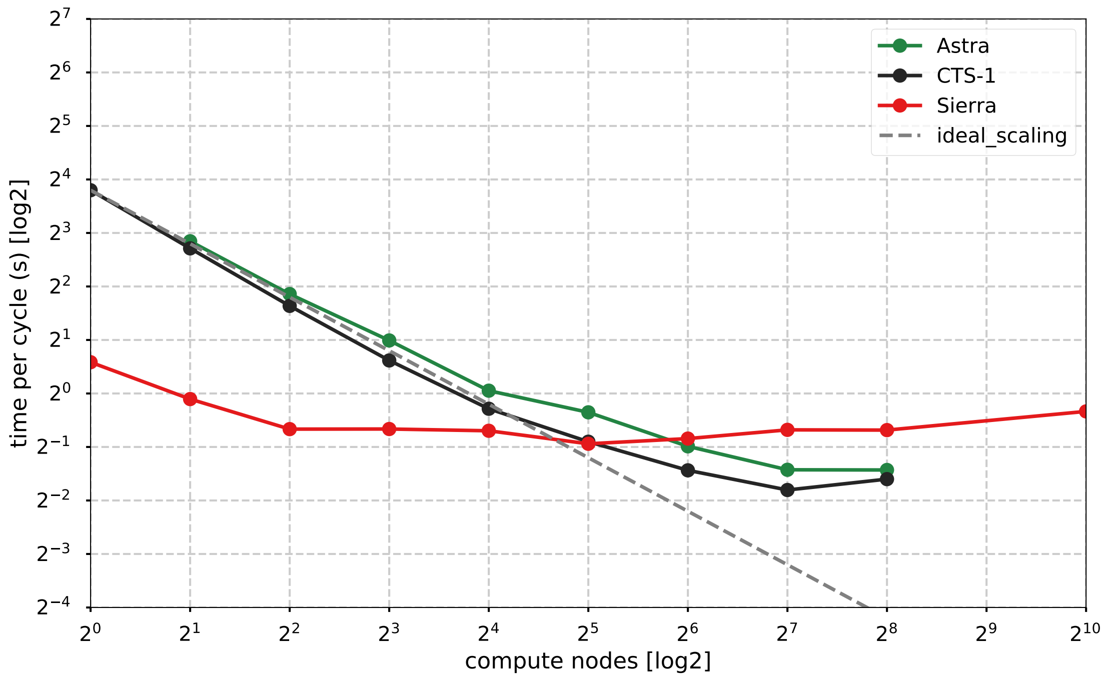


Figure 3.41: Node-to-node strong scaling for Eulerian Triple-Pt-3D problem with two levels of uniform refinement. Each data point represents a 500 cycle Eulerian hydro simulation. Data is plotted on a log2-log2 scale where the independent axis measure the number of compute nodes while the dependent axis measures the time per cycle. We also plot ideal strong scaling performance for the CTS-1 series: a line with slope -1. Data is from a strong scaling study performed on 12Nov2020.

We can then write an expression for the node to node speedup, give the performance model as:

$$S(N) = \frac{T_{wall,cpu}}{T_{wall,gpu}} \tag{3.6}$$

$$S(N) = S_0 \frac{\left(1 + \frac{1}{2S_0}\left(\frac{T_{data}}{T_{comp}}\right)\right)}{1 + \frac{T_{data}}{T_{comp}}} \tag{3.7}$$

where $S_0 = \frac{T_{comp,cpu}}{T_{comp,gpu}}$, which is the node to node speedup assuming data communication times is negligible.

A useful observation from this model is that it predicts that the speedup at one quarter of Sierra is $S(4096) = 3.93$, which is consistent with what was observed in Figure 3.43. Furthermore, one can compute the point at which the speedup hits the break-even point which we computed as $S(20,000) = 1.0$, which is approximately 117% the size of Sierra. This model does suggest, however, that scaling for very large scale problems may become an issue for future platforms.

We can address this lack of scaling with alternative finite-difference schemes that do not require a linear solve and therefore, eliminate the `MPI_Allgather` and alleviate the weak scaling issues. Figure 3.43 shows a comparison of the CTS-1 and Sierra scaling using a 10th order scheme and also shows results for a 6th order finite-difference scheme that shows there is no large penalty from the `MPI_Allgather`. In additional, the performance model is included for the CTS-1 and Sierra results.

Eulerian throughput. Figure 3.44 plots Eulerian throughput on Sierra (red), Astra (green) and CTS-1 (black) on a log-log plot. Similarly to the Lagrangian case, we observe that the CPU-based CTS-1 and Astra platforms saturate quickly on very small problem sizes, where performance scales linearly as the problem increases. In contrast, the throughput on GPU-based Sierra saturates at a much higher problem size with several million grid points per GPU.

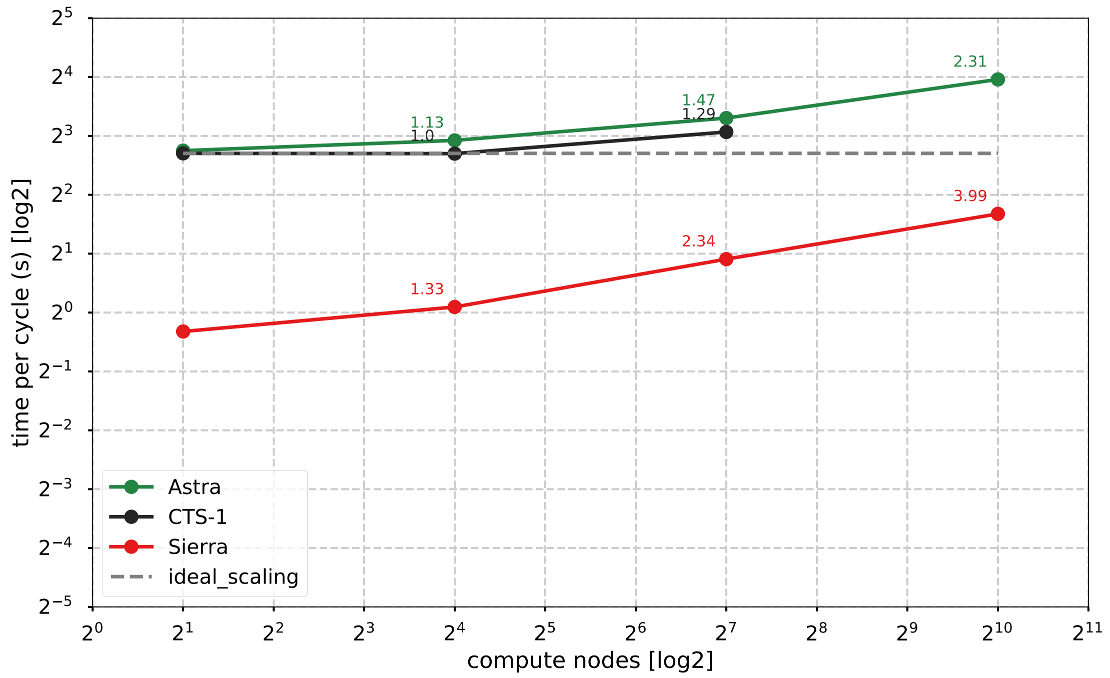


Figure 3.42: Node-to-node weak scaling for Eulerian Triple-Pt-3D problem. Annotated numbers indicate weak scaling efficiency with respect to the first run in the series (2 compute nodes). Data is plotted on a log2-log2 scale, where a horizontal line corresponds to ideal weak scaling. Study goes out to 1024 nodes, ¼ of Sierra. Data is from a weak scaling study performed on 12Nov2020.

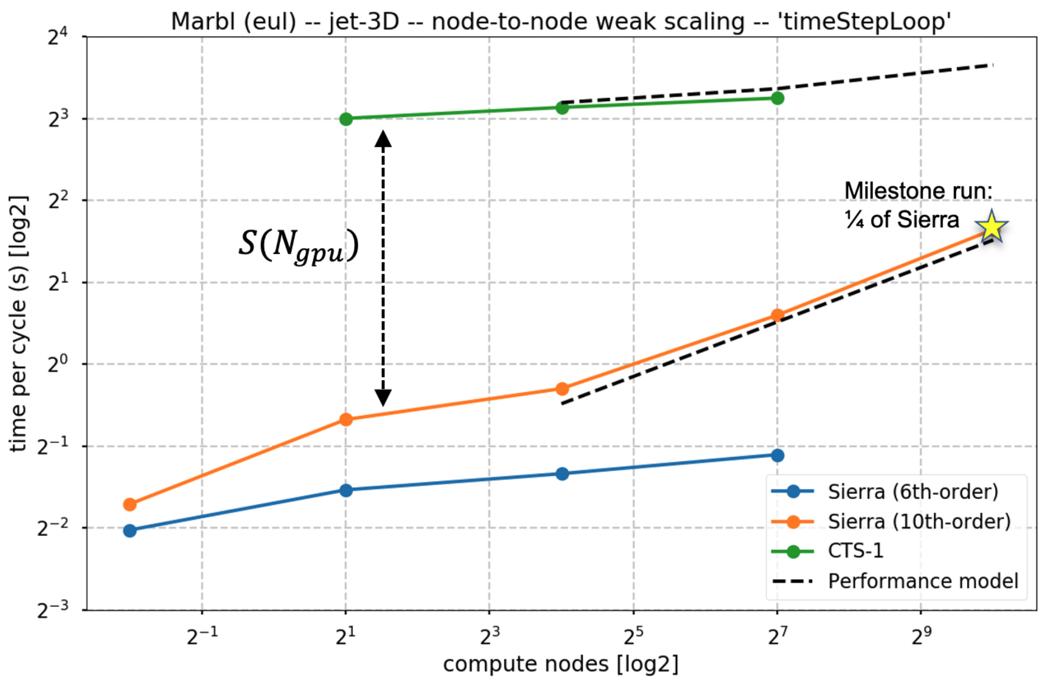


Figure 3.43: Node-to-node weak scaling for Eulerian two-species-3D jet problem with $0.5 \cdot 256^3$ DOFs per node. Data are plotted on a log2-log2 scale, where a horizontal line corresponds to ideal weak scaling. Study goes out to 1024 nodes, ¼ of Sierra.

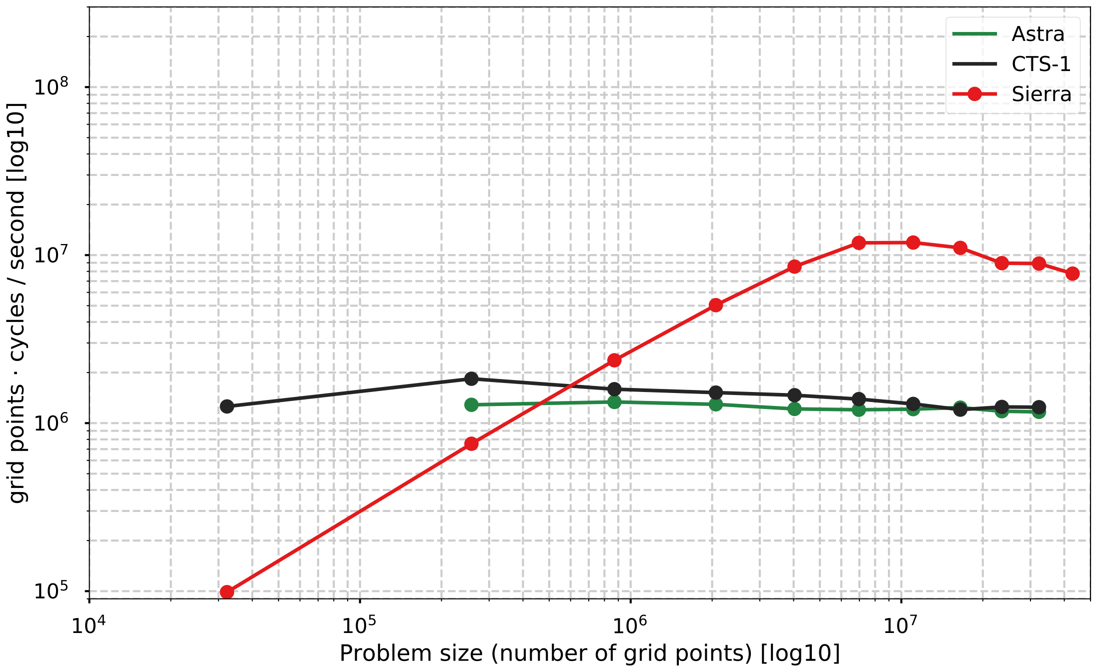


Figure 3.44: Node-to-node throughput study for Eulerian Triple-Pt-3D problem. We plot the problem size (total number of grid points) on the independent axis using a log10 scale and throughput, the number of grid points processed per unit of time, on the dependent axis using a log10 scale. Data is from an Eulerian throughput study performed on 12Nov2020.

3.9 Power Studies

To better understand power needs for large scale simulations, this section investigates energy usage of the Lagrangian hydrodynamics option. As we move to advanced architectures understanding power requirements becomes a general area of interest. A common surrogate for this is to use thermal design power (TDP) due to its simplicity and availability; however, TDP is not indicative of how much power a machine will use once it is actually running. Using the advance technology system, Sierra, we perform a series of experiments which suggest that large scale LLNL application codes are more efficient than a standard TDP model might suggest. Lastly, we present a cross platform comparison between Astra, Sierra, and a CTS-1 machine.

3.9.1 Background and Methodology

For many large scale LLNL application codes, running effectively on Sierra required refactoring key computational kernels and revisiting memory management strategies. For the Lagrangian hydrodynamics package, a full description of the tools lessons learned may be found in Sections 3.7.1, 3.7.3, 3.7.4, 3.7.5. In this work we investigate the power usage of the Lagrangian option in MARBL as kernels are ported and optimized. All experiments are carried out on rzAnsel, which has the same compute architecture as Sierra. Each node of rzAnsel consist of two IBM Power9 CPUs and 4 NVIDIA V100 GPUs. The TDP for each Power9 CPU is 250W and the TDP for each V100 is 300W. For Sierra, we use IBMs csm_allocation_query tool to monitor energy usage. This tool operates on a per job basis and supplies information about how much energy was consumed by the nodes in the application. It further includes energy consumed specifically by the GPU. Since it is a node level utility, it does not include energy consumed by the switches. In order to perform a higher-level study, more advanced tools and dedicated time on Sierra would be needed.

Duration (Seconds)	Job Time (seconds)	Total Energy Consumed (Joules)	GPU Energy (Joules)	Non-GPU Energy (Joules)
3600	3651	1520572	508197	1012375

3.9.2 Results

We begin with a simple baseline experiment to understand IBM's power management tools. For this, we launch a single node job on Sierra, which sleeps for an hour and we measure the results. The table below presents baseline energy consumed for the sleep command.

Using the nvidia-smi command, we can determine that an idle V100 GPU consumes roughly 35W of power, including overhead for starting and ending the job, consistent with our measurements above. The formula a single GPU's power (Watts) is given by

$$\text{Power (Watts)} = \frac{\text{GPU Energy (Joules)}}{\text{Job Time (Seconds)} \times 4(GPUs)}.$$

We also note that as a baseline, the non-GPU components of the job are responsible for the majority of the consumed energy.

A recent effort on the MARBL project has been the refactoring of the ALE hydrodynamics package to enable GPU acceleration. Evaluation of performance was largely done through a node to node comparison of different compute architectures. Shortly after reaching a breakeven point, equal run-time performance for the same problem on different architectures (CPU vs GPU), the MARBL team began monitoring how further porting and optimizations impacted power consumption on a node of rzAnsel. To monitor power changes during the optimization phase, the 3D Brl81a shaped charge problem was routinely exercised as code changes were made. The shaped charge problem was periodically run for 1000 cycles on a mesh containing 28020 elements. Power studies were run on a single node of rzAnsel using all four GPUs. Figures 3.45, and 3.46 illustrate changes in energy and total run-time as a function of time illustrating the impact of code optimizations. The general trend we see is that every decrease in runtime is accompanied by a decrease in energy usage. Lastly, Figure 3.47 illustrates that average GPU power consumption does not exceed 90 Watts, below a third of the 300W TDP.

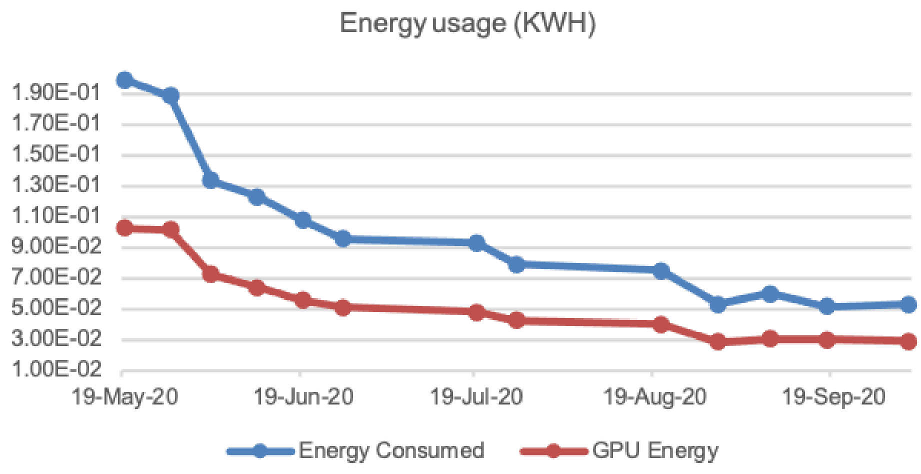


Figure 3.45: Energy usage of the Lagrangian package as kernels are ported. Lagrangian is used to simulate a 1000 cycles of the 3D Brl81a shaped charge problem on 4 GPUs. As more kernels are ported to GPUs energy usage decreases.

In collaboration with Livermore and Sandia computing divisions we performed a cross platform energy usage study using the 3D triple point problem and dedicated time on the machines. For meaningful measurements it was necessary to run on 360 nodes of Sierra, 62 nodes of a CTS machine, and 72 nodes of Astra on account of tool limitations. Additionally, the problem

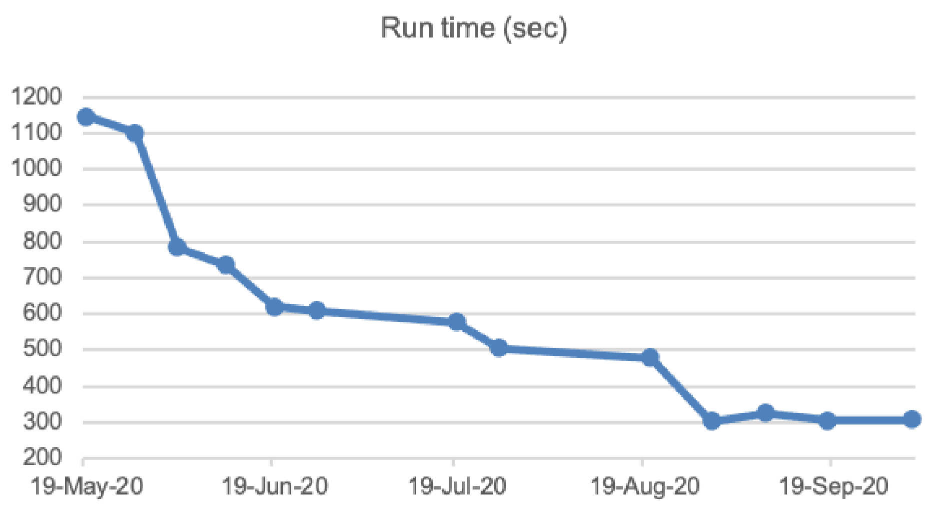


Figure 3.46: Run time of the Lagrangian package as kernels are ported. Lagrangian is used to simulate a 1000 cycles of the 3D Brl81a shaped charge problem on 4 GPUs. As more kernels are ported to GPUs run time decreases.

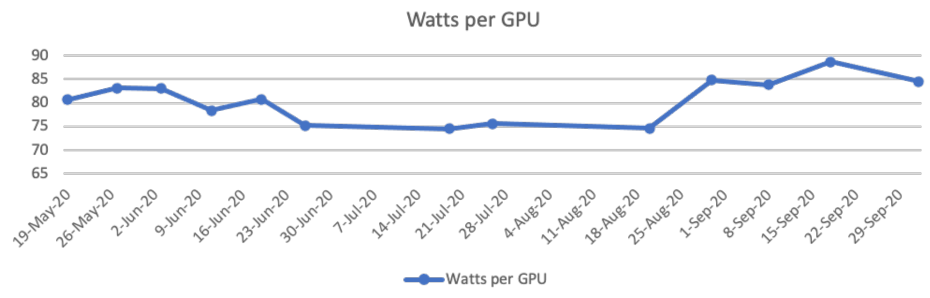
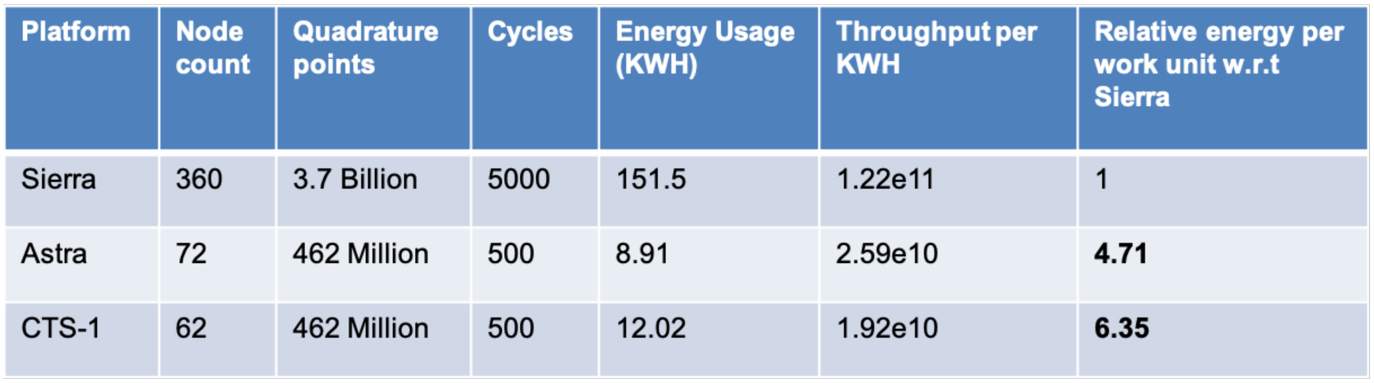


Figure 3.47: Watts used per GPU when solving for 1000 cycles of the 3D Brl81a shaped charge problem as the Lagrangian package is ported to GPUs. We observe that watts used remain below 90, less than a third of the TDP (350 Watts).

was required to be run for a sufficiently long time to reduce impact of start up and clean up cost when running a job. Given the different requirements for each platform we chose to run the same problem on the CTS and Astra machines, more specifically we solved on a mesh with 462 million quadrature points for 500 cycles. On the Sierra system we solved a problem with 3.7 billion and simulated out to 5000 cycles. Figure 3.48 presents a summary of our main findings, notably we find that the GPU efficiency enables higher throughput within a limited power budget. Our finding was derived by comparing throughput per kilowatt hour for each platform. Throughput per KWH is defined as the ratio of the product of number of elements, quadrature points, and cycles and KWH used.



Platform	Node count	Quadrature points	Cycles	Energy Usage (KWH)	Throughput per KWH	Relative energy per work unit w.r.t Sierra
Sierra	360	3.7 Billion	5000	151.5	1.22e11	1
Astra	72	462 Million	500	8.91	2.59e10	**4.71**
CTS-1	62	462 Million	500	12.02	1.92e10	**6.35**

Figure 3.48: Cross platform summary of power usage using the 3D triple point problem. To obtain meaningful measurements (due to tool limitations), the problem was run on 62 nodes of a CTS-1 cluster, 72 nodes of Astra, and 360 nodes of Sierra. The same size problem was run on Astra and Sierra while the problem was made larger for the Sierra system. Results show the Sierra is able to achieve a higher throughput compared both CTS-1 and Astra systems using the same energy.

Experiments with the MARBL code were limited to second order finite element schemes. As a next step we plan to leverage MARBL's high-order capabilities to investigate the relationship between compute intensity and power usage. Additionally, we are actively investigating the notion of the breakeven point to understand how efficiently GPUs must be utilized to deliver a clear win over CPU based architectures in terms of both speed ups and power efficiency.

3.10 Problem Setup

3.10.1 IREP and well-known tables

This subsection describes the *Intermediate Representation* (IREP)[81] library used by MAPP codes.

IREP is a small (about 700 LOC) Lua-based C and Fortran library. In operation, it reads program input in the form of Lua tables, and places the input values into compiled, structured variables, which can then be read from either C++ or Fortran. The data is "plain old data": scalar or one-dimensional arrays of integers, doubles, boolean, or string values. It can also define Lua callback functions, that can be accessed and executed at arbitrary later times from the compiled code. The compiled data store meets the constraints of the Fortran ISO C binding[82], and thus is shareable between Fortran, C, and C++.

IREP is designed to handle the initial problem setup task, in a way that is the same for both C++ or Fortran. It takes advantage of a simple correspondence between Lua *tables*, C *structs*, and Fortran *derived types*: (See Figure 3.49)

As seen in the figure, the IREP definition is written using a set of C preprocessor macros. That definition is then translated into C or Fortran, or into a symbol table that can be used as a template to parse the eventual Lua input table. The MAPP codes have defined an IREP datastore that contains more than 600 variables, along with about 100 Lua callback functions in about 10 "well-known tables".

```
Lua             C/C++               Fortran
---             -----               -------
t = {           struct irt_t {      type, bind(c) :: irt_t
  a = 3,          int a;              integer(c_int) :: a=3
  b = 7.2,        double b;           real(c_double) :: b=7.2
  s = "abc",      char[8] s;          character(c_char) :: s(8)="abc"
}               };                  end type

// And the IREP definition for the same table:
Beg_struct(irt_t)
  ir_int(a,3)       // Integer named ‘‘a’’, default value 3.
  ir_dbl(b,7.2)     // Double named ‘‘b’’, default 7.2.
  ir_str(s,8,"abc") // String ‘‘s’’, maxlen 8, default "abc".
End_struct(irt_t)
```

Figure 3.49: Lua Tables versus C/Fortran Structured Variables

The MAPP project is an ongoing collaboration between a team of C++ developers and a (mostly) different team of Fortran developers. The great advantage of IREP is that the data tables are simple enough so that nearly all their design, upkeep, and usage in the C++ or Fortran source code can be done by those domain experts. Negotiation regarding how to name a physics variable, or its precise semantics, is much more successful if it can be done directly, instead of requiring a computer scientist to mediate. IREP has been successful in this regard.

On the down side, IREP does require both a C and Fortran compiler, and does make some compromises to meet the needs of both languages. This is not (much of) an issue for the MAPP project, but codes that don’t otherwise require both might find IREP’s necessary constraints to be limiting.

3.10.2 High-order Meshing

This section presents a number of techniques that can be implemented in meshing codes (or, more generally, CAD modeling codes) to generate high quality NURBS meshes from CAD model geometry for use in hydrodynamics codes (e.g. Blast) while respecting user meshing constraints. The techniques presented are applicable only to block-structured 2D and 3D input geometries and exclusively produce quad and hex NURBS meshes (respectively). Additionally, these methods can only be applied to CAD input geometries described using certain primitives and account for only zone count/spacing user constraints (see sections 3.10.2 and 3.10.2 for more information). These techniques are presented in the context of the general transformation procedure used in this section, which consists of two main steps: (1) transform the CAD model geometry into equivalent NURBS geometry and (2) transform the raw NURBS geometry into a proper NURBS mesh. This section also explores the strengths and limitations of each of these techniques, compares the performance of different methods that fill the same role, and evaluates the amalgamation of these techniques in the context of producing meshes for hydrodynamics simulations.

Meshing Background

CAD Inputs To fully describe the CAD-to-NURBS transformations that will be presented in this work, it’s important to first outline the types of input CAD geometry that these methods support. This supported set of CAD inputs can be understood specifically as the set of CAD geometry output by ACIS (the CAD modeler used to generate the CAD inputs for this work). There are a few existing publications that already outline ACIS’ supported set of CAD representations in depth [83], but a short overview will be included in this work as well for the sake of expediency and completeness.

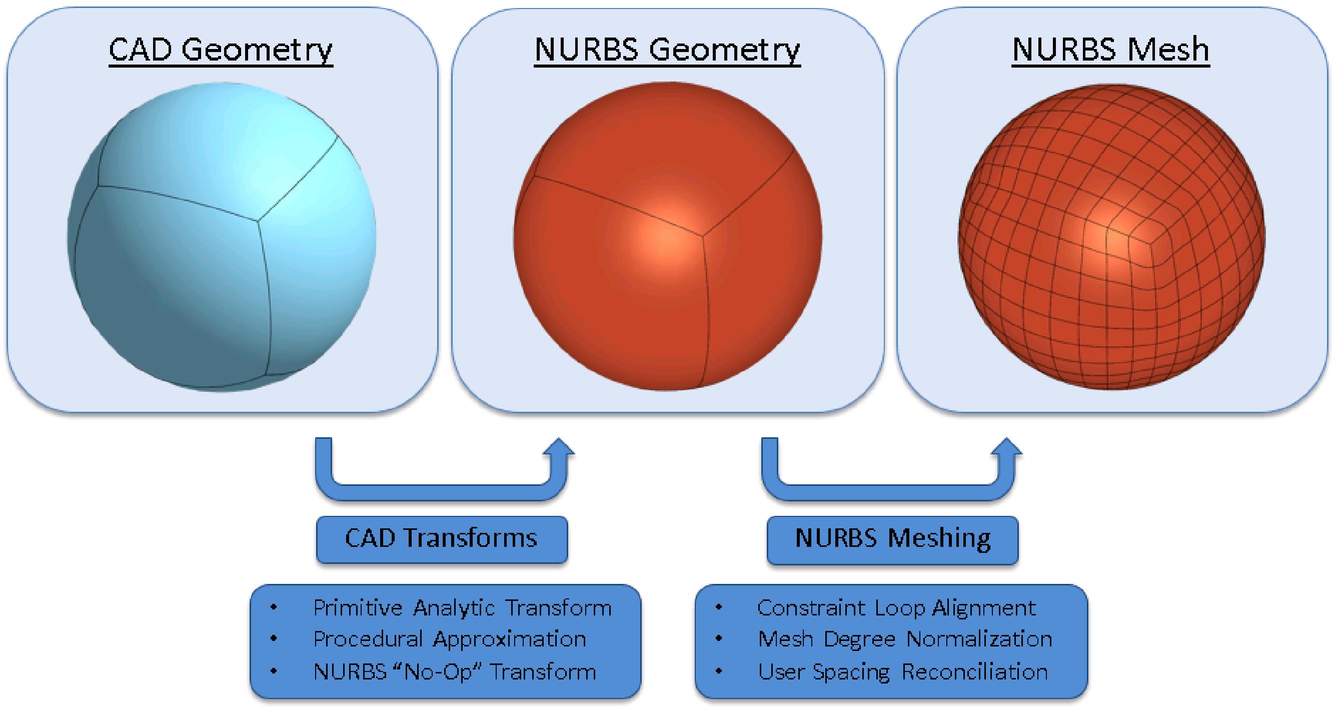


Figure 3.50: An abstract view of the stages in the CAD-to-NURBS meshing algorithms.

The CAD geometries generated by ACIS can be understood broadly as combinations of components that are each described using so-called primitive descriptions. These descriptions (referred to as geometric primitives) fall into one of three categories: (1) analytic formulations (e.g. the equation for a sphere), (2) procedural formulations (e.g. a solid produced via volumetric intersection), and (3) NURBS elements (e.g. a NURBS surface). Since each geometric entity can be broken down into a discrete sequence of geometric primitive pieces, processing these components piecewise is equivalent to processing them as a unit. Since the former method simplifies the interface a bit, the CAD-to-NURBS methods presented in this work will intake singular geometric primitives as input.

Mesh Constraints There are a plethora of constraints a user may wish to impose on a mesh, but only a subset of the most widely used of these constraints are considered in this work. In particular, the paper solely explores how to apply zone count (i.e. the number of zones to propagate along a constraint loop) and zone spacing (i.e. the spacing between the zones propagated along a constraint loop) user constraints. Since the structure of the geometry in a NURBS meshes adds additional nonremovable zone constraints to each constraint loop, zone count is interpretted as a minimum number of zones for a constraint loop. For similar reasons, linear zone spacing is the only zone spacing constraint that's considered for nontrivial constraint loops (i.e. loops with NURBS knot vectors containing one or more intermediate values). It's also important to note that this paper presents methods that prioritize zone quality over mesh efficiency, so zone spacing constraints may introduce additional zones to improve mesh quality.

CAD-to-NURBS Transformations

In this work, *CAD-to-NURBS transformations* are defined to be the set of transformations that take CAD geometries defined by analytic, procedural, and NURBS primitives and produce equivalent or near-equivalent NURBS geometry. This set is further divided into three subsets based on the dimensionality of the geometry being transformed, resulting in three main categories: curve (1D), surface (2D), and solid (3D) CAD-to-NURBS transformations. For any of these dimensions d, the set of dD transformations takes the transformed versions of its $\{\forall d_i \in \mathbb{Z}^+ \mid d_i < d\}$D boundaries as input. As a result, the algorithm that performs the CAD-to-NURBS transformation on the entire input CAD geometry must be structured so that all geometry of a particular dimension is processed before geometry of successive dimensions. A pseudocode implementation of this overarching algorithm can be found in Listing 3.13.

Listing 3.13: CAD-to-NURBS Algorithm

```
def cad_to_nurbs(cad_primitives):
  # Static list of CAD to NURBS transformation functions by dimension.
  cad_to_nurbs_functions = [cad_to_nurbs_1d, cad_to_nurbs_2d, cad_to_nurbs_3d]

  # Table that maps CAD primitives to their NURBS equivalents.
  cad_to_nurbs_table = {}

  # Process all geometry of previous dimensions before successive dimensions.
  for dim in [1, 2, 3]:
    dim_cad_primitives = [cp for cp in cad_primitives if cp.dim == dim]

    for cad_primitive in dim_cad_primitives:
      nurbs_boundaries = [cad_to_nurbs_table[cpb] for cpb in cp.boundaries]
      nurbs_primitive = cad_to_nurbs_functions[dim-1](cad_primitive, nurbs_boundaries)
      cad_to_nurbs_table[cad_primitive] = nurbs_primitive

  return cad_to_nurbs_table.values()
```

Since each CAD primitive type is so drastically different in representation, each CAD-to-NURBS transformation algorithm must process each of these types differently. Fortunately, this caveat doesn't require too much additional functionality as the analytic and NURBS types can be handled simply and uniformly across transformation algorithms. In particular, NURBS primitives can be skipped since they're already in the correct geometric format and analytic primitives can be fed through an analytic conversion table [84] based on type and parameters to produce equivalent NURBS results. Each of our methods focus on transformations for procedural and complicated analytics primitives (i.e. analytics that don't have exact NURBS representations) and omit the details related to the uniform processing for NURBS and simple analytic primitives for the sake of brevity.

Curve (1D) Transformations The curve transformation procedure that we present consists of three primary steps: (1) reduce the curve's procedural definition into simple analytic/NURBS components, (2) transform each of these individual components into NURBS components, and (3) use NURBS operations based on the original procedural definition to generate a final NURBS curve. The algorithm required for the first step is entirely dependent on the opacity of the CAD kernel; this step will either require querying the kernel for a curve's primitives (if the kernel supports such an operation) or reconstructing components based on the operations performed on a curve (if the previously described kernel operation is unavailable). The second step requires simply processing each individual primitive component in the same way that base primitives were processed: by doing nothing for NURBS and using an analytic conversion table [84] for analytics. The procedure for the third and final step will involve uniting, intersecting, and splitting the NURBS components based on the procedural definition of the curve, all of which are well-defined NURBS operations [85, 86].

It's worth mentioning that the accuracy of the curves produced by this algorithm will be defined primarily by the accuracy of the chosen NURBS operations for step (3). Since the result curves will be used heavily in following 2D and 3D CAD-to-NURBS transformation algorithms, we recommend choosing highly accurate methods to execute these NURBS operations.

Surface (2D) Transformations The process for transforming a CAD primitive into a NURBS element becomes a bit trickier when the problem moves to higher-dimensional geometry. The crux of the problem is that procedural geometries that span two or more dimensions can be defined by geometric intersections that cannot be easily simplified by manipulating the parameter space of the source geometry (as can be done with curves). Consider a surface defined by projecting an elliptical curve onto a sphere; it's possible to break this problem down into a set of NURBS components, but impossible to adjust the source component (i.e. the spherical NURBS surface) to fit the intersection parameters (i.e. the elliptical NURBS boundary curve) using linear parameter space adjustments. CAD modelers typically represent such geometries using trimming, but this form of representation doesn't translate directly to NURBS geometry. This fact combined with the nuances introduced by additional geometric dimensions require completely different techniques to be applied in order to perform proper CAD-to-NURBS translations.

There are a few existing works that have explored the problem of producing NURBS geometry from procedural CAD geometry, but the solutions they propose either utilize techniques that are incompatible with standard NURBS definitions [87] or produce NURBS geometry with varying element quality [88]. While solutions in the former class are completely unusable for our purposes (most hydrodynamics codes that support NURBS elements only support their standard definitions), solutions in the latter set could be used to produce NURBS meshes at the cost of element quality and (consequently) simulation accuracy. In order to avoid these costs, we propose a few simple and practical alternative methods that prioritize the element quality of outputs: *boundary interpolation* and *surface sampling*. The rationale for the development of two techniques as well as their individual pros and cons will be explored in the following sections.

Boundary Interpolation *Boundary interpolation* is a 2D CAD-to-NURBS transformation technique that utilizes the input surface's four NURBS boundary curves and interpolation methods in order to generate an approximately equivalent NURBS surface. The procedure can be broken down into three major steps: (1) use knot insertion to reconcile the differences between the knot vectors of the pairs of boundary curves that lie on opposite sides of the surface, (2) use polynomial blending on the four finalized boundary curves to create a set of internal control points, and (3) combine the produced internal control net with the boundary curves to generate the full approximated surface. The first step involves generating a superset knot vector from two opposite boundary curves and then performing the NURBS knot insertion method [89] on each curve for each of the values in this superset (see Listing 3.14 below). The second step is essentially a function that intakes the four reconciled NURBS surface boundaries, generates a point net using polynomial blending, refines this point net into a control net (i.e. a net of points and weights) using a degree of Hermite interpolation based on the degrees of the boundary curves, and then returns this final control net [3]. The final step is trivial, simply requiring the combination of the boundary and interior NURBS data into a proper NURBS data structure.

Listing 3.14: Knot Reconciliation Algorithm

```python
def reconcile_knots(nurbs1, nurbs2):
  # Assuming that the knot vectors are returned as lists of real values.
  knots1, knots2 = nurbs1.get_knot_vector(), nurbs2.get_knot_vector()

  # Create superset knot vector by combining contents of opposite NURBS.
  knotset = set()
  for knot in knots1 + knots2:
    knotset.insert(knot)

  # Assuming that the "insert_knot" implements the "knot insertion" algorithm.
  for knot in knotset:
    nurbs1.insert_knot(knot)
    nurbs2.insert_knot(knot)
```

The primary benefits of the *boundary interpolation* method are its ability to generate exact results for CAD surfaces that naturally interpolate between their boundary curves (e.g. cylindrical surfaces) and its ability to generate surfaces with minimal knot complexity that still exhibit good mesh quality. Unfortunately, this method performs poorly when used to approximate surfaces that don't interpolate between boundaries (e.g. spherical surfaces) and thus should only be used for interpolation surfaces.

Surface Sampling *Surface sampling* is an alternate 2D CAD-to-NURBS transformation technique that directly utilizes the input CAD surface definition to generate points that are subsequently used in forming the output interpolated NURBS surface. Much like the previous technique, this method is composed of three steps: (1) query the input CAD surface along a rectilinear grid in parameter space for surface positions, (2) query the CAD surface along the boundary points of the aforementioned grid for surface tangents, and (3) use cubic Hermite interpolation to generate an approximated NURBS surface from the previously generated surface positions and tangents. The first two steps should be straightforward sequences of queries to the CAD kernel

[3]It's important to note that the implementation of this function is hidden by the ACIS interface and thus the finer details presented here are speculative.

(with some minor transformations in the second step to generate vectors in the direction of the surface from the surface tangents). The third step involves taking the matrix of surface points and boundary tangents and generating a corresponding Hermite bicubic surface, forming a final NURBS surface by imposing uniform weighting and uniform third-degree knot vectors on the position net of the bicubic surface [4].

The *surface sampling* method essentially fills the role of a fallback for the *boundary interpolation* method. The former should be used when the latter cannot produce exact results since in such circumstances the former produces elements with high zone quality that tend to be more accurate to the source CAD geometry. It's worth noting that the rectilinear grid of surface parameters used to specify the sampling points for this method is configurable and can be set statically, based on the structure of the input CAD surface, or based on user input. Regardless of the specification method used, using higher grid densities for more complicated surfaces is preferable as the accuracy of this method increases with the density of the sampling grid (much like linear meshing methods).

Solid (3D) Transformations In the abstract, the CAD-to-NURBS solid (3D) conversion problem is an extension of the previously explored surface (2D) conversion problem with a few additional caveats. In particular, the addition of another dimension frees up a degree of flexibility, which is further broadened by the fact that there are no hard geometric constraints on the internals of a NURBS solid (unlike curves and surfaces, which must fit a set of particular geometric constraints to properly represent the source geometry). With these freedoms in mind, we have chosen to pursue conversion methods that attempt balance satisfying user-imposed constraints with ensuring good zone quality. The body of research supporting these techniques is much more shallow than those for 1D and 2D conversion techniques, so our methods are much more primitive; even still, they tend to produce fairly good mesh quality for well-behaved meshes[5].

The techniques we've developed all follow the same general procedure, which is comprised of the following steps:

1. Locate the transformed NURBS curves and surfaces of the source CAD solid.
2. Generate a set of internally interpolated points using the boundary NURBS curves/surfaces and CAD TFI sample points.
3. Create the NURBS solid defined by the source NURBS curves/surfaces and the internally interpolated control points.

The primary degrees of freedom in this procedure lie in (2) how points are interpolated and (3) in how these points are combined to form a NURBS solid. After trying a couple of different techniques, we landed on a simply and robust combination of methods: (2) interpolate using TFI-generated points informed by the control lattice of the original CAD solid and (3) to coalesce these points simply as a solid of the correct degree comprised of the set of aforementioned interpolated TFI control nets. A high-level overview of this algorithm is presented in Listing 3.15.

Listing 3.15: CAD-to-NURBS 3D Algorithm

```python
def cad_to_nurbs_3d(cad_solid, nurbs_surfaces):
  solid_degree = max(s.get_degree(d) for s in nurbs_surfaces for d in ['U', 'V'])

  # use surface directions to determine logical NURBS origin, then return
  # curves grouped by bottom (curves connected to origin w/o opposite corner),
  # top (curves connected to opposite corner w/o touching origin), and middle
  (bot_curves, mid_curves, top_curves) = nurbs_group_curves(nurbs_surfaces)
  # use the first of the middle-spanning curves as the anchor for interim
  # control point/surface generation
  anchor_curve = mid_curves[0]

  # use points generated with TFI of CAD solid, sorted by plane from bottom
  # to top of logical NURBS orientation
  bot2top_sample_planes = mesh_get_samples(cad_solid)
```

[4]The implementation of this step is also hidden behind the ACIS interface and thus its exact implementation details are unknown; however, we did speak to ACIS developers and confirmed that some form of cubic interpolation is used.

[5]Where a "well-behaved mesh" is defined to be a mesh with low-skew geometric blocks and relatively low-density mesh zones

```
# use points generated with TFI on the reference middle curve as an
# anchor for generating interim NURBS curves
bot2top_sample_points = mesh_get_samples(anchor_curve)

plane_params = [anchor_curve.get_param(p) for p in bot2top_sample_points]
interp_surfaces = [bot2top_sample_planes[0]]
for control in anchor_curve.get_controls()[1:-1]:
  param = anchor_curve.get_param(control.point)
  interp_index = bisect.bisect_left(plane_params, param)
  interp_param = plane_params[interp_index]

  interp_surface = lerp(param-interp_param,
    bot2top_samples[interp_index], bot2top_samples[interp_index+1])
  interp_surfaces.append(interp_surface)
interp_surfaces.append(bot2top_sample_planes[-1])

return nurbs_solid_from_controls(interp_surfaces, solid_degree)
```

NURBS Meshing

Now that we have a collection of NURBS elements to work with, we need to modify these elements to produce valid NURBS meshes that match user constraints. We refer to the process of sewing of NURBS elements into a cohesive whole as *loop knot alignment* and the process of imposing mesh quality/density constraints requested by the user as *zone spacing imposition*. Both of these steps are outlined in the sections to follow.

Loop Knot Alignment We employ two methods for *loop knot alignment*: knot adjustment and mesh resampling.

The procedure for knot adjustment is to (1) create a superset knot vector for each constraint loop in the mesh by iterating through each curve/surface/solid and taking the maximum multiplicity for each knot value and (2) impose this superset knot vector onto each constituent NURBS curve/surface/solid through knot insertion [86] (see Figure 3.51).

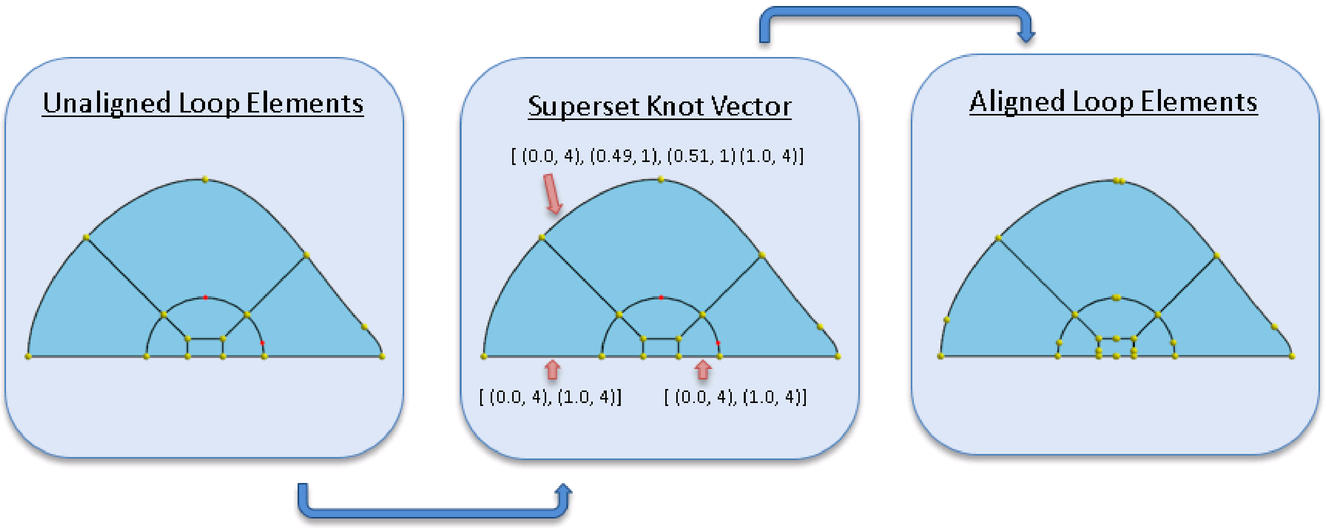


Figure 3.51: Steps in ”Knot Adjustment” Knot Alignment Strategy

The procedure for mesh resampling involves (1) sampling each element at the specified zone spacing to generate new points, (2) calculating the tangents at the boundaries of the NURBS elements, and finally (3) creating new NURBS elements through cubic Hermite interpolation of the calculated samples/tangents. A nice property of this technique is that it occurs ”automatically” if the CAD-to-NURBS transforms were performed using uniform resampling techniques (see Figure 3.52).

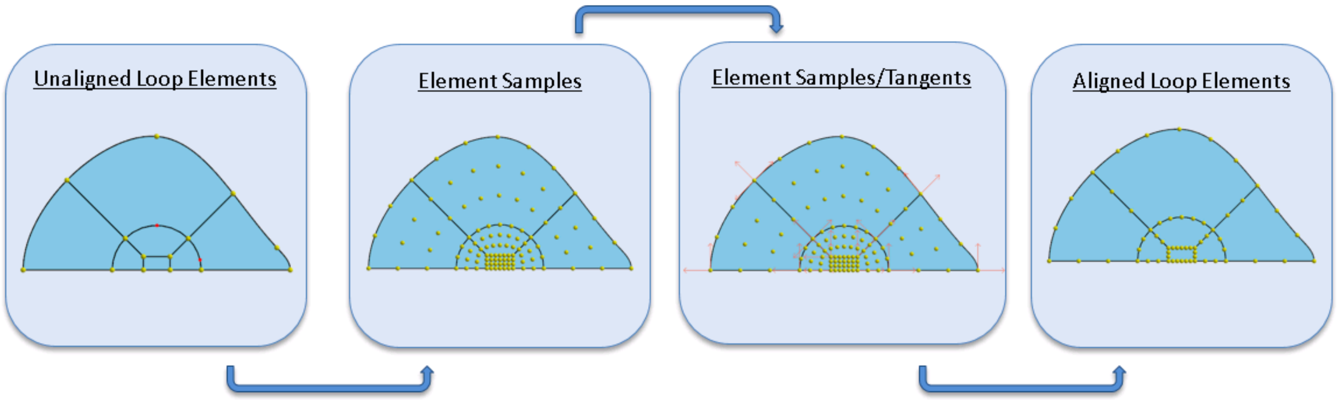


Figure 3.52: Steps in ”Mesh Resample” Knot Alignment Strategy

Zone Spacing Imposition Finally, now we have a valid NURBS, the final step is to perform *zone spacing imposition* to ensure that the mesh follows user requirements for mesh quality and density. Imposing zone spacing constraints that are uniform are much simpler than those that are not, so we cover algorithms for these two cases separately.

If a given constraint loop needs to have uniform zone spacing imposed, the procedure is to (1) find the smallest knot span in the constraint loop’s knot vector, (2) iterate over each knot span ks_i and record the number of times it can be subdivided by the minimum knot span, i.e. $\left\lfloor \frac{ks_i}{min(ks_j)} \right\rfloor - 1$, and finally (3) iterate over the subdivision record and perform knot insertion for each knot span subdivision it describes (see Figure 3.53).

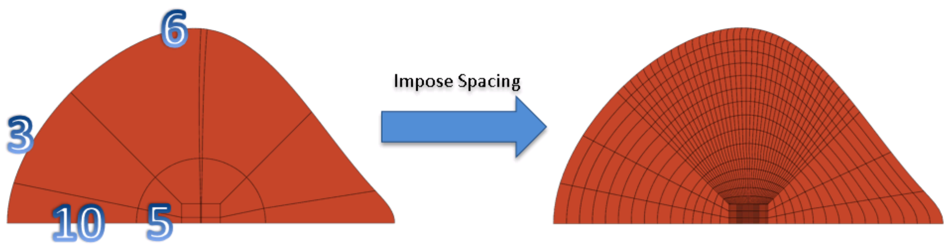


Figure 3.53: An example of imposing simple uniform zoning constraints onto a NURBS mesh.

If a constraint loop has been marked by the user for non-uniform zone spacing (e.g. geometric spacing), then we apply the following algorithm: (1) verify that the given constraint loop has a trivial knot vector (i.e. the knot vector contains no interim control points; just endpoints), (2) transform the parameterization of the given zone spacing into the abstract parameter space that spans the endpoint knots using linear interpolation, and finally (3) use knot insertion to place each of the knot values represented in the new parameterization (see Figure 3.54).

Results

The meshes output using the techniques described in this work have been tested in a number of hydrodynamics simulations. The results for a subset of these tests were published in [13] and some of these results have been provided in Figure 3.55. Additional results have been used in code-to-code comparison tests; one such example is presented in Figure 3.56.

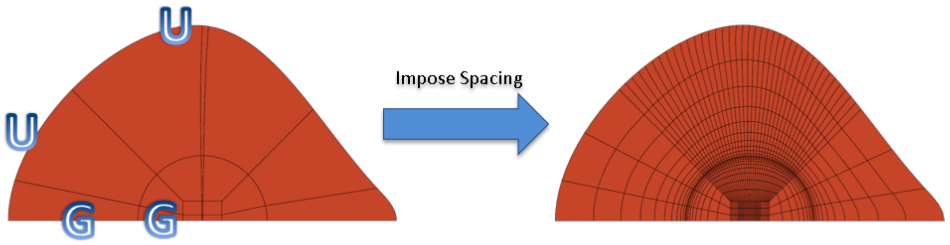


Figure 3.54: An example of imposing nontrivial uniform/geometric zoning constraints onto a NURBS mesh.

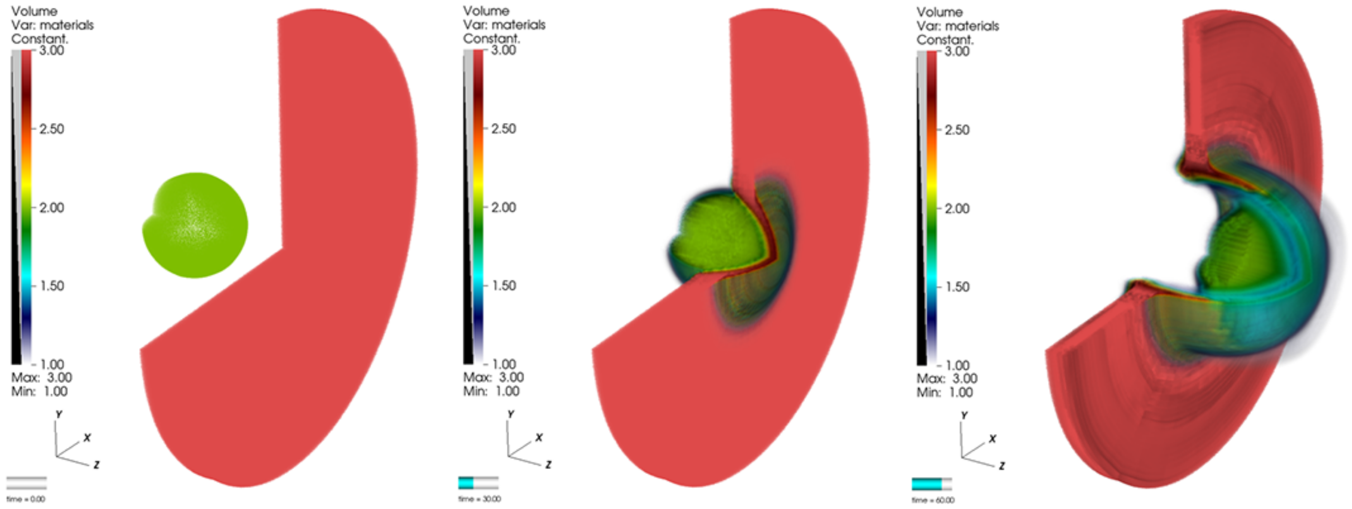


Figure 3.55: The results of using a high-order mesh in Blast to simulate the steel ball impact problem.

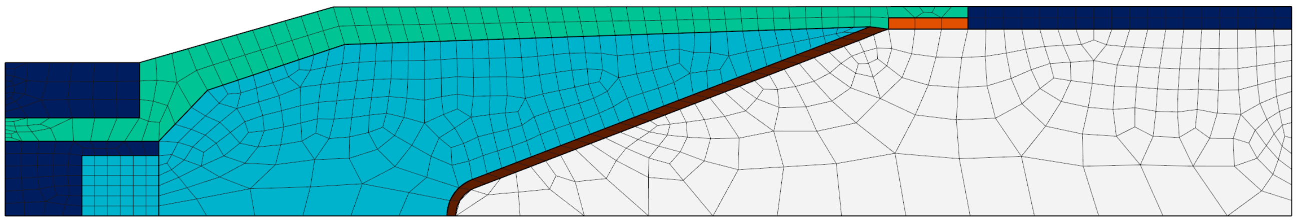
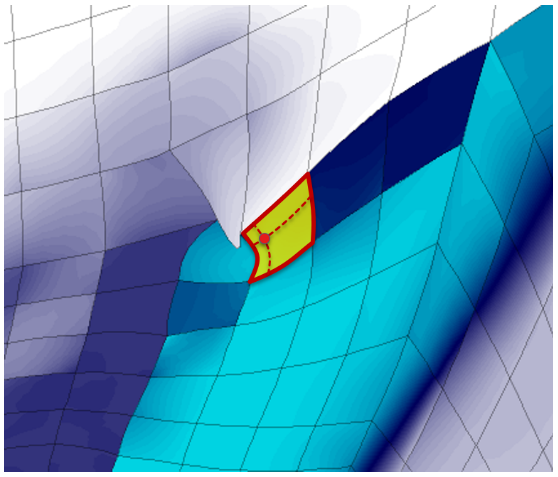
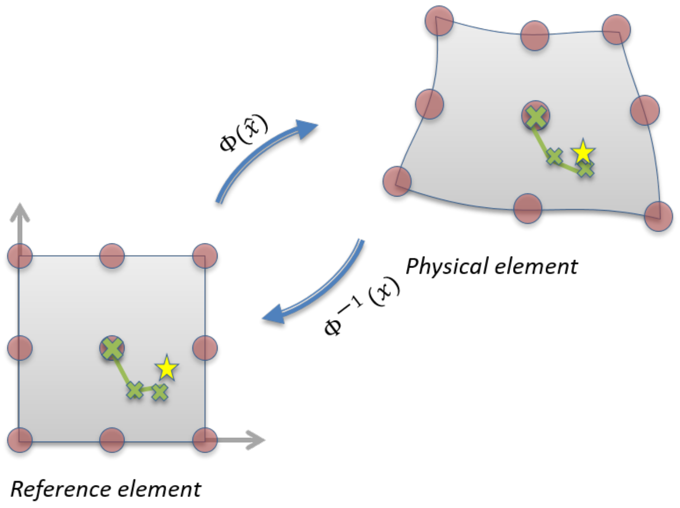


Figure 3.56: The baseline high-order 2D mesh for the BRL-81a shaped charge problem (Blast/Ares comparison problem).

3.10.3 Tracer particles

MARBL's tracer particle package is based on an efficient and accurate point-in-cell query for high order elements. The point-in-cell query, implemented in Axom and MFEM finds the mesh element containing a point in space, as well as the parametric coordinates of the point within that element, see Figure 3.57(a). It allows probing field values at arbitrary points in space. When tracking a particle, we utilize an efficient spatial index from Axom to reduce the number of candidate elements that must be searched. For each such candidate, we utilize a robust Newton-Raphson solver from MFEM to determine if the point is contained within that high order element, see Figure 3.57(b). During that process, we obtain the parametric coordinates of the point within the element. These are used to interpolate field values, for example, the value at that location within a quadratic density field, as


(a) Point location



(b) Newton-Raphson search

Figure 3.57: Point location. Probing density in a HO mesh.

in Figure 3.57(a).

3.11 Steering

The main topic of this section is code steering. We define code steering as the runtime interactive state, dynamic user input, and the front-end that interacts with the runtime.

3.11.1 Runtime Interactive State

The runtime interactive state is a developer-defined set of Lua objects that alias the `Simulation` (3.3.1) along with each `Package` (3.3.1) used in a given problem and can be queried and manipulated in the Lua interpreter at runtime. For example, with the `simulation` object a user can query the current cycle or time step and set cycle, simulation time, or wall time stopping criteria; additionally, methods for writing a checkpoint, dynamically adding or removing packages, and problem regeneration (among others) are available with the `simulation` object.

Each Package defines a set of attributes and functions that are exposed to users, and upon initialization each package is added as a sub-table to the global `simulation` Lua table as `simulation.<package name>`. For developers, adding new attributes is easily done whether they want to alias to a direct value or a callback for lazy initialization:

```
// 'ie' is a member double, we capture the alias
// exposed as 'simulation.blast.ie'
registerScalar("internal_energy", ie, "r");
// exposed as 'simulation.blast.grind_time'
registerScalar("grind_time", [this, num_qpts]() {
  return m_sim->m_cycle_time / num_qpts(MPI_SUM) / m_sim->getNumRanks();
});
// exposed as 'simulation.blast.kinetic_energies[material_id]'
registerArray("kinetic_energies", [this](int i) { return getKineticEnergy(i); });
```

3.11.2 Dynamic User Input

Dynamic user input is built primarily with the `events` table, which is a collection of predicate-callback function pairs used to describe runtime behavior. MARBL includes a set of common predicates in the `timers` table. Next we'll describe events through a series of examples:

Uniform Checkpointing Suppose you want to write a fixed number of checkpoints with a uniform time spacing in between each. We can easily create this using the `timers.uniform` trigger and the `simulation:checkpoint()` function:

```
events = {
  {
    -- write 10 checkpoints between 0 and stop_time
    timers.uniform(0, simulation.stop_time, 10),
    function() simulation:checkpoint() end
  }
}
```

Printing problem size at startup With the `timers.at_start()` trigger we can create an event that prints the number of degrees of freedom in a MARBL-lag simulation:

```
events = {
  {
    timers.at_start(),
    function() print(F"total qpts across {mpi.nprocs} ranks: {simulation.blast.num_qpts}") end
  }
}
```

The `F"..."` strings automatically substitute the result of the enclosed Lua expression, much like F-Strings in Python.

In memory problem regeneration at a specific time Sometimes a user may want to trigger an event at a specific cycle. While they could write their own predicate (`function() return simulation.time == t end`) MARBL includes the `timers.times({t1, t2, ...})` as an easy alternative.

Here we pulled an example from an input file where the `ale_conditions` IREP variable is updated and the problem is regenerated (in memory restart) using the `simulation:generate()` function:

```
t_release = 21.2e-3
-- Release gas/ice interface for 2D/3D
-- Append to the existing events table
events[#events+1] = {
  timers.times({t_release}),
  function()
    print("==> releasing Lagrangian interface")
    ale_conditions.specified = {1,2,3,6,8}
    method.lagrangian.remesh_tmop_target = "IDEAL_SHAPE_EQUAL_SIZE"
    method.lagrangian.remesh_tmop_limit_scale = 5e-3
    simulation:generate()
  end
}
```

Events can be added, removed, and listed during runtime as well as in the input file. A full list of timers is available by tab-completing on the `timers` table:

```
lua> timers.<tab>
timers.all                  timers.cycles                timers.taper                 timers.uniform
timers.any                  timers.delay                timers.time_delta_uniform
timers.at_exit              timers.every                timers.times
timers.at_start             timers.once                 timers.timeslice
```

3.11.3 Front-end

By default MARBL is run in batch mode and does not accept runtime input from the user, but there are a few optional front-ends that can be enabled by the user with command line flags.

REPL

The Read-Eval-Print-Loop (REPL) is available by passing the `-i` option on the command line. The REPL is just a command line Lua interpreter tied to MARBL's internal Lua state and supports history and tab-completion. An example is included next.

In this example the user starts a simulation; changes the stopping criterion and runs to completion; continues the simulation after the previous stopping criterion; queries some global and package-specific attributes; adds the `Ascent` package (See 3.12.3) and saves the density field data; lists the file contents of MARBL's current output directory; and finally writes a checkpoint to disk.

```
> /usr/apps/marbl/bin/marbl-dev -lag ../blast/examples/sedov/sedov.lua -i -new
...
[marbl INFO]: cyc:    0, t: 0.000e+00,  init dt: 2.934e-03, dE: 0.000e+00, iter<hydro:0>, rtime: 0.00e+00
lua> simulation.stop_cycle=43
lua> run
[marbl INFO]: Repeating cycle 0 with dt: 2.934e-03 --> 2.494e-03
[marbl INFO]: Repeating cycle 0 with dt: 2.494e-03 --> 2.119e-03
[marbl INFO]: Repeating cycle 0 with dt: 2.119e-03 --> 1.802e-03
[marbl INFO]: Repeating cycle 2 with dt: 1.802e-03 --> 1.531e-03
[marbl INFO]: cyc:   20, t: 3.363e-02, hydro dt: 1.867e-03, dE: 1.735e-13, iter<hydro:33>, rtime: 3.07e-01
[marbl INFO]: Repeating cycle 22 with dt: 1.867e-03 --> 1.587e-03
[marbl INFO]: Repeating cycle 24 with dt: 1.587e-03 --> 1.349e-03
[marbl INFO]: Repeating cycle 27 with dt: 1.349e-03 --> 1.146e-03
[marbl INFO]: Repeating cycle 34 with dt: 1.193e-03 --> 1.014e-03
[marbl INFO]: Repeating cycle 38 with dt: 1.055e-03 --> 8.965e-04
[marbl INFO]: Repeating cycle 39 with dt: 9.145e-04 --> 7.773e-04
[marbl INFO]: Repeating cycle 39 with dt: 7.773e-04 --> 6.607e-04
[marbl INFO]: cyc:   40, t: 5.853e-02, hydro dt: 6.739e-04, dE: 2.304e-13, iter<hydro:115>, rtime: 6.72e-01
[marbl INFO]: stop cycle reached: 43 >= 43

lua> simulation.cycle
43
lua> step(57)
[marbl INFO]: cyc:   60, t: 7.221e-02, hydro dt: 7.589e-04, dE: 1.865e-14, iter<hydro:34>, rtime: 9.25e-01
[marbl INFO]: Repeating cycle 63 with dt: 7.896e-04 --> 6.712e-04
[marbl INFO]: Repeating cycle 64 with dt: 6.712e-04 --> 5.705e-04
[marbl INFO]: cyc:   80, t: 8.433e-02, hydro dt: 5.705e-04, dE: 1.829e-13, iter<hydro:36>, rtime: 1.22e+00
[marbl INFO]: Repeating cycle 88 with dt: 5.935e-04 --> 5.045e-04
[marbl INFO]: cyc:  100, t: 9.504e-02, hydro dt: 5.045e-04, dE: 2.031e-13, iter<hydro:36>, rtime: 1.50e+00

lua> simulation.cycle
100
lua> simulation.time
0.095044403352986
lua> simulation.blast.timestep_min.jacobian
0.028568830240978
lua> ascent = simulation:add_package("ascent")
lua> ascent:<tab>
```

```
ascent:dump_fields        ascent:execute_actions    ascent:print_variables    ascent:results_directory
ascent:snapshot
lua> ascent:dump_fields({"density"})
[marbl INFO]: Ascent data dump: 0.024 seconds
lua> os.execute(F"ls {simulation.output_dir}")
build_info.log  field_dump.cycle_000099       marbl_0000060       user_input_log_0000000.lua
checkpoint.log  field_dump.cycle_000099.root  marbl_0000060.root
0
lua> simulation:checkpoint()
[marbl INFO]: writing marbl-out/marbl_0000100.root
```

Jupyter

For a richer interactive experience the MAPP codes also support using Jupyter as a front-end. While this includes less well-known Jupyter front-ends like the command-line console, the QT console, and nteract, the main target was the Jupyter Notebook and Jupyter Lab, which many users are familiar with.

The Notebook/Lab front-end exposes new features beyond that of a traditional command-line interface, including inline graphic and dynamic content along with inline documentation and instructions. In Figure 3.58 we can see some of the inline visualization features offered by the Notebook, including inline plots with Matplotlib and inline, interactive, field visualization with PyGLVis (3.12.2). The Notebooks run under Livermore Computing's (LC) managed JupyterHub service, which supports both the Notebook and Lab front-ends.

The centralized authentication to LC services allows the MARBL team to publish tutorial Notebooks to MARBL's web-docs (see 3.6.5), which manifests in a section in the documentation on Jupyter with links that open each notebook in the user's Jupyter-Hub account (See the "Example Notebooks" section in Figure 3.59).

Just as Jupyter supports a variety of kernels in different development languages we install Python-based and Lua-based kernels for connecting to MARBL.

Lua Kernel The Lua kernel behaves in much the same way as the built-in REPL; code blocks are just Lua. The PyGLVis widget is supported in the Lua kernel but unlike input files and the REPL the `tide.python` table is not supported due to the existing Kernel setup. We plan to address this in a future version.

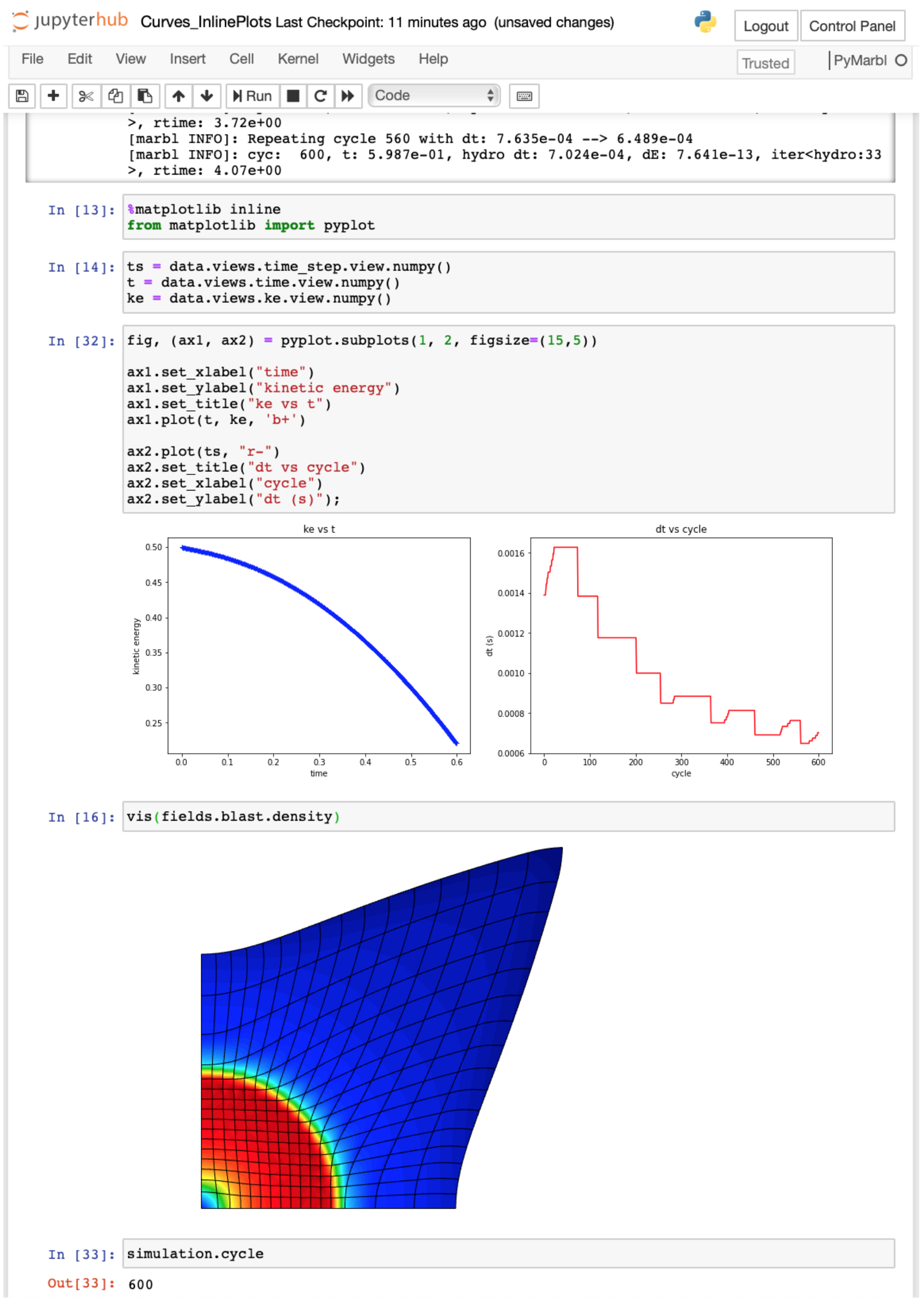
Python Kernel The Python kernel tries to create a hybrid experience for those more familiar with Python and its ecosystem. The kernel is basically a vanilla iPython kernel with the addition of a `lua` object for interacting with MARBL's built-in Lua state and `%lua` and `%%lua` line and cell magics for explicit inline Lua code blocks.

Security and Connecting to Remote Kernels When LC's JupyterHub is used, Notebooks are run on the login node of whichever machine the Notebook server is running. This is okay for small, tutorial-style Notebooks but prevents users from running compute-heavy ones. Additionally, a user may just want to attach to a running simulation to temporarily query the state, something which isn't supported with the current Jupyter infrastructure.

The MARBL teams has developed 2 solutions to the problem of connecting Notebooks to simulations on compute nodes.

First is a secure bridge kernel that lets users connect to simulations running on a compute node of cluster A from a login-node on cluster B (when A and B are on the same network). The bridge kernel supports attaching to existing simulations but suffers from a less-intuitive interface; the user opens a Notebook that is stateless, from which they must connect to a running job with a Widget (interactive graphical object).

The second approach is a plugin that lets a user connect to an existing native kernel. The native kernel approach is advantageous because a Notebook is now stateful in a way the bridge kernel was not, but this approach suffers from an inability to connect to a



```
>, rtime: 3.72e+00
[marbl INFO]: Repeating cycle 560 with dt: 7.635e-04 --> 6.489e-04
[marbl INFO]: cyc:  600, t: 5.987e-01, hydro dt: 7.024e-04, dE: 7.641e-13, iter<hydro:33
>, rtime: 4.07e+00
```

```
In [13]: %matplotlib inline
         from matplotlib import pyplot
```

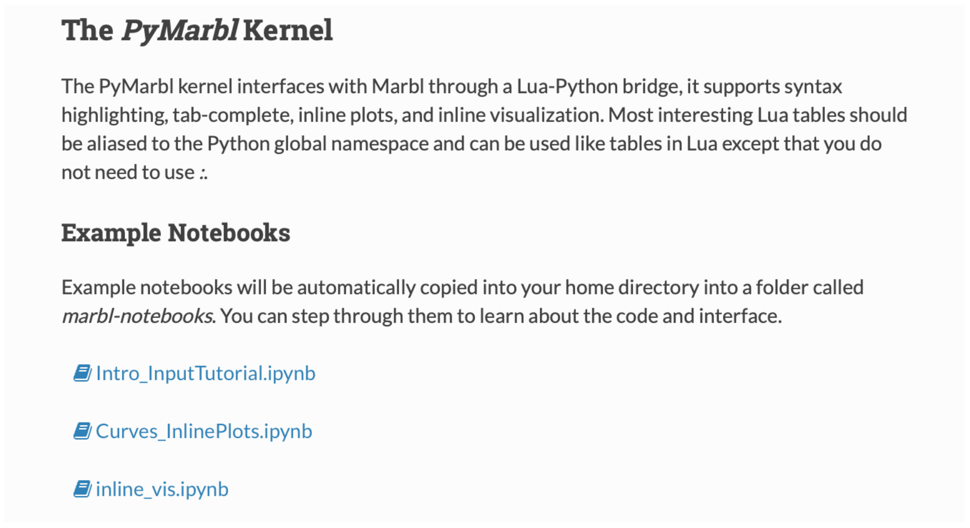
```
In [14]: ts = data.views.time_step.view.numpy()
         t = data.views.time.view.numpy()
         ke = data.views.ke.view.numpy()
```

```
In [32]: fig, (ax1, ax2) = pyplot.subplots(1, 2, figsize=(15,5))

         ax1.set_xlabel("time")
         ax1.set_ylabel("kinetic energy")
         ax1.set_title("ke vs t")
         ax1.plot(t, ke, 'b+')

         ax2.plot(ts, "r-")
         ax2.set_title("dt vs cycle")
         ax2.set_xlabel("cycle")
         ax2.set_ylabel("dt (s)");
```

```
In [16]: vis(fields.blast.density)
```

```
In [33]: simulation.cycle
Out[33]: 600
```

Figure 3.58: The screenshot shows a Jupyter Notebook running on Livermore Computing's JupyterHub connected to a MARBL Noh simulation running on a CTS-1 cluster. In this example the user has run the simulation for a few hundred cycles and recorded time-histories for a few values. The time-histories are plotted with Matplotlib after retrieving the data from Sidre as a Numpy array alias. Additionally the density field is visualized inline using the PyGLVis Jupyter widget



The *PyMarbl* Kernel

The PyMarbl kernel interfaces with Marbl through a Lua-Python bridge, it supports syntax highlighting, tab-complete, inline plots, and inline visualization. Most interesting Lua tables should be aliased to the Python global namespace and can be used like tables in Lua except that you do not need to use *:*.

Example Notebooks

Example notebooks will be automatically copied into your home directory into a folder called *marbl-notebooks*. You can step through them to learn about the code and interface.

Intro_InputTutorial.ipynb

Curves_InlinePlots.ipynb

inline_vis.ipynb

Figure 3.59: MARBL's web-docs includes a section with Jupyter tutorial links that when clicked open a new Notebook with the user's JupyterHub account

simulation that is actively taking time-steps and can only connect to one that is in a blocked ”waiting-for-connection” state. The MARBL team is planning to combine the best of both approaches for a native Notebook experience that supports easy connection to running jobs.

Both approaches were developed with security as a priority. The bridge kernel uses TLS-encrypted TCP sockets or UNIX permission controlled file-system sockets. The native kernel approach uses file-system sockets. TCP sockets are not used because Jupyter does not support encryption on them. Both approaches use SSH tunneling for node-to-node connections. Modern implementations of SSH libraries, which are available on all our systems of interest, support file-system socket tunneling.

3.12 Outputs and Analysis

3.12.1 Curves with PyDV

To many users, time-history values are critical for simulation analysis. In MARBL we introduced the `curves` table to make it easy for user's to setup time-history recordings and to write them out to disk. The `curves` table lets a user choose the name, type, precision, and frequency of time-history values and then gives them a simple way to register key-value pairs to be evaluated during curve updates.

```lua
-- create a new curve group called 'c' with a file output name of 'data.csv'
c = curves:new("data.csv")

-- curves are registered as expression-key pairs
-- the expression can either be a string of valid Lua, which will be evaluated
-- every `frequency` cycles
c:register("simulation.time", "time")
c:register("simulation.time_step", "dt")

-- or a function, which will be called every `frequency` cycles
c:register(function() return simulation.time / simulation.cycle end, "current avg dt")
```

For a real-life example, let's look at the curve setup in the Shaped Charge problem:

```lua
c = curves:new("time_history")

c.frequency = 10
c.format = 'ult'
c.reference_column = 'time'

-- cs for integrated quantities
c:register('simulation.time',                   'time')
c:register('simulation.time_step',              'time_step')
c:register('simulation.blast.kinetic_energy',   'kinetic_energy')
c:register('simulation.blast.internal_energy',  'internal_energy')
c:register('simulation.blast.mass',             'mass')
c:register('simulation.runtime',                'run_time')
c:register('simulation.walltime',               'wall_time')
c:register('simulation.cycle_time',             'cycle_time')

-- cs for material-based integrated quantities
for i=1,table.getn(material) do
  local mname = material[i].name
  -- using F-strings it's easy to substitute the iterant into the Lua expression
  c:register(F'simulation.blast.volumes[{i}]',             F'{mname} vol')
  c:register(F'simulation.blast.masses[{i}]',              F'{mname} mass')
  c:register(F'simulation.blast.internal_energies[{i}]',   F'{mname} ie')
  c:register(F'simulation.blast.kinetic_energies[{i}]',    F'{mname} ke')
end

-- cs for particle-based quantities
for i=1,5 do
  c:register(F'simulation.tracer.particle[{i}].velocity:norm()', F'vel_0{i}')
end
```

The curve output format can either be `csv` or `ult` (ultra-curve), a popular format at Livermore, that can be used with PyDV.

PyDV

PyDV is the follow-up, or modernized re-implementation, of the pdv application. MARBL supports native PyDV output and lets users choose the reference value via the `reference_column` attribute on a curve group.

3.12.2 GLVis

GLVis is a native high-order interactive field and mesh visualization software built on MFEM and developer by the MFEM team. GLVis works in a client-server setting; a user runs the GLVis server application and tells the client where the server is (host and port). When the client sends a visualization request the server creates a new window and renders the request. The server uses OpenGL and X Windows. Since MARBL-lag is built on MFEM we can easily use GLVis for inline visualization. MARBL-lag exposes its fields to the user in the `fields.blast` table. In a simple Sedov run we can tab-complete to see the available fields:

```
lua> fields.blast.<tab>
fields.blast.burn_fraction           fields.blast.inverse_dt_estimates   fields.blast.velocity
fields.blast.density                 fields.blast.mesh_displacement      fields.blast.volume_fraction
fields.blast.energy                  fields.blast.pressure
fields.blast.initial_lengthscale     fields.blast.sound_speed
```

Any of these fields can be passed to the `vis` Lua object that, when configure in GLVis client-server mode, sends visualization requests to the user's GLVis server.

For example, if the user ran the Sedov problem above for 500 cycles then ran `vis(fields.blast.density)` the user would get a new window like than seen in Figure 3.60.

In addition to command-line visualization the `vis` object also supports registering vis fields in an input file. For example, the snippet below will register the energy field for visualization every 100 cycles with some setup options:

```
vis:register(fields.blast.energy, {
  title = 'energy',
  keys = 'Rmjl',
  caption = 'energy',
  x = 0,
  y = 0,
  width = 400,
  height = 400
})

simulation.output_frequency = 100
```

PyGLVis

During a summer internship a student with the MARBL project updated GLVis to use a more modern OpenGL standard. This new version allow us to transpile GLVis, which is written in C++, to JavaScript and WebAssembly using emscripten ([90]), which allowed it to be run in the browser. Once we had this web-accessible library we created a Jupyter Notebook widget in Python, which we called PyGLVis, that allows GLVis to be used in Jupyter Notebooks (and the web more generally). See Figure 3.58 for an example.

3.12.3 Ascent and Devil Ray

Ascent is another option for in situ visualization. Unlike GLVis, it does not require a windowing system to run, which makes it especially useful for hands-off visualization during batch execution. Ascent leverages both distributed-memory and shared-

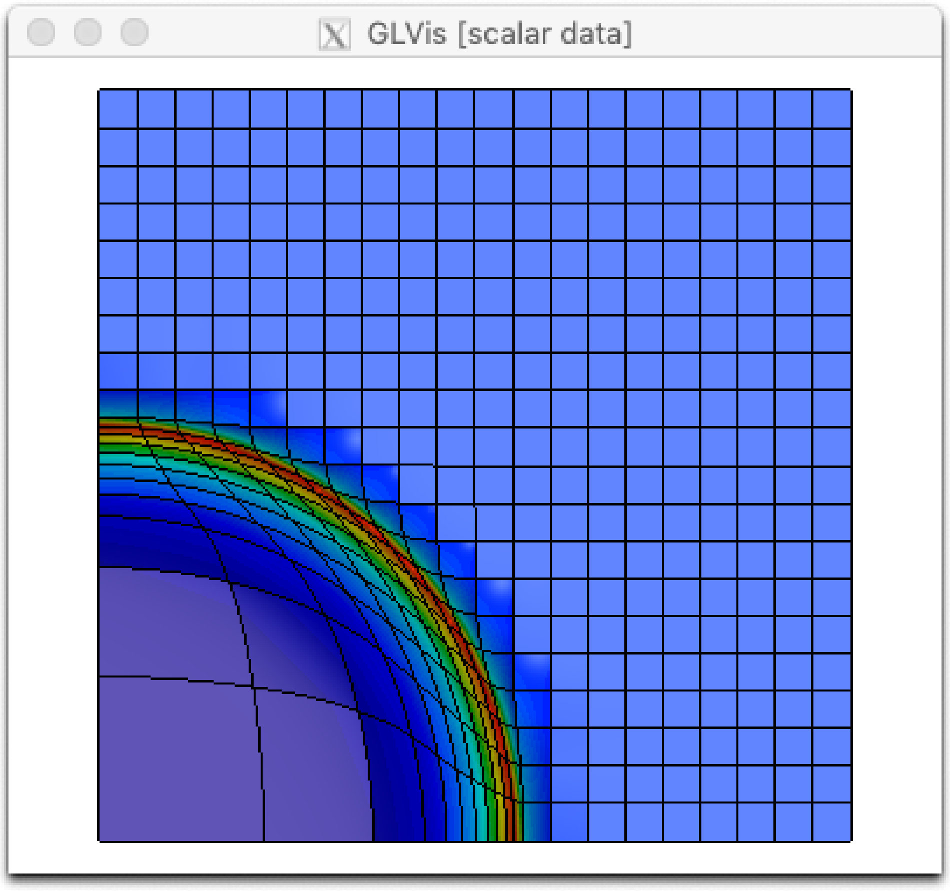


Figure 3.60: GLVis visualization of a 2D Lagrangian Sedov simulation after 500 cycles. The `Rmj` keys were used.

memory parallelism to provide scalable and performant rendering. It supports ray-traced surface and volume rendering, along with a standard menu of visualization operations including clipping, contouring, thresholding, etc. Ascent uses both VTK-m [91] and RAJA [72, 73] to implement visualization algorithms that are portable to multiple HPC architectures. It uses Devil Ray, described below, for native high-order ray tracing. Devil Ray lets Ascent visualize fields in MARBL-lag without having to first refine them to a low-order representation. Native high order rendering uses less memory and produces better results. Details on Ascent's general design are outlined in Ascent's documentation [92] and in [93].

Ascent accepts Mesh Blueprint data, the same format used by MAPP. This support traces back to the Strawman proxy application [94], which demonstrated new ideas for mesh specification and user-facing APIs. Ascent evolved from Strawman and the mesh specification work evolved into the Conduit Mesh Blueprint, which is now a key part of a broader strategy for sharing mesh data between MAPP components and other tools.

Ascent is configured with YAML ”action” files that dictate which fields to save and what the view(s) to save for each field.

In an example taken from our Shaped Charge problem we can see the optional Ascent visualization setup. If the user adds `avis=true` to their command line when launching MARBL they will get a visualization dump corresponding to the `devil_ray_{2,3}d.yaml` description file 140 uniform steps between the simulation start and end, at `t=35`.

```
avis = default(avis, false)   -- Ascent In-situ visualization
...

tstop  = 35.                  -- Simulation stop time

...

if (avis) then
  simulation:add_package("ascent")

  -- the actions files are expected to be in the cwd
  actions_file = pathof("devil_ray_" .. mdim .. "d.yaml")

  events[#events+1] = {
    timers.uniform(0.,tstop,4*tstop),
    function()
        annotate_begin("ascent")
        simulation.ascent:execute_actions(actions_file)
        annotate_end("ascent")
    end
  }
end
```

Action files can be very long but we include the 2D example here because it is more manageable:

```
# DevilRay: high-order in situ visualization parameters
-
  action: "add_extracts"
  extracts:
    dpseudocolor:
      type: "dray_pseudocolor"
      params:
        field: "density"
        draw_mesh: "true"
        image_prefix: "devil_ray_mesh_%06d"
        image_width:  1024
        image_height: 1024
        color_table:
          name: "rambo"
        log_scale: "true"
        min_value: 0.1
        max_value: 12.0
```

Support for visualizing high-order meshes is a key requirement for MAPP. Like VisIt, Ascent provides an MFEM-based LOR (low-order refinement) path to bridge high-order data into traditional visualization pipelines. Even with this path, the high-order meshes used by MAPP pose challenges for traditional the visualization algorithms. Direct support for high-order is preferred. Ray tracing methods can directly incorporate high-order solves. The Devil Ray project was established to provide this support.

Devil Ray is a ray tracer that directly uses the high-order mesh representation. By leveraging high-order, we can both lower the overall memory requirements and improve accuracy over low-order methods. Devil Ray was initially developed for visualization and supports surface intersections, slicing, volume rendering and simulated radiography. Devil ray has been demonstrated on over 4000 GPUs in MAPP.

Ascent development is supported by both the Exascale Computing Program (ECP) as part of the ALPINE S&T project [95] and by LLNL's WSC program. Ascent is also integrated into several ECP codes. Ascent's use in ECP has help evolve and validate aspects of the Mesh Blueprint relevant to MAPP, notably nesting information for patch-based AMR.

3.13 Developer Survey

As discussed at the beginning of this chapter, a major design goal for MAPP's infrastructure layer has been to make it easy to incorporate new physics models and code capabilities into our modular codebase. To gauge how well we are meeting these objectives, we conducted a survey of 12 developers that are not part of our core MAPP team who have worked on integrating or improving support for modular packages within our code base. One developer provided feedback for work on two separate efforts, leading to 13 total survey responses.

The backgrounds of the surveyed developers were representative of our developer community: of the 12 developers surveyed, 6 are Computer Scientists, 4 are Applied Mathematicians and 2 are Code Physicists. The corresponding projects that they worked on include 4 Physics projects, 2 Math projects and 7 Computer Science/infrastructure projects. Of the 13 projects, 9 resulted in successful integrations, 3 are still in progress and one was ultimately deferred.[6]

We designed our survey with the hopes of gaining actionable responses about our developers' experiences working within our code base. Our five survey questions are therefore somewhat open-ended, leading to feedback that is more qualitative than quantitative as we felt this would better help us improve our workflows and processes. In the remainder of this section, we summarize the developer responses with selected quotes that are representative of the overall feedback that we received. We provide the full survey responses (anonymized and lightly edited for clarity) in Appendix A.1.

Q1: Was the documentation for cloning/building/running and testing the code sufficient to accomplish your tasks?

Dev 01: *"The documentation has always been very complete; I don't believe I've ever had any issues outside of synchronizing the code repositories (which the project has made very easy through utility scripts)"*

Dev 02: *"I must admit that I just asked [the MARBL team] how to do this. The human interaction was much more direct and efficient than going through the documentation."*

Dev 04: *"The docs were really good, and just written documentation got us 95% of the way there. For the small corner cases that popped up, I'd just email you and you'd know what to do."*

Dev 07: *"Yes and no. I was able to clone/build/run/test in the default manner relatively easy, but I needed to contact the developers directly to figure out how to do a build with a custom library."*

Dev 12: *"The docs did a fine job to get me going. There were two items I remember working through... Figuring out how to get a debug build... [and] switching from the default compiler.*

[6]The deferred case was for an LLNL research debugging tool, fpchecker, that detects floating point errors within device code. We hoped this would help us pinpoint an issue related to uninitialized device memory. This tool had some assumptions about device compilers that were incompatible with the device compiler model within the MARBL build system, leading to a complicated integration. Ultimately, we were able to resolve the floating point errors using other tools and deferred the fpchecker integration. We are planning to rework how MBS deals with device compilers to enable support for this tool in the future.

Summary/analysis: While the documentation appears to be sufficient for the general case, we still need to interact more collaboratively with developers for customized builds and features. We are satisfied with this result as it strikes a happy medium between personally getting developers up to speed and with maintaining fully comprehensive documentation that meets all developer use cases within our rapidly evolving codebase. This also helps us gauge which processes are in need of improvement.

Q2: Was the build system sufficiently flexible to integrate your package into the code?

Dev 01: *"Integrating my code was a bit clumsy at first since all updated versions had to be deployed as a new 3rd-party archive, but the more recent introduction of 2nd-party libraries has smoothed this process over considerably."*

Dev 02: *"I liked how 3rd-party libraries were built into the code design upfront... I could offload 90% of building my library into the code without having to worry about [inconsistent] compiler options"*

Dev 06: *"Yes. I mostly just grep'd MAPP for the names of other packages that had been added, and duplicated that infrastructure... There wasn't anything so special about my packages to make that difficult."*

Dev 08: *"Absolutely: it is straight forward and flexible enough to pick just was is needed to run"*

Dev 09: *"An enthusiastic yes: the ability to build with specified branches from different libraries was exceedingly helpful"*

Dev 11: *"Not quite; building the code with a not-before-tried compiler was not straightforward, and we are yet to resolve the compilation errors"*

Summary/analysis: Developers appreciated our distinction between 2nd and 3rd party libraries, as well as our built-in support for consistent flags and "package farms". We note that the Spack project is currently adopting our notion of 2nd party libraries to improve support for co-development across code repositories.

Q3: Was it clear how to test / run your development code to verify its functionality?

Devs 01, 03, 04, 05, 09, 13: *"Yes"*

Dev 01: *"Yes, the documentation on running/testing the code has always made it very easy to run regression tests."*

Dev 06: *"I didn't figure out running/testing from the online documentation, and I had to send an email for recommendations. You pointed me at a small problem I could use for quick iteration testing, and you helped me with the right command line for running tests. With that email help I was able to figure it out."*

Dev 07: *"Sort of? At the time, there was a test problem that I was trying and it had some out of date settings. Rob [Rieben] helped get things in order."*

Dev 10: *"The documentation was clear on how to build MAPP in debug mode, but there wasn't much guidance after that. It would be nice to have additional documentation on how to effectively debug in MAPP. For instance, if I'm just writing code for a single package, I might only want to build the debuggable version of that package so the rest of the code is optimized. It's not clear how to do this using the build system from the documentation."*

Summary/analysis: New feature development can expose bugs and untested/undertested features, which requires more interaction. We value these opportunities, since they inevitably lead to a more robust and featureful codebase.

Q4: How many code team members did you need to consult with in order to accomplish your work?

Dev 01: *"At the start of my time with MAPP, I needed to communicate with 2 or more people in order to work through build processes/integrations, though mine was a frontier use case. Since this use case has become better supported, I've generally had to consult with one person or fewer."*

Dev 02: *"The great thing about the code was that it was a team with interlocking expertise. Sometimes that was inconvenient, but it ensured multiple perspectives were applied to the integration. Each of them provided unique improvements to the overall project. I would say that it encompassed about 5-6 people."*

Dev 04: *Two. Kenny [Weiss] for technical stuff, Kenny + [Rob] Rieben for "is this doing what you want?"*

Summary/analysis: It was interesting to see how our developers responded to this question. 75% interacted with one or two members of the MAPP team in support of their package integration. The remaining cases interacted with up to six team members as these integrations were part of long-term collaborative efforts e.g. in support of a year-long project milestone.

Q5: What recommendations do you have about things that are unclear or need improvement?

The developer responses include the following suggestions:

- (3 developers): Improve information about "package farms" and how to only build necessary packages.
- (2 developers): Improve documentation about how the build system works under the covers. This will help when facing corner case.
- Add an example package recipe exercising most features within our build system.
- Make sure that all test problems in the repository work without any modifications
- Add more examples about how to visualize the results of test problems
- Add more info about how to debug the code

Summary/analysis: We received a lot of actionable feedback in response to this open-ended question that will improve our developer experience. Some of these have already been added or are already in flight.

Chapter 4

Algorithms and Applied Math

4.1 Introduction

This sections discusses key technologies and theory behind our modular math libraries. Several of these developments have been prototyped in our proxy apps: Laghos, Remhos and Pyranda (among others).

4.2 High-Order Finite Elements

4.2.1 MFEM

The Finite Element Method (FEM) is a powerful discretization technique that uses general unstructured grids to approximate the solutions of many partial differential equations (PDEs). It has been exhaustively studied, both theoretically and in practice, in the past several decades [96, 97, 98, 99, 100, 101, 102, 18].

MFEM [103, 104] is an open-source, lightweight, modular and scalable software library for finite elements, featuring arbitrary high-order finite element meshes and spaces, support for a wide variety of discretization approaches and emphasis on usability, portability, and high-performance computing (HPC) efficiency, see Figure 4.1. The MFEM project performs mathematical research and software development that aims to enable application scientists to take advantage of cutting-edge algorithms for high-order finite element meshing, discretizations, and linear solvers. MFEM also enables researchers and computational mathematicians to quickly and easily develop and test new research algorithms in very general, fully unstructured, high-order, parallel settings. The MFEM source code is freely available via Spack, OpenHPC, and GitHub, `https://github.com/mfem`, under the open source BSD license.

MFEM was chosen as the discretization foundation of the Multiphysics on Advanced Platforms Project because of its unique combination of features, including its massively parallel scalability, HPC efficiency, support for arbitrary high-order finite elements, generality in mesh type and discretization methods, support for GPU acceleration and the focus on maintaining a clean, lightweight code base.

Conceptually, MFEM can be viewed as a finite element toolbox that provides the building blocks for developing finite element algorithms in a manner similar to that of MATLAB for linear algebra methods. MFEM includes support for the full high-order de Rham complex [105]: H^1-conforming, discontinuous (L^2), $H(Div)$ -conforming, $H(Curl)$ -conforming and NURBS finite element spaces in 2D and 3D, as well as many bilinear, linear, and nonlinear forms defined on them, including linear operators such as gradient, curl, and embeddings between these spaces. It enables the quick prototyping of various finite element discretizations including: Galerkin methods, mixed finite elements, discontinuous Galerkin (DG), isogeometric analysis, hybridization, and

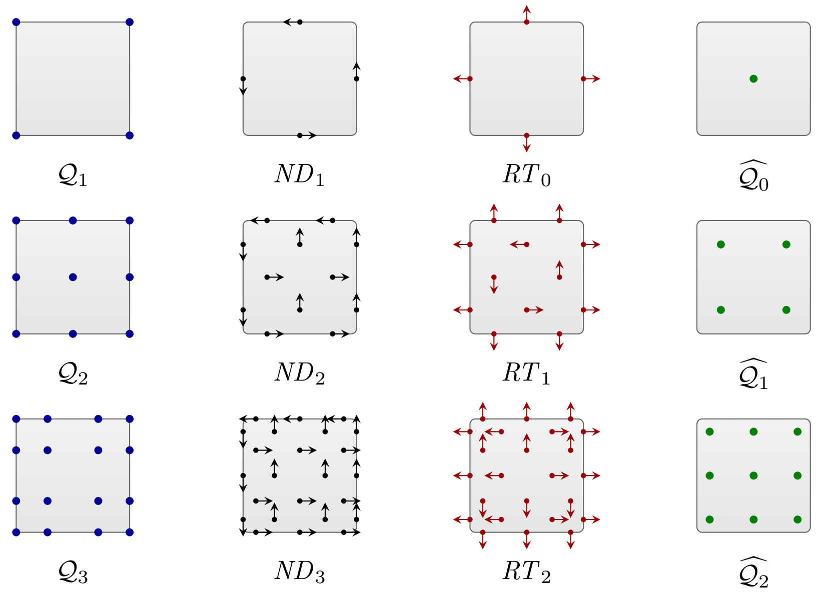


Figure 4.1: Linear, quadratic and cubic H^1 finite elements and their respective $H(curl)$, $H(div)$ and L^2 counterparts in 2D. Note that the MFEM degrees of freedom for the Nedelec (*ND*) and Raviart-Thomas (*RT*) spaces are not integral moments, but dot products with specific vectors in specific points as shown above.

discontinuous Petrov-Galerkin approaches.

MFEM contains classes for dealing with a wide range of mesh types: triangular, quadrilateral, tetrahedral, hexahedral, prismatic as well as mixed meshes, surface meshes and topologically periodic meshes. It has general support for mesh refinement and optimization including local conforming and non-conforming adaptive mesh refinement (AMR) with arbitrary-order hanging nodes, powerful node-movement mesh optimization, anisotropic refinement, derefinement, and parallel load balancing . Arbitrary element transformations allowing for high-order mesh elements with curved boundaries are also supported. Some commonly used linear solvers, nonlinear methods, eigensolvers, and a variety of explicit and implicit Runge-Kutta time integrators are also available.

MFEM supports Message Passing Interface (MPI)-based parallelism throughout the library and can readily be used as a scalable unstructured finite element problem generator. Starting with version 4.0, MFEM offers initial support for GPU acceleration, and programming models, such as CUDA, OCCA, RAJA and OpenMP. MFEM-based applications have been scaled to hundreds of thousands of cores. The library supports efficient operator partial assembly and evaluation for tensor-product high-order elements. A serial MFEM application typically requires minimal changes to transition to a scalable parallel version of the code where it can take advantage of the integrated scalable linear solvers from the *hypre* library, including the BoomerAMG, AMS, and ADS solvers. Both the serial and parallel versions can make use of high-performance, *partial assembly* kernels.

MFEM's development grew out of a need for robust, flexible, and efficient simulation algorithms for physics and engineering applications at Lawrence Livermore National Laboratory (LLNL). The initial open-source release of the library was in 2010, followed by version 1.2 in 2011 that added MPI parallelism. Versions 2.0, 3.0 and 3.4 released in 2011, 2015 and 2018 added new features such as arbitrary high-order spaces, non-conforming AMR, HPC miniapps and mesh optimization. An important milestone was the initial GPU support added in MFEM-4.0, which was released in May 2019. The latest version is 4.1, released in March 2020.

A comprehensive list of publications making use of MFEM can be found at `https://mfem.org/publications`.

4.2.2 Partial and Full Assembly

Finite element operators are typically defined through weak formulations of partial differential equations that involve integration over a computational mesh. The required integrals are computed by splitting them as a sum over the mesh elements, mapping each element to a simple reference element (e.g. the unit square) and applying a quadrature rule in reference space.

This sequence of operations highlights an inherent hierarchical structure present in all finite element operators where the evaluation starts on *global (trial) degrees of freedom* on the whole mesh, restricts to *degrees of freedom on subdomains* (groups of elements), then moves to independent *degrees of freedom on each element*, transitions to independent *quadrature points* in reference space, performs the integration, and then goes back in reverse order to global (test) degrees of freedom on the whole mesh.

$$\boxed{A = P^T G^T B^T D B G P}$$

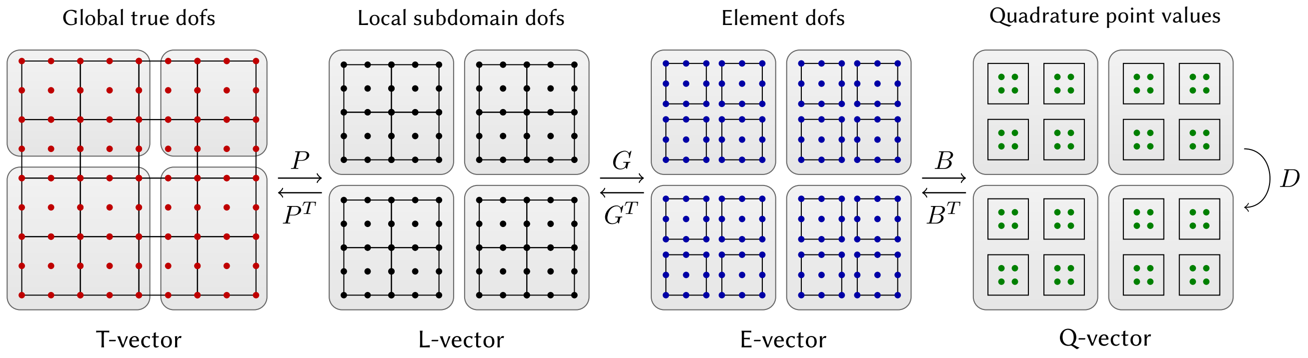


Figure 4.2: Fundamental finite element operator decomposition. This algebraically factored form is a much better description than a global sparse matrix for high-order methods and is easy to incorporate in a wide variety of applications. See also the libCEED library in [106].

This is illustrated in Figure 4.2 for the simple case of a symmetric linear operator on second order (Q_2) scalar continuous (H^1) elements, where we use the notions **T-vector** (true vector), **L-vector** (local vector), **E-vector** (element vector) and **Q-vector** (quadrature vector) to represent the sets corresponding to the (true) degrees of freedom on the global mesh, the split local degrees of freedom on the subdomains, the split degrees of freedom on the mesh elements, and the values at quadrature points, respectively. Note that class `(Par)GridFunction` represents an L-vector, and T-vector is typically represented by either `HypreParVector` or `Vector`. We remark that although the decomposition presented in Figure 4.2 is appropriate for square, symmetric linear operators, the generalization of this finite element decomposition to rectangular and nonlinear operators is straightforward.

One of the challenges with high-order methods is that a global sparse matrix is no longer a good representation of a high-order linear operator, both with respect to the FLOPs needed for its evaluation [107], as well as the memory transfer needed for a matrix-vector product (matvec) [108, 109]. Thus, high-order methods require a new "format" that still represents a linear (or more generally, nonlinear) operator, but not through a sparse matrix.

We refer to the operators that connect the different types of vectors as:

- Subdomain restriction P.
- Element restriction G.
- Basis (DOFs to quadrature points) evaluator B.
- Operator at quadrature points D.

More generally, when the test and trial space differ, each space has its own versions of P, G and B.

Note that in the case of adaptive mesh refinement (AMR), the restriction P will involve not just extracting sub-vectors, but evaluating values at constrained degrees of freedom through the AMR interpolation. There can also be several levels of subdomains (P_1, P_2, etc.), and it may be convenient to split D as the product of several operators (D_1, D_2, etc.).

After the application of each of the first three transition operators, P, G and B, the operator evaluation is decoupled on their ranges, so P, G and B allow us to "zoom-in" to subdomain, element, and quadrature point level, ignoring the coupling at higher levels. Thus, a natural mapping of A on a parallel computer is to split the **T-vector** over MPI ranks in a non-overlapping decomposition, as is typically used for sparse matrices, and then split the rest of the vector types over computational devices (CPUs, GPUs, etc.) as indicated by the shaded regions in Figure 4.2.

One of the advantages of the decomposition perspective in these settings is that the operators P, G, B and D clearly separate the MPI parallelism in the operator (P) from the unstructured mesh topology (G), the choice of the finite element space/basis (B) and the geometry and point-wise physics D. These components also naturally fall in different classes of numerical algorithms – parallel (multi-device) linear algebra for P, sparse (on-device) linear algebra for G, dense/structured linear algebra (tensor contractions) for B and parallel point-wise evaluations for D.

Since the global operator A is just a series of variational restrictions (i.e. transformations $Y \to X^T YX$) with B, G and P, starting from its point-wise kernel D, a matrix-vector product with A can be performed by evaluating and storing some of the innermost variational restriction matrices, and applying the rest of the operators "on-the-fly". For example, one can compute and store a global matrix on the **T-vector** level. Alternatively, one can compute and store only the subdomain (**L-vector**) or element (**E-vector**) matrices and perform the action of A using matvecs with P or P and G. While these options are natural for low-order discretizations, they are not a good fit for high-order methods due to the amount of FLOPs needed for their evaluation, as well as the memory transfer needed for a matvec.

Much higher performance can be achieved by the use of *partial assembly* algorithms, as described in the following section. In this case, we compute and store only D (or portions of it) and evaluate the actions of P, G and B on-the-fly. Critically for performance, we take advantage of the tensor-product structure of the degrees of freedom and quadrature points on quadrilateral and hexahedral elements to perform the action of B without storing it as a matrix. Implemented properly, the partial assembly algorithm requires the optimal amount of memory transfers (with respect to the polynomial order) and near-optimal FLOPs for operator evaluation. It consists of an operator *setup* phase, that evaluates and stores D and an operator *apply* (evaluation) phase that computes the action of A on an input vector. When desired, the setup phase may be done as a side-effect of evaluating a different operator, such as a nonlinear residual. The relative costs of the setup and apply phases are different depending on the physics being expressed and the representation of D.

In the traditional finite element setting, the operator is assembled in the form of a matrix. The action of the operator is computed by multiplying with this matrix. At high orders this requires both a large amount of memory to store the matrix, as well as many floating point operations to compute and apply it. By exploiting the finite element operator decomposition structure as well as the basis functions structure, there are options for creating operators that require much less storage and scale better at high orders. This section introduces partial assembly and sum factorization [107, 101], which reduce both the assembly storage and number of floating point operations required to apply the operator, and discusses general algorithm opportunities and challenges in the MFEM code.

Removing the finite element space restriction operator from the assembly for domain-based operators[1] yields the element-local matrices at the **E-vector** level. This storage can lead to faster data access, since the block is stored contiguously in memory, and applications of the block can be designed to maximally use the cache.

Partial assembly operates at the **Q-vector** level, after additionally removing the basis functions and gradients, B, from the assembled operator. This leaves only the D operator to store for every element. This by itself reduces the storage but not the number of floating point operations required for evaluation. As will be discussed later, this is key to offloading the operator action to a co-processor that may have less memory.

[1]Domain-based operators correspond to bilinear forms which use integrals over the problem domain, as opposed to its boundary, for example.

As an illustration of partial assembly, consider the decomposition of the mass matrix evaluated on a single element E

$$(M_E)_{ij} = \int_E \rho\, \varphi_j \varphi_i \, dx \tag{4.1}$$

where ρ is a given density coefficient and $\{\varphi_i\}$ are the finite element basis functions on the element E. Changing the variables in the integral from E to the reference element $\hat{E}$ and applying a quadrature rule with points $\{\hat{x}_k\}$ and weights $\{\alpha_k\}$ yields

$$(M_E)_{ij} = \sum_k \alpha_k \, (\rho \circ \Phi)\,(\hat{x}_k)\, \hat{\varphi}_j(\hat{x}_k) \hat{\varphi}_i(\hat{x}_k)\, \det(J(\hat{x}_k)). \tag{4.2}$$

In the last expression, Φ is the mapping from the reference element $\hat{E}$ to the physical element E, J is its Jacobian matrix, and $\{\hat{\varphi}_i\}$ are the finite element basis functions on the reference element. Defining the matrix B of basis functions evaluated at quadrature points as $B_{ki} = \hat{\varphi}_i(\hat{x}_k)$, the above equation can be rewritten as

$$(M_E)_{ij} = \sum_k B_{ki} (D_E)_{kk} B_{kj}, \qquad \text{where} \quad (D_E)_{kk} = \alpha_k \det(J(\hat{x}_k))\,(\rho \circ \Phi)\,(\hat{x}_k), \quad (D_E)_{kl} = 0,\ k \neq l. \tag{4.3}$$

Using this definition, the matrix operator can be written simply as $M_E = B^t D_E B$. Matrix-vector evaluations are computed as the series of products by B, D_E, and B^t without explicitly forming M_E.

For general B, its application requires the same order of floating point operations as applying the fully-assembled M_E matrix: $\mathcal{O}(p^{2d})$ (assuming that the number of quadrature points is $\mathcal{O}(p^d)$). Taking advantage of the tensor-product structure of the basis functions and quadrature points on quad and hex elements, B_{ki} can be written as

$$B_{ki} = \hat{\varphi}^{1d}_{i_1}\left(\hat{x}^{1d}_{k_1}\right) \dots \hat{\varphi}^{1d}_{i_d}\left(\hat{x}^{1d}_{k_d}\right), \qquad k = (k_1, \dots, k_d), \quad i = (i_1, \dots, i_d) \tag{4.4}$$

with d the number of dimensions. In this case the matrix B itself is decomposed as a tensor product of smaller one-dimensional matrices $B^{1d}_{lj} = \hat{\varphi}^{1d}_j\left(\hat{x}^{1d}_l\right)$ so that

$$B_{ki} = B^{1d}_{k_1 i_1} \dots B^{1d}_{k_d i_d}. \tag{4.5}$$

Applying the series of B^{1d} matrices reduces the overall number of floating point operations when applying M_E to $\mathcal{O}(p^{d+1})$ (assuming that the number of 1D quadrature points is $\mathcal{O}(p)$). This evaluation strategy is often referred to as *sum factorization*.

To make this point concrete, consider the application of a quad basis to a vector v for interpolation at a tensor product of quadrature points. Without taking advantage of the structure of the basis, the product takes the form

$$(Bv)_k = \sum_i B_{ki} v_i = \sum_i \hat{\varphi}_i(\hat{x}_k) v_i, \tag{4.6}$$

which requires $\mathcal{O}(p^{2d})$ $(d = 2)$ storage and operations for the matrix-vector product. When using the alternative form (4.5) the operation can be rewritten as

$$(Bv)_k = \sum_i B_{ki} v_i = \sum_{i_1, i_2} B^{1d}_{k_1 i_1} B^{1d}_{k_2 i_2} V_{i_1 i_2} = \left[B^{1d} V \left(B^{1d}\right)^t \right]_{k_1 k_2}, \tag{4.7}$$

where V is the vector v viewed as a square matrix: $V_{i_1 i_2} = v_i$. This highlights an interesting aspect of sum factorization: with each smaller matrix product with B^{1d}, an additional axis is converted from basis (i_j) to quadrature (k_j) indices. The same reasoning can also be applied to three spatial dimensions. Using the sum factorization approach, the storage was reduced to $\mathcal{O}(p^d)$ and the number of operations to $\mathcal{O}(p^{d+1})$.

Choosing to store the partially assembled operator instead of the full matrix affects the solvers that can be used, since the full matrix is not available to be queried. This means for instance that traditional multigrid solvers are difficult to apply.

The storage and asymptotic number of floating point operations required for assembly and evaluation using the different methods are recorded in Table 4.1. Sum factorization can be utilized to reduce the cost of assembling the local element matrices and thus the cost of full assembly (**T-vector** level) – this is shown in the second row of the table. Furthermore, partial assembly has improved the asymptotic scaling for high orders in both storage and number of floating point operations for assembly and evaluation. Therefore, partial assembly is well-suited for high orders.

There are many opportunities and challenges for parallelization with partial assembly using sum factorization. At the **E-vector** level the products can be applied independently for every element in parallel, which makes partial assembly with sum factorization a promising portion of the finite element algorithm to offload to co-processors, such as GPUs. In MFEM, partial assembly and sum factorization are implemented in the bilinear and nonlinear form integrators themselves. Specifically, in the base class `BilinearFormIntegrator`, the assembly and evaluation are performed by the virtual methods `AssemblePA` and `AddMultPA`, respectively. MFEM supports partially assembly for the entire de Rham complex, including H^1,H(Curl),H(Div), and L^2 spaces. MFEM currently supports partial assembly for tensor-product elements (quadrilaterals and hexahedra), for which sum factorization is most efficient. Partial assembly on simplices and mixed meshes are partially supported through MFEM's integration with the libCEED library. In the case of simplices, sum factorization cannot be used for the evaluation of the action of the B operator, however, other efficient algorithms exist, for example using the Bernstein basis [110].

4.2.3 De Rham Cohomology

The *de Rham complex* [111, 112] is a compatible multi-physics discretization framework that naturally connects the solution spaces for many common PDEs. It is illustrated in Figure 4.3. The finite element method provides a compatible approach to preserve the de Rham complex properties on a fully discrete level. In MFEM, constructing a `FiniteElementSpace` using the `*_FECollection` with `*` replaced by `H1`, `ND`, `RT`, or `L2`, creates the compatible discrete finite element space for the continuous H^1, H(Curl), H(Div), or L^2 space, respectively. Note that the order of the space is simply a parameter in the constructor of the respective `*_FECollection`, see Figure 4.1.

The finite element spaces in the de Rham sequence are the natural discretization choices respectively for: kinematic variables (e.g., position, velocity), electromagnetic fields (e.g., electric field in magnetohydrodynamics (MHD)), diffusion fluxes (e.g., in flux-based radiation-diffusion) and thermodynamic quantities (e.g., internal energy, density, pressure). MFEM includes full support for the de Rham complex at arbitrary high order, on arbitrary order meshes.

4.2.4 HO to LOR to HO

Let H and L be finite dimensional subspaces of a larger Hilbert space U, with corresponding inner products $(\cdot,\cdot)_H$, $(\cdot,\cdot)_L$ and $(\cdot,\cdot)_U$. Denote with $L^\perp$ the orthogonal complement of L with with respect to $(\cdot,\cdot)_U$ in U. We are interested in linear maps between H and L that are accurate (e.g. preserve constants, recover functions) and conservative (e.g. preserve integrals).

Theorem 2. *Assume that L is large enough so $H \cap L^\perp = \{0\}$, and in particular $dim(H) \le dim(L)$. Define $R : H \mapsto L$ and $P : L \mapsto H$ by*

$$(Ru_H, v_L)_L = (u_H, v_L)_U \qquad \forall u_H \in H, v_L \in L,$$

Method	**Storage**	**Assembly**	**Evaluation**
Traditional full assembly + matvec	$\mathcal{O}(p^{2d})$	$\mathcal{O}(p^{3d})$	$\mathcal{O}(p^{2d})$
Sum factorized full assembly + matvec	$\mathcal{O}(p^{2d})$	$\mathcal{O}(p^{2d+1})$	$\mathcal{O}(p^{2d})$
Partial assembly + matrix-free action	$\mathcal{O}(p^{d})$	$\mathcal{O}(p^{d})$	$\mathcal{O}(p^{d+1})$

Table 4.1: Comparison of storage and Assembly/Evaluation FLOPs required for full and partial assembly algorithms on tensor-product element meshes (quadrilaterals and hexahedra). Here, p represents the polynomial order of the basis functions and d represents the number of spatial dimensions. The number of DOFs on each element is $\mathcal{O}(p^d)$ so the "sum factorization full assembly" and "partial assembly" algorithms are nearly optimal.

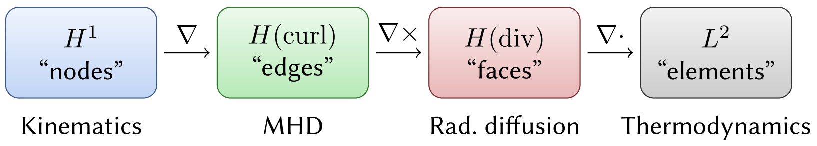
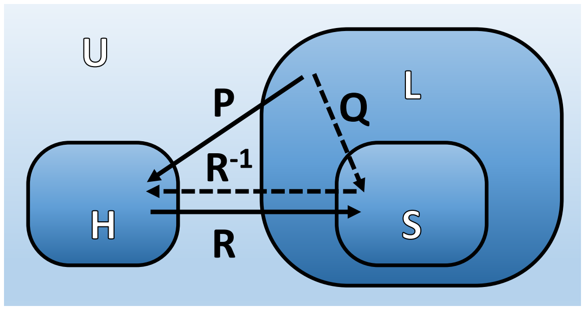


Figure 4.3: Continuous de Rham complex in 3D and example physical fields that can be represented in the respective spaces.

and

$$(Pv_L, Ru_H)_U = (v_L, Ru_H)_L \qquad \forall v_L \in L, u_H \in H.$$

Set $S = R(H) \subseteq L$ and $Q = RP$.



Then, we have the following properties:

1. *R is injective, P is surjective, and $R : H \mapsto S$ is a bijection*
2. *$PR : H \mapsto H$ is the identity operator (i.e. P is a left inverse of R)*
3. *$Q : L \mapsto S$ is a projection, and $P = R|_S^{-1}Q$*
4. *For any function $1_L \in L$ and $1_S \in S$ we have the conservation properties:*

$$(Ru_H, 1_L)_L = (u_H, 1_L)_U \quad \text{and} \quad (Pv_L, 1_S)_U = (v_L, 1_S)_L\,.$$

Proof. Let $\{\phi_k^H\}$ and $\{\psi_k^L\}$ be fixed bases in H and L, and let M_L and M_{HL} be the matrices

$$(M_L)_{ij} = (\phi_i^L, \phi_j^L)_L \quad \text{and} \quad (M_{HL})_{ij} = (\phi_i^H, \phi_j^L)_U\,.$$

Them M_L is invertible as a Gram matrix, and the matrix representations of the operators R is

$$R = M_L^{-1} M_{HL}^T.$$

The assumption $H \cap L^\perp = \{0\}$ implies that R is injective, so $R^T M_L R$ is also invertible and the operator P is well defined with matrix representation

$$P = (R^T M_L R)^{-1} R^T M_L\,,$$

so $PR = I$ and therefore P is surjective. The definitions of P and R imply

$$(RPv_L, Ru_H)_L = (v_L, Ru_H)_L \qquad \forall v_L \in L, u_H \in H\,,$$

which is the same as the third property:

$$(Qv_L, w_S)_L = (v_L, w_S)_L \qquad \forall v_L \in L, w_S \in S\,.$$

Finally, the conservation properties follows simply by setting $v_L = 1_L$ and $Ru_H = 1_S$ in the operator definitions. Note that R has stronger conservation properties, since L is larger than S. □

Example 1 (HO/LOR transfer)**.** *Let H and L be a high-order (HO) and a corresponding low-order refined (LOR) space on the same **linear** quad/hex mesh. We allow the HO space to be continuous or discontinuous, but the LOR space is assumed to be discontinuous of order zero (p.w. constant).*

Then, we can ensure the $H \cap L^{\perp} = \{0\}$ property by picking the LOR space to have the same number of degrees of freedom as the HO space on each element. This is because $(e_H, \psi_k^L) = 0$ implies, by the mean value theorem, that e_H has a zero in the LOR element corresponding to ψ_k^L.

Let $U = L^2(\Omega)$ and all inner products be just the regular L^2 inner product. Then that the constant function is in the range of R because $R1_H = 1_L$, which further implies $P1_L = 1_H$.

Therefore, if R and P are used to map between $u_H \in H$ and $u_L \in L$, we have accuracy in the sense that constants are preserved and HO functions are recovered by P and mass conservation in the sense that

$$\int_\Omega u_L = \int_\Omega u_H \,.$$

Example 2 (Conservation of multiple fields)**.** *In the setting of the previous example, let the inner products be given in terms of the weights ρ_L and ρ_H:*

$$(u_L, v_L)_L = \int_\Omega \rho_L u_L v_L \,, \quad (u_H, v_L)_U = \int_\Omega \rho_H u_H v_L \,. \tag{4.8}$$

Assume that ρ_L and ρ_H represent previously mapped density field on the same LOR mesh with the same LOR space, which means that we have the conservation property

$$\int_\Omega \rho_L v_L = \int_\Omega \rho_H v_L$$

for any $v_L \in L$. Then that the constant function is still in the range of R and we still have $R1_H = 1_L$. Therefore, we have the density-weighted conservation:

$$\int_\Omega \rho_L u_L = \int_\Omega \rho_H u_H \,.$$

This implies that we can transfer density and velocity while preserving both mass and momentum.

To illustrate this technique, we consider a high-order discontinuous space, H_ρ and a high-order continuous space H_u. The high-order density and velocity fields will be defined in these spaces, i.e. $\rho_H \in H_\rho$ and $u_H \in H_u$. For simplicity, we only consider one discontinuous space L, such that $\rho_L, u_L \in L$. Let R denote the standard (unweighted) restriction operator. Then, we can compute $\rho_L = R\rho_H$, which requires only the local inversion of the block diagonal mass matrix defined on L. Once ρ_L is computed, we can compute the weighted restriction operator R_ρ, defined in terms of the weighted inner products given in (4.8)*. This allows us to compute $u_L = R_\rho u_H$, which requires the local inversion of the block diagonal ρ_L-weighted mass matrix. These projections are mass and momentum conserving, as shown above.*

Table 4.2: Mass and momentum conservation mapping from H to L

Mass (high-order)	14.38082468
Mass (low-order)	14.38082468
Mass conservation error	1.98×10^{-15}
Momentum (high-order)	0.39239862
Momentum (low-order)	0.39239862
Momentum conservation error	8.49×10^{-16}

Suppose now that we are given ρ_L and u_L in the space L and we wish to compute $\rho_H \in H_\rho$ and $u_H \in H_u$. We first compute $\rho_H = P\rho_L$, where P is the left-inverse of R. Since the spaces L and H_ρ are discontinuous, the matrices M_L and $R^T M_L R$ are block *diagonal, and thus can be inverted locally. Therefore, $P\rho_L$ can be computed element-by-element. On the other hand, since H_u is a continuous space, computing the corresponding high-order function u_H requires a global, coupled solve. Let P_ρ denote the left-inverse to R_ρ defined by $P_\rho = (R_\rho^T M_L R_\rho)^{-1} R_\rho^T M_L$. The matrix $R_\rho^T M_L R_\rho$ is no longer block-diagonal (as in the discontinuous case), and therefore $P_\rho u_L$ cannot be computed element-by-element. To solve the resulting system, we can use a preconditioned conjugate gradient iteration with diagonal preconditioning.*

Table 4.3: Mass and momentum conservation mapping from L to H

Mass (high-order)	14.38176887
Mass (low-order)	14.38176887
Mass conservation error	8.65×10^{-16}
Momentum (high-order)	0.3927651071
Momentum (low-order)	0.3927651071
Momentum conservation error	5.67×10^{-13}

Sample high-order and low-order density and velocity fields obtained using this approach are shown below. The high-order space uses degree-4 polynomials, and the low-order space uses piecewise constants defined on the low-order refined mesh obtained from the high-order Gauss-Lobatto points.

Example 3 (Curved meshes)**.** *In the setting of Example 1, assume that the HO mesh is curved while the LOR mesh is linear. All properties considered are valid on the LOR space that uses curved LOR elements (mapped with the HO mapping) so the only approximation is in the transition of p.w. constant fields from the curved to the linear LOR elements. The areas/volumes of these elements are different, so we have to pick between accuracy (preservation of constant) and conservation (mass conservation). Our default is to choose the former which introduces a second order error in the geometry (we are approximating a curve with a linear function) and so we introduce a second-order error in the mass conservation.*

Example 4 (AMR derefinement)**.** *Let H be a coarse and L be a refined space in adaptive mesh refinement. Since $H \subseteq L$, R is just the natural injection and $H \cap L^\perp = \{0\}$ is straightforward. The operator P in this case can be used to transfer functions when coarsening.*

Example 5 (LOR preconditioning)**.** *We use HO and LOR spaces also in the context of LOR preconditioning, but note that there the LOR space is of the same size or smaller than the HO spaces and the transfer operators can be very simple, e.g. the identity.*

Accuracy of the prolongation

Suppose we have a function $f \in U$ such that $f = f_H + g$, with $f_H \in H$ and $\|g\| = \mathscr{O}(h^{p+1})$ for some p. Let $f_L = \Pi_L f$ be the projection of f onto L, i.e. f_L satisfies

$$(f_L, w_L) = (f, w_L) \quad \text{for all } w_L \in L.$$

We are interested in the accuracy of Pf_L compared with f_H.

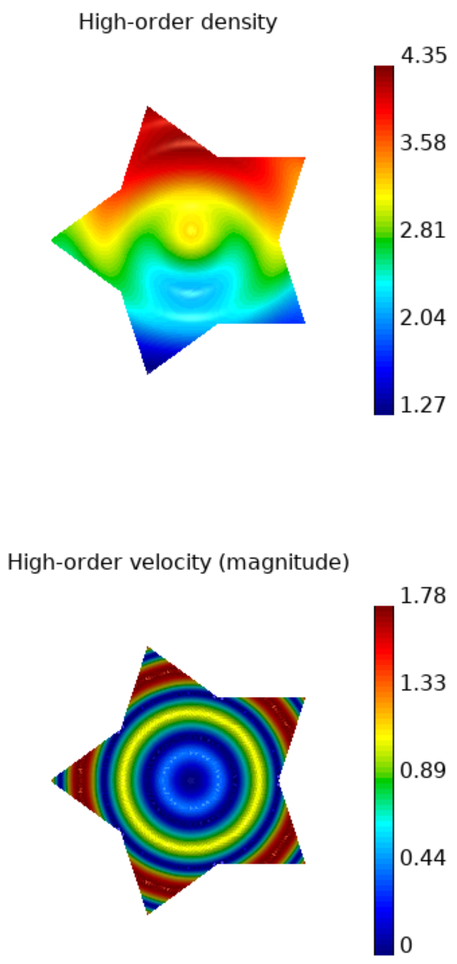
First, note that $\Pi_L f = \Pi_L f_H + \Pi_L g$, and so $Pf_L = P\Pi_L f_H + P\Pi_L g$. The term $P\Pi_L f_H$ is defined as the unique member of H such that

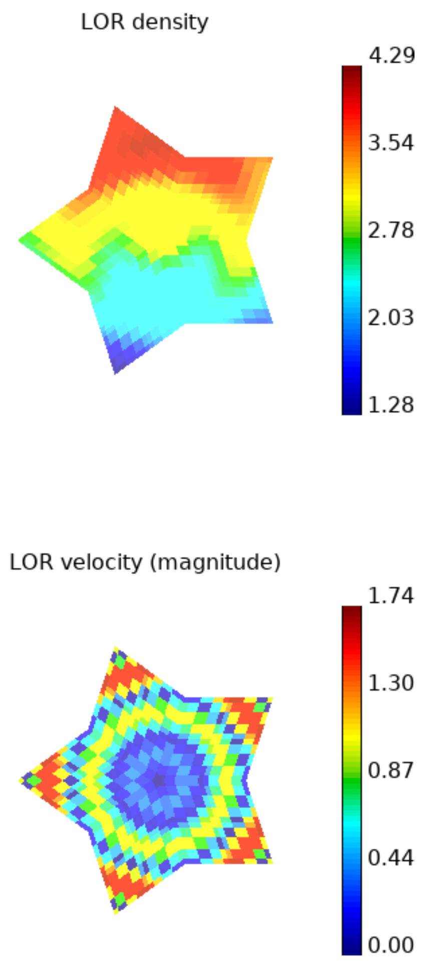
$$(P\Pi_L f_H, Rv_H) = (\Pi_L f_H, Rv_H) \quad \text{for all } v_H \in H.$$

By definition of the projection Π_L, we have

$$(\Pi_L f_H, Rv_H) = (f_H, Rv_H) \quad \text{for all } v_H \in H,$$

and so we see that $P\Pi_L f_H = f_H$. This can also be seen using the fact that $\Pi_L|_H = R$, and $PR = I$ from Theorem 2.





Therefore, $Pf_L = f_H + P\Pi_L g$. We have

$$\|Pf_L - f\| = \|P\Pi_L g - g\| \leq \|g\| + \|P\Pi_L g\|, \tag{4.9}$$

and so we now wish to estimate $\|P\Pi_L g\|$. Note that since Π_L is a projection, we have that $\|\Pi_L g\| \leq \|g\| = \mathscr{O}(h^{p+1})$. We now estimate the operator norm of P.

$$\begin{aligned} \|RPu_L\|^2 &= (RPu_L, RPu_L) \\ &= (Pu_L, RPu_L) && \text{by definition of } R \\ &= (u_L, RPu_L) && \text{by definition of } P \\ &\leq \|u_L\| \|RPu_L\|. \end{aligned}$$

Therefore, $\|RPu_L\| \leq \|u_L\|$. Since the operator R is injective, we have $\|Rv\| \geq \alpha \|v\|$ for some α, and hence

$$\|Pu_L\| \leq \frac{1}{\alpha} \|u_L\|.$$

Combining this estimate with (4.9), we have

$$\|Pf_L - f\| \le \|g\| + \|P\Pi_L g\| \le \|g\| + \frac{1}{\alpha}\|g\| = \mathscr{O}(\alpha^{-1}h^{p+1}).$$

It remains to provide an estimate for α (and in particular, to quantify its dependence on the polynomial degree p).

Lemma 1. *Each point in the Gauss-Legendre quadrature rule of length n lies in exactly one subinterval generated by the Gauss-Lobatto quadrature rule of length $n+1$.*

Proof. Let $P_n(x)$ denote the Legendre polynomial of degree n. Then, the Gauss-Legendre quadrature rule of length n is given by the roots of $P_n(x)$. The Gauss-Lobatto quadrature rule of length $n+1$ is given by the roots of $(x^2-1)P_n'(x)$.

Consider two successive Gauss-Legendre points, x_i and x_{i+1}. We have $P_n(x_i) = P_n(x_{i+1}) = 0$, and so, by Rolle's theorem, $P_n'(\xi_i) = 0$ for some $\xi_i \in (x_i, x_{i+1})$. Therefore, a Gauss-Lobatto point ξ_i lies in between the Gauss-Legendre points x_i and x_{i+1}. The result follows by noting that $-1 < x_i < 1$ for all i. □

4.2.5 A brief tour through some code features / capabilities

Mesh Optimization via the Target-Matrix Optimization Paradigm (TMOP)

A vital component of high-order methods is the use of high-order representation not just for the *physics* fields, but also for the geometry, represented by a high-order computational mesh. High-order meshes can be relatively coarse and still capture curved geometries with high-resolution, leading to equivalent simulation quality for a smaller number of elements. High-order meshes can also be very beneficial in a wide range of applications, where e.g. radial symmetry preservation, or alignment with physics flow or curved model boundary is important [113, 114, 115]. Such applications can utilize *static* meshes, where a good-quality high-order mesh needs to be generated only as an input to the simulation, or *dynamic* meshes, where the mesh evolves with the problem (e.g. following the motion of a material) and its quality needs to be constantly controlled. In both cases, the quality of high-order meshes can be difficult to control, because their properties vary in space on a sub-zonal level. Such control is critical in practice, as poor mesh quality leads to small time step restrictions or simulation failures.

The MFEM project has developed a general framework for the optimization of high-order curved meshes based on the node-movement techniques of the Target-Matrix Optimization Paradigm (TMOP) [116, 117]. This enables applications to have precise control over local mesh quality, while still optimizing the mesh globally. Note that while our new methods are targeting high-order meshes, they are general, and can also be applied to low-order mesh applications that use linear meshes.

TMOP is a general approach for controlling mesh quality, where mesh nodes (vertices in the low-order case) are moved so-as to optimize a multi-variable objective function that quantifies global mesh quality. Specifically, at a given point of interest (inside each mesh element), TMOP uses three Jacobian matrices:

- The Jacobian matrix $A_{d\times d}$ of the transformation from reference to physical coordinates, where d is the space dimension.
- The *target matrix*, $W_{d\times d}$, which is the Jacobian of the transformation from the reference to the *target* coordinates. The target matrices are defined according to a user-specified method prior to the optimization; they define the desired properties in the optimal mesh.
- The *weighted Jacobian* matrix, $T_{d\times d}$, defined by $T = AW^{-1}$, represents the Jacobian of the transformation between the target and the physical (current) coordinates.

The T matrix is used to define the *local quality measure*, $\mu(T)$. The quality measure can evaluate shape, size, or alignment of the region around the point of interest. The combination of targets and quality metrics is used to optimize the node positions, so that they are as close as possible to the shape/size/alignment of their targets. This is achieved by minimizing a global *objective function*, $F(x)$, that depends on the local quality measure throughout the mesh:

$$F(x) := \sum_{E\in\mathscr{E}} \int_{E_t} \mu(T(x))dx_t = \sum_{E\in\mathscr{E}} \sum_{x_q\in Q_E} w_q \det(W(x_q))\, \mu(T(x_q)), \tag{4.10}$$

where E_t is the target element corresponding to the physical element E, Q_E is the set of quadrature points for element E, w_q are the corresponding quadrature weights, and both $T(x_q)$ and $W(x_q)$ are evaluated at the quadrature point x_q of element E. The objective function can be extended by using combinations of quality metrics, space-dependent weights for each metric, and limiting the amount of allowed mesh displacements. As $F(x)$ is nonlinear, MFEM utilizes Newton's method to solve the critical point equations, $\partial F(x)/\partial x = 0$, where x is the vector that contains the current mesh positions. This approach involves the computation of the first and second derivatives of $\mu(T)$ with respect to T. Furthermore, boundary nodes are enforced to stay fixed or move only in the boundary's tangential direction. Additional modifications are performed to guarantee that the Newton updates do not lead to inverted meshes, see [117].

The current MFEM interface provides access to 12 two-dimensional mesh quality metrics, 7 three-dimensional metrics, and 5 target construction methods, together with the first and second derivatives of each metric with respect to the matrix argument. The quality metrics are defined by the inheritors of the class `TMOP_QualityMetric`, and target construction methods are defined by the class `TargetConstructor`. MFEM supports the computation of matrix invariants and their first and second derivatives (with respect to the matrix), which are then used by the `NewtonSolver` class to solve $\partial F(x)/\partial x = 0$. The library interface allows users to choose between various options concerning target construction methods and mesh quality metrics and adjust various parameters depending on their particular problem. The mesh optimization module can be easily extended by additional mesh quality metrics and target construction methods. Illustrative examples are presented in the form of a simple mesh optimization miniapp, `mesh-optimizer`, in the `miniapps/meshing` directory, which includes both serial and parallel implementations. Some examples of simulations that can be performed by this miniapp are shown in Figure 4.4.

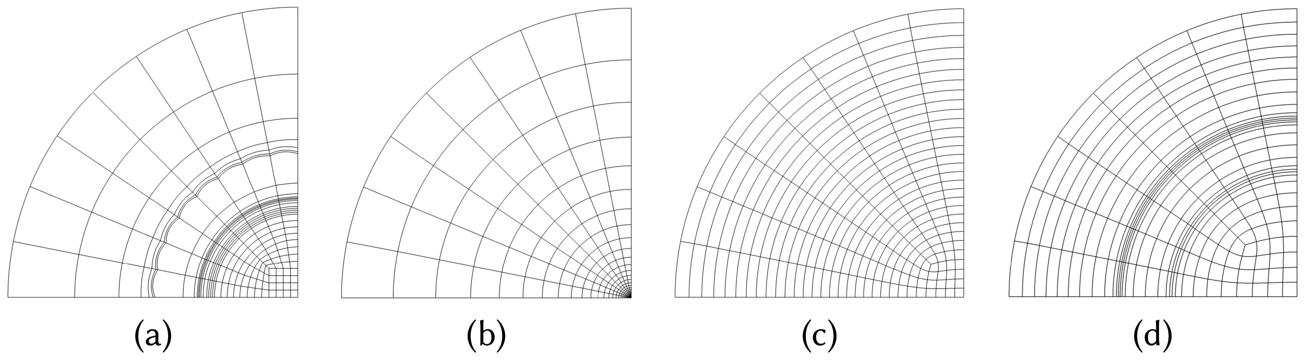


Figure 4.4: A perturbed fourth order 2D mesh (a) is being optimized by targeting shape-only optimization (b), shape and equal size (c), and finally shape and space-dependent size (d).

Work to extend MFEM's mesh optimization capabilities to simulation-driven adaptivity (a.k.a. *r*-adaptivity) [118], and coupling *h*- and *r*-adaptivity of high-order meshes by combining the TMOP and AMR concepts is ongoing. See Figure 4.5 for some preliminary results in that direction.

Non-Conforming Adaptive Mesh Refinement (AMR)

Many high-order applications can be enriched by parallel adaptive mesh refinement (AMR) on unstructured quadrilateral and hexahedral meshes. Quadrilateral and hexahedral elements are attractive for their tensor product structure (enabling efficiency) and for their refinement flexibility (enabling e.g., *anisotropic* refinement). However, as opposed to the bisection-based methods for simplices considered in the previous section, *hanging* nodes that occur after local refinement of quadrilaterals and hexahedra

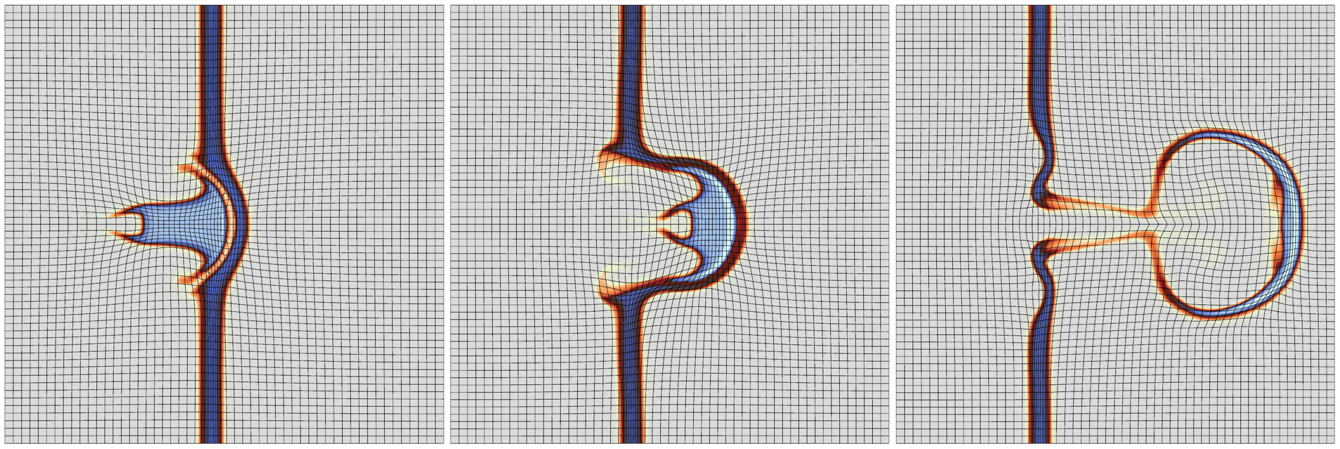


Figure 4.5: Example of MFEM *r*-adaptivity to align the mesh with materials in a multi-material ALE simulation of high velocity gas impact, cf. [119]. Time evolution of the materials and mesh positions at times 2.5 (left), 5 (center), and 10 (right). See [118] for details.

are not easily avoided by further refinement [120, 121, 122]. We are thus interested in *non-conforming* (irregular) meshes, in which adjacent elements need not share a complete face or edge and where some finite element degrees of freedom (DOFs) need to be constrained to obtain a conforming solution.

In this section we review MFEM's software abstractions and algorithms for handling parallel non-conforming meshes on a general discretization level, independent of the physics simulation. These methods support the entire de Rham sequence of finite element spaces, at arbitrarily high-order, and can support high-order curved meshes, as well as finite element techniques such as hybridization and static condensation. They are also highly scalable, easy to incorporate into existing codes, and can be applied to complex, anisotropic 3D meshes with arbitrary levels of non-conforming refinement. While MFEM's approaches can be extended to general *hp*-refinement, the current implementation focuses exclusively on non-conforming *h*-refinement with fixed polynomial degree.

These approaches are based on a variational restriction approach to AMR, described below. For more details, see [123]. Consider a weak variational formulation with a bilinear form $a(\cdot,\cdot)$ is symmetric. To discretize the problem, we cover the computational domain Ω with a mesh consisting of mutually disjoint elements K_i, their vertices V_j, edges E_m, and faces F_n. Except for the vertices, we consider these entities as open sets, so that $\Omega = (\cup_i K_i) \cup (\cup_j V_j) \cup (\cup_m E_m) \cup (\cup_n F_n)$. In the case of non-conforming meshes, there exist faces F_s that are strict subsets of other faces, $F_s \subsetneq F_m$, see Figure 4.6. We call F_s *slave faces* and F_m *master faces*. The remaining standard faces F_c are disjoint with all other faces and will be referred to as *conforming faces*. Similarly, we define *slave edges*, *master edges* and *conforming edges*.

Non-conforming meshes in MFEM are represented by the `NCMesh` and `ParNCMesh` classes. We use a tree-based data structure to represent refinements which has been optimized to rely only on the following information: 1) elements contain indices of eight vertices, or indices of eight child elements if refined; 2) edges are identified by pairs of vertices; 3) faces are identified by four vertices. Edges and faces are tracked by associative maps (see below), which reduce both code complexity and memory footprint. In the case of a uniform hexahedral mesh, our data structure requires about 290 bytes per element, counting the complete refinement hierarchy and including vertices, edges, and faces.

To construct a standard finite dimensional FEM approximation space $V_h \subset V$ on a given non-conforming mesh, we must ensure that the necessary conformity requirements are met between the slave and master faces and edges so that we get V_h that is a (proper) subspace of V. For example, if V is the Sobolev space H^1, the solution values in V_h must be kept continuous across the non-conforming interfaces. In contrast, if V is an $H(Curl)$ space, the tangential component of the finite element vector fields in V_h needs to be continuous across element faces. More generally, the conformity requirement can be expressed by requiring that values of V_h functions on the slave faces (edges) are interpolated from the finite element function values on their master faces (edges). Finite element degrees of freedom on the slave faces (and edges) are thus effectively constrained and can be expressed as

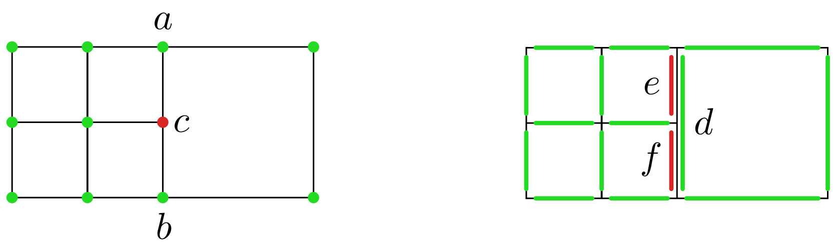


Figure 4.6: Illustration of conformity constraints for lowest order nodal elements in 2D. Left: Nodal elements (subspace of H^1), constraint $c = (a+b)/2$. Right: Nedelec elements (subspace of $H(Curl)$, constraints $e = f = d/2$. In all cases, fine degrees of freedom on a coarse-fine interface are linearly interpolated from the values at coarse degrees of freedom on that interface.

linear combinations of the remaining degrees of freedom. The simplest constraints for finite element subspaces of H^1 and H(Curl) in 2D are illustrated in Figure 4.6.

The degrees of freedom can be split into two groups: *unconstrained (or true)* degrees of freedom and *constrained (or slave)* degrees of freedom. If z is a vector of all slave DOFs, then z can be expressed as $z = Wx$, where x is a vector of all true DOFs and W is a global interpolation matrix, handling indirect constraints and arbitrary differences in refinement levels of adjacent elements. Introducing the *conforming prolongation matrix*

$$P = \begin{pmatrix} I \\ W \end{pmatrix},$$

we observe that the coupled AMR linear system can be written as

$$P^t A P x_c = P^t b, \tag{4.11}$$

where A and b are the finite element stiffness matrix and load vector corresponding to discretization of the weak variational formulation on the "cut" space $\widehat{V}_h = \cup_i (V_h|_{K_i})$. After solving for the true degrees of freedom x_c we recover the complete set of degrees of freedom, including slaves, by calculating $x = Px_c$. Note that in MFEM this is handled automatically for the user via `FormLinearSystem()` and `RecoverFEMSolution()` An illustration of this process is provided in Figure 4.7.

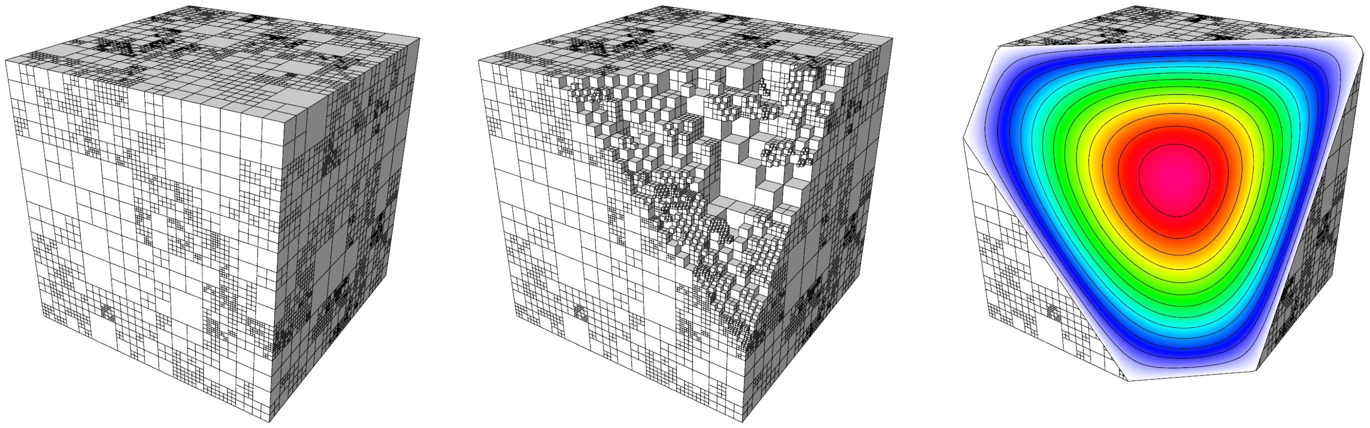


Figure 4.7: Illustration of the variational restriction approach to forming the global AMR problem. Randomly refined non-conforming mesh (left and center) where we assemble the matrix A and vector b independently on each element. The interpolated solution $x = Px_c$ (right) of the system (4.11) is globally conforming (continuous for an H^1 problem).

In MFEM, given an `NCMesh` object, the conforming prolongation matrix can be defined for each `FiniteElementSpace` class and accessed with the `GetConformingProlongation()` method. The algorithm for constructing this operator can be interpreted as a sequence of interpolations $P = P_k P_{k-1} \cdots P_1$, where for a k-irregular mesh the DOFs in $\widehat{V}_h$ are indexed as follows: 0 corresponds to true DOFs, 1 corresponds to the first generation of slaves that only depend on true DOFs, 2 corresponds to second generation of slaves that only depend on true DOFs and first generation of slaves, and so on. k corresponds to the last generation of slaves. We have

$$P_1 = \begin{pmatrix} I \\ W_{10} \end{pmatrix}, P_2 = \begin{pmatrix} I & 0 \\ 0 & I \\ W_{20} & W_{21} \end{pmatrix}, \ldots, P_k = \begin{pmatrix} I & 0 & \cdots & 0 \\ 0 & I & \cdots & 0 \\ & & \ddots & \\ 0 & 0 & \cdots & I \\ W_{k0} & W_{k1} & \cdots & W_{k(k-1)} \end{pmatrix}$$

are the local interpolation matrices defined only in terms of the edge-to-edge and face-to-face constraining relations. Note that while MFEM supports meshes of arbitrary irregularity ($k \geq 1$), the user can specify a limit on k when refining elements, if necessary (an example of a 1-irregular mesh is shown in Figure 4.11).

The basis for determining face-to-face relations between hexahedra is the function `FaceSplitType`, sketched below. Given a face (v_1, v_2, v_3, v_4), it tries to find mid-edge and mid-face vertices and determine if the face is split vertically, horizontally (relative to its reference domain), or not split.

```
Split FaceSplitType(v1, v2, v3, v4)
{
   v12 = FindVertex(v1 , v2);
   v23 = FindVertex(v2 , v3);
   v34 = FindVertex(v3 , v4);
   v41 = FindVertex(v4 , v1);

   midf1 = (v12 != NULL && v34 != NULL) ? FindVertex(v12, v34) : NULL;
   midf2 = (v23 != NULL && v41 != NULL) ? FindVertex(v23, v41) : NULL;

   if (midf1 == NULL && midf2 == NULL)
      return NotSplit;
   else
      return (midf1 != NULL) ? Vertical : Horizontal;
}
```

The function `FindVertex` uses a hash table to map end-point vertices to the vertex in the middle of their edge. This algorithm naturally supports anisotropic refinement, as illustrated in Figure 4.8.

The algorithm to build the P matrix in parallel is more complex, but conceptually similar to the serial algorithm. We still express slave DOF rows of P as linear combinations of other rows, however some of them may be located on other MPI tasks and may need to be communicated first.

Unlike the conforming `ParMesh` class, which is partitioned with METIS, the `ParNCMesh` is partitioned between MPI tasks by splitting a space-filling curve obtained by enumerating depth-first all leaf elements of all refinement trees [124]. The simplest traversal with a fixed order of children at each tree level leads to the well-known Morton ordering, or the Z-curve. We use instead the more efficient Hilbert curve that can be obtained just by changing the order of visiting subtrees at each level [125]. The use of space-filling curve partitioning ensures that balancing the mesh so that each MPI task has the same number of elements (± 1 if the total number of elements is not divisible by the number of tasks) is relatively straightforward.

These algorithms have been heavily optimized for both weak and strong parallel scalability as illustrated in Figure 4.9, where we report results from a 3D Poisson problem on the unit cube with exact solution having two shock-like features. We initialize the mesh with 32^3 hexahedra and repeat the following steps, measuring their wall-clock times (averaged over all MPI ranks): 1) Construct the finite element space for the current mesh (create the P matrix); 2) Assemble locally the stiffness matrix A and right hand side b; 3) Form the products $P^t AP$, $P^t b$; 4) Eliminate Dirichlet boundary conditions from the parallel system; 5) Project the exact solution u to u_h by nodal interpolation; 6) Integrate the exact error $e_i = ||u_h - u||_{E,K_i}$ on each element; 7) Refine elements

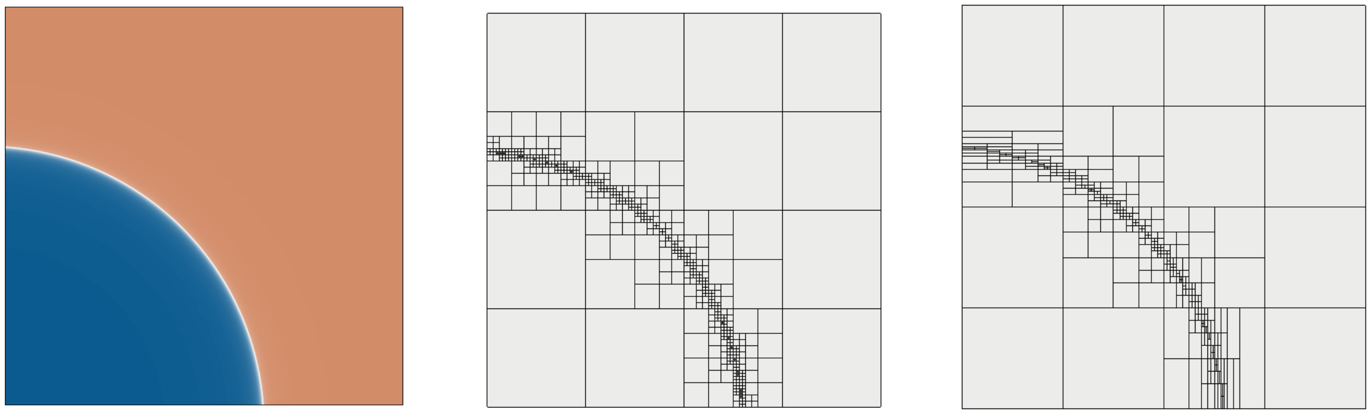


Figure 4.8: Left: 2D benchmark problem for a Poisson problem with a known exact solution. Center: Isotropic AMR mesh with 2197 DOFs. Right: Anisotropic AMR mesh with 1317 DOFs. Even though the wave front in the solution is not perfectly aligned with the mesh, many elements could still be refined in one direction only, which saved up to 48% DOFs in this problem for similar error.

with $e_j > 0.9\max\{e_i\}$; 8) Load balance so each process has the same number of elements (± 1). We run about 100 iterations of the AMR loop and select iterations that happen to have approximately 0.5, 1, 2, 4, 8, 16, 32 and 64 million elements in the mesh at the beginning. We then plot the times of the selected iterations as if they were 8 independent problems. We run from 64 to 393,216 (384K) cores on LLNL's Vulcan BG/Q machine. The solid lines in Figure 4.9 show strong scaling, i.e. we follow the same AMR iteration and its total time as the number of cores doubles. The dashed lines skip to a double-sized problem when doubling the number of cores showing weak scaling, and should ideally be horizontal.

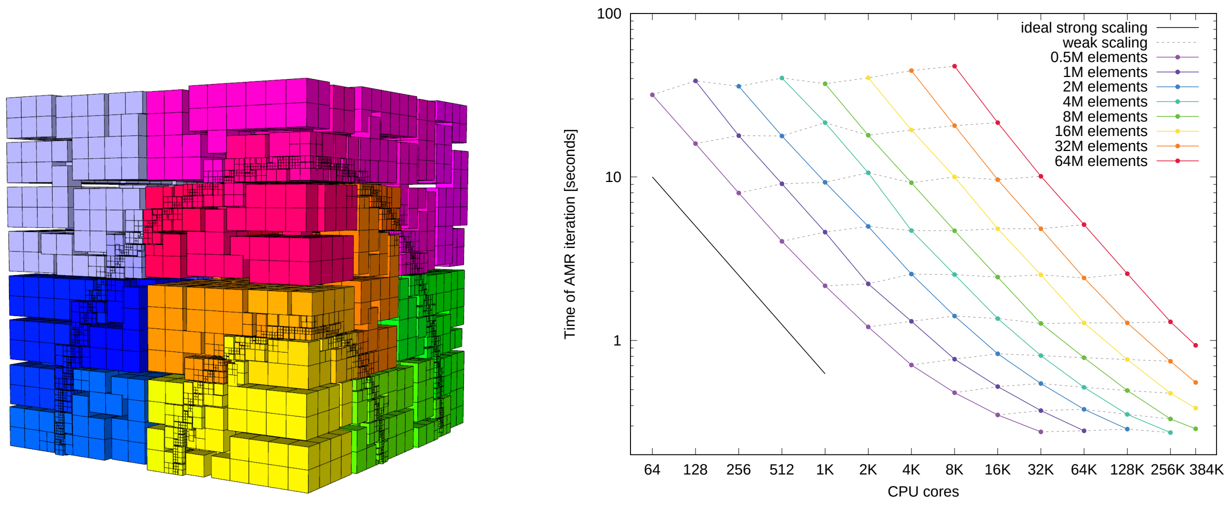


Figure 4.9: Left: One octant of the parallel test mesh partitioned by the Hilbert curve (2048 domains shown). Right: Overall parallel weak and strong scaling for selected iterations of the AMR loop. Note that these results test only the AMR infrastructure, no physics computations are being timed.

MFEM's variational restriction-based AMR approach can be remarkably unintrusive when it comes to integration in a real finite element application code. To illustrate this point we show two results from the *Laghos* miniapp which required minimal changes for static refinement support (see Figure 4.10) and about 550 new lines of code for full dynamic AMR, including derefinement (see Figure 4.11).

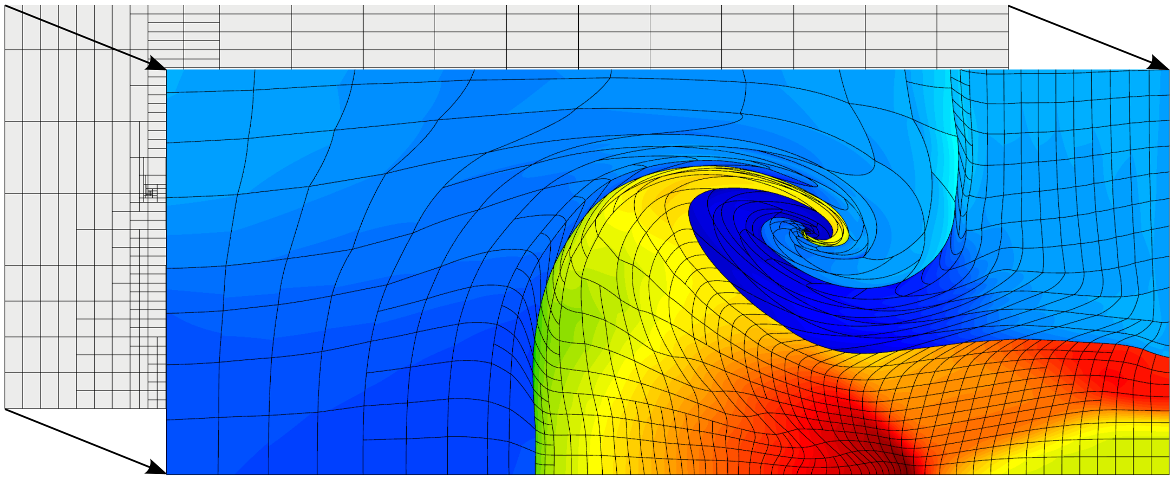


Figure 4.10: MFEM-based static refinement in a triple point shock-interaction problem. Initial mesh at $t = 0$ (background) refined anisotropically in order to obtain more regular element shapes at target time (foreground).

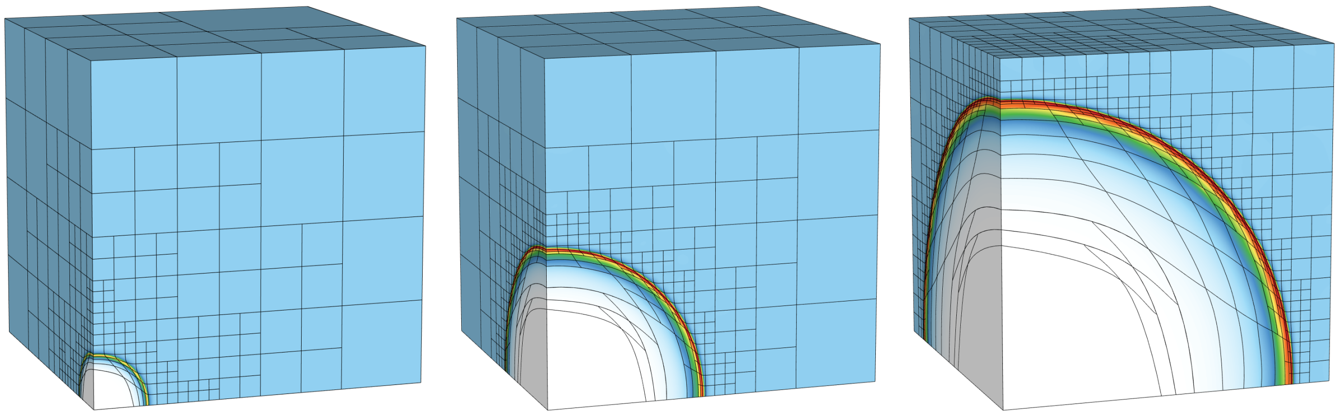


Figure 4.11: MFEM-based dynamic refinement/derefinement in the 3D Sedov blast problem. Mesh and density shown at $t = 0.0072$ (left), $t = 0.092$ (center) and $t = 0.48$ (right). Q3Q2 elements ($p = 3$ kinematic, $p = 2$ thermodynamic quantities).

4.3 High-Order Finite Difference Methods

4.3.1 Parcop: compact finite difference library (Pyranda)

High-order, spectral-like compact operators are generically given by the solution of the linear equation $\mathbf{Ax} = \mathbf{By}$, where $\mathbf{y}$ is the variable to be operated on and $\mathbf{x}$ is the result, typically a derivative or a filtered version of the variable in a given spatial direction. The matrices $\mathbf{A}$ and $\mathbf{B}$ are sparse band-diagonal matrices (see Lele 1992 [126] for an extensive analysis). For high-order operations, we use pentadiagonal matrices for $\mathbf{A}$ and band-diagonal matrices with a stencil width of 7 or 9 for $\mathbf{B}$, the solution whereof requiring a global matrix inversion of $\mathbf{A}$. In cases where $\mathbf{A}$ is the unit matrix, the operation $\mathbf{x} = \mathbf{By}$ is explicit and obviously requires no matrix inversion. For example, away from boundaries, a centered 10th-order 1st derivative f' in any direction is given by

$$. \sum_{p=0}^{2} a_p (f'_{i+p} + f'_{i-p}) = \sum_{e=1}^{3} b_e \frac{(f_{i+e} - f_{i-e})}{2e\Delta}. \tag{4.12}$$

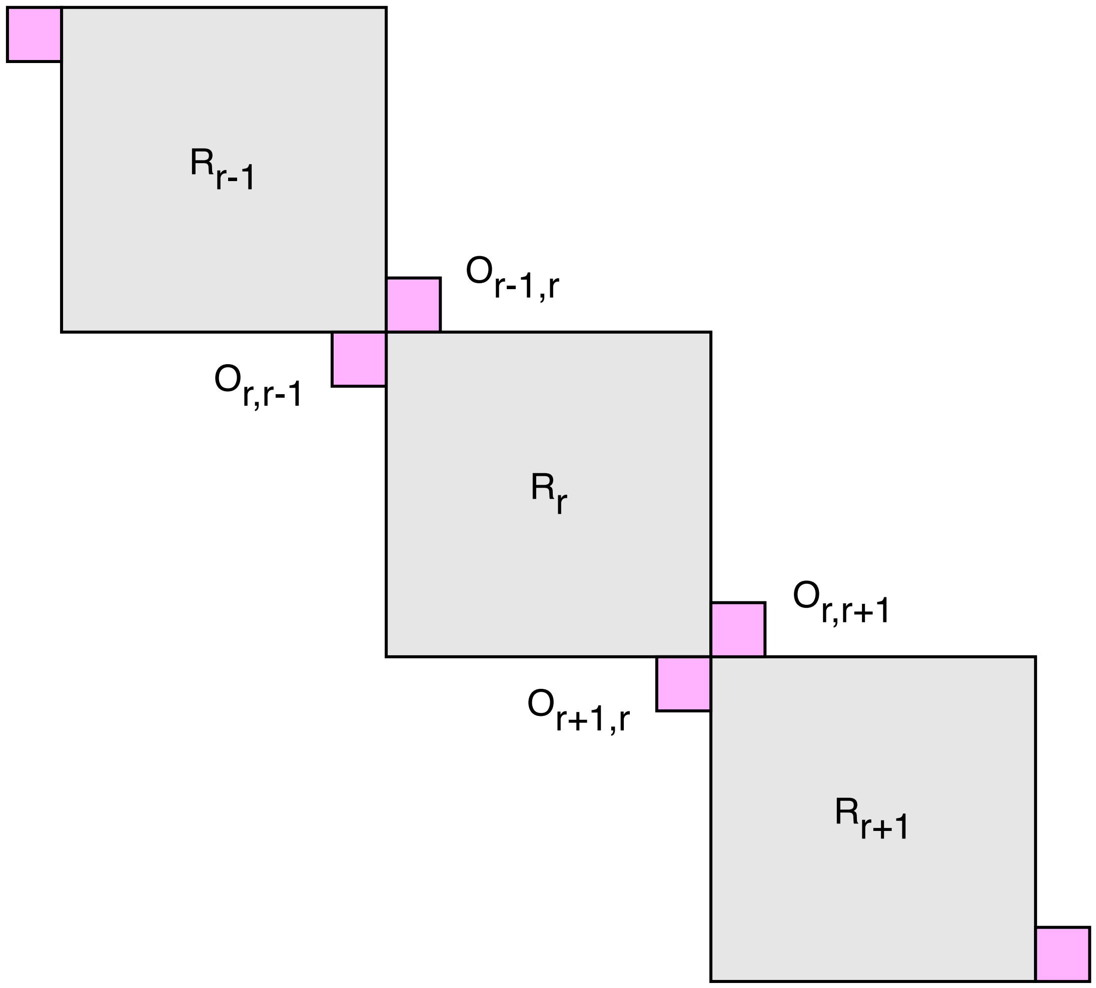


Figure 4.12: Decomposition of the left-hand side matrix **A** for implicit compact operations into the parts defined locally on each rank **R** (grey) and the parts overlapping neighboring ranks **O** (pink).

and a centered 8th-order filter $\tilde{f}$ is given by

$$. \sum_{p=0}^{2} a_p(\tilde{f}_{i+p} + \tilde{f}_{i-p}) = \sum_{e=1}^{4} b_e \frac{(f_{i+e} + f_{i-e})}{2e\Delta} \tag{4.13}$$

(e.g., see Cook et al. 2004 [127]).

When these operations are performed on multiple MPI ranks, the explicit right-hand side operation requires an exchange of 3 or 4 "halo" or "ghost" cells from neighboring ranks on the MPI communicator in the direction of the operation, which should nominally have good weak scaling. The implicit solution involving the pentadiagonal matrix on the left-hand side is obtained directly by inverting the pentadiagonal matrix with standard LU decomposition methods. On multiple MPI ranks, the pentadiagonal matrix **A** is decomposed into elements that are defined on each rank **R** and those that extend onto neighboring ranks **O** (see Fig. 4.12). The on-rank matrix **R** is inverted locally on each rank to give an incomplete solution $\mathbf{x}^{\star} = \mathbf{R}^{-1}\mathbf{B}\mathbf{y}$. Letting $\mathbf{x} = \mathbf{x}^{\star} + \delta\mathbf{x}$, the correction needed to obtain the global solution is given by $\mathbf{A}\delta\mathbf{x} = \mathbf{O}\mathbf{x}^{\star}$, where the global overlap vector $\mathbf{O}\mathbf{x}^{\star}$ is very sparse (with only 4 nonzero points from each rank). Currently $\mathbf{O}\mathbf{x}^{\star}$ is assembled using an `MPI_Allgather` operation, and each rank computes the global overlap solution that can be cast in terms of a block-tridiagonal matrix with 2x2 blocks. However, the `MPI_Allgather` operation does not scale in the weak sense, depending rather on the volume of overlap data. For CTS1 runs, the compute times are typically large compared with communication times, so this lack of communication scaling is not appreciable, but it comes very much to the forefront in large-scale simulations on GPU machines with the much shorter compute times (see Section 3.8.4). In the future, we will investigate more scalable (albeit less accurate) methods, like iterative or explicit solutions.

Peer-to-peer communication, also known as GPUDirect™, has been implemented in the code for compact operations on IBM-NVIDIA architectures (e.g., Sierra). While this speeds up on-node halo communication, the cross-node communication is still performed by MPI via the CPU. We have also found that `MPI_Allgather` operations are performed on the CPU for this architecture. As a result there is no added speed-up in communication for large-scale simulations. For future ATS architectures with network interconnects directly connected to the GPU, this approach could lead to significant reductions in data motion time.

Domain boundaries. For ranks bounded by a physical boundary, the stencils for the compact operations can incorporate built-in symmetry (e.g., for reflective boundary conditions), or the stencils can be tailored to do one-sided, typically non-centered and lower-order, operations. For periodic boundary conditions, the pentadiagonal left-hand side matrix **A** has additional corner bands to account for the solution wrapping around the domain, such that the matrix becomes essentially nonadiagonal and more expensive to invert; on multiple ranks, the local pentadiagonal solutions are the same, but the global overlap solution with the periodic wraparound becomes essentially a block-pentadiagonal system and is again somewhat more expensive to invert than non-periodic solutions.

Pyranda (parcop library) [128]. The compact operators used in `MARBL -eul` are available to the general public in the *Pyranda* open-source code in the `parcop` library (https://github.com/LLNL/pyranda).

4.3.2 AMR strategy

The implementation of adaptive mesh refinement (AMR) into the high-order Eulerian capabilities has been done using the SAMRAI [129] library. SAMRAI is a C++ library developed at LLNL that enables the integration of AMR capabilities into massively-parallel physics applications that use structured meshes. SAMRAI provides the tools to construct and manage a multi-level patch-based mesh hierarchy with flexible software components that allow the application to maintain control of its simulation data structures and its numerical algorithms.

SAMRAI's patch-based AMR hierarchy (Figure 4.13) is comprised of a set of spatially-nested levels with different mesh resolutions from the coarsest (base) level to the finest, with the location of the finer levels chosen in places where higher resolution is desired due to the features of the simulation. Each level is decomposed in parallel into logically-rectangular regions called "patches" which serve as the mesh space on which data is allocated and where local numerical kernels are executed. Local operations on the patches occur in a distributed fashion until the point where parallel operations are needed to exchange data between patches. SAMRAI provides the communication infrastructure for data to be exchanged between adjacent patches on the same level as well as between overlapping patches on different levels. As the simulation advances in time, the regions requiring the highest resolution will change, due to the motion of such things as shock fronts or material interfaces and SAMRAI provides the tools to adaptively change the AMR hierarchy according to criteria that can be specified at run-time.

The approach chosen to implement AMR involves two modes: one for the base level and one for all finer levels. The base level is organized and operates exactly as it would if it were not using AMR. The uniform parallel decomposition, which existed in the code before any AMR additions, stays in place, and all of the communication patterns needed to execute the global solves for the directional partial derivatives described in section 2.3 are unchanged. The mode for finer levels is different due to the frequent changes to the mesh location and parallel distribution of the AMR patches.

On fine levels the logically rectangular mesh is expanded by a specified number of ghost layers in each direction. The ghost layers are initialized with the best available data from other patches, which would be the data from the overlapping zones of adjacent patches of the same resolution where they exist, or interpolated data from the next coarser level in locations where there is no adjacent patch. During the hydrodynamic advance, the fine patches operate independently until they reach a synchronization point after the RK4 scheme is completed. In the synchronization the ghost layers are updated with new best available data

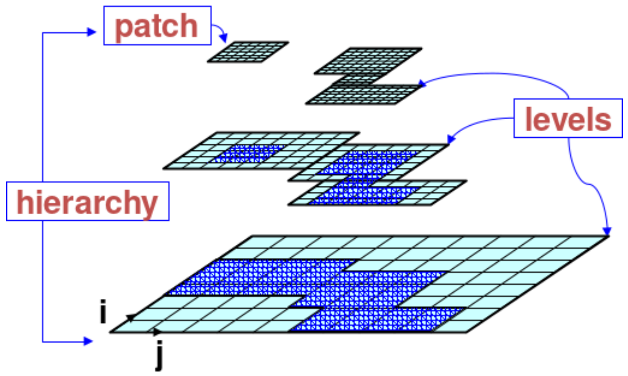


Figure 4.13: A structured AMR hierarchy contains nested levels wich in turn contain distributed patches.

communicated from adjacent or coarser patches.

4.3.3 Interpolation and coarsening operation schemes

The AMR implementation required interpolation and coarsening algorithms written specifically to be compatible with the high-order scheme. The interpolation algorithm is a high-order, compact implicit operator which refines the content on a coarse grid to a higher resolution. In the interior of a mesh patch, the interpolator is a 10th-order Pade scheme using information from near-neighbor points. Specifically, the operator takes nine coarse points as a unit of input, and outputs five points at a fine spacing around the center of the input. The coefficients of the compact interpolator are pre-calculated by a high-order Taylor-series expansion of the approximation expression about the center. Similar to other finite-difference implicit operators for derivative evaluation, the interpolation operator is applied through solving a banded penta-diagonal matrix system employing LU decomposition. For 2D or 3D interpolation, the algorithm also averages over different sequences of 1d interpolations to minimize numerical anisotropy. The spectral response of the interpolator is a uniform function smoothly tapering off near the Nyquist limit of the fine grid. As a result, the content from a coarse grid is preserved and enhanced by simulation in the subsequent time step. For mass conservation, field variables weighted by coarse density are interpolated and normalized by refined density.

Conversely, the coarsening algorithm is also a high-order, compact implicit operator which downsamples a high-resolution input and overwrites the content on a coarse grid. The coarsening operator takes a unit of nine input points and outputs five points around the center of the input. The operator coefficients are pre-calculated by a high-order Taylor-series expansion of the approximation expression and constraints at the Nyquist limit. Similar to the implicit interpolator, the coarsening operator is applied through solving a banded penta-diagonal matrix system employing LU decomposition. The spectral response of the coarsening operator preserves low wavenumbers and tapers off smoothly near the Nyquist limit of the coarse grid, following a decaying Gaussian function. In essence, the coarsening operator acts as a low-pass and anti-aliasing filter to the input data prior to sample decimation. As a result, the enriched simulation at a high resolution transfers only the long-wavelength content, which can be adequately resolved by the coarse grid. For mass conservation, field variables weighted by refined density are decimated and normalized by decimated density.

The accuracy of interpolation and coarsening operators have been tested on several functions. One example is the advection of an energy pulse through a uniform background. Measurements of the L2-norm errors show the convergence rates of AMR operations involving both interpolation and coarsening in Figure 4.14. While the pulse is stationary, the repeated interpolation followed by coarsening is able to restore the original pulse function with negligible errors. When the pulse advects, numerical errors increase primarily due to the truncation effects along the coarse-fine mesh interfaces.

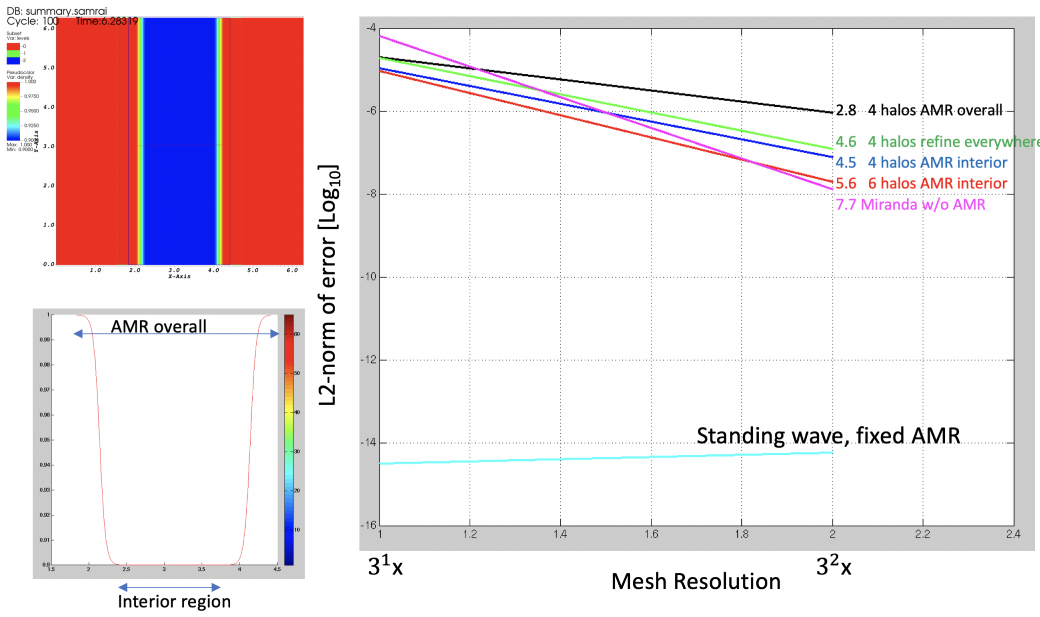


Figure 4.14: Convergence rates of the L2-norm of the errors between simulations and analytical solutions.

4.3.4 AMR applications

The objective of AMR-assisted simulation is to keep track of content variation and refine important features without excessive use of computation and memory resources. Throughout time-stepping, the simulated fields are examined at a regular time interval to evaluate the need of AMR coverage based on user specified conditions and thresholds, such as shock, pressure, temperature, material, and interface. Different AMR coverage criteria can be logically combined to follow active features and update the evolving AMR mesh hierarchy accordingly. Additional AMR-specific parameters can help either extend AMR mesh coverage or adjust the minimum size of the mesh. Since numerical truncation errors primarily arise from coarse-fine mesh interfaces, the AMR coverage criteria shall be optimized to keep active features well covered by the mesh hierarchy.

The implemented AMR scheme has been applied to a variety of 2D and 3D applications, including smooth benchmarks such as Taylor-Green evolution, blast propagation with sharp pressure gradients (Taylor blast and Sedov), multi-material hypersonic, and multi-physics applications involving both hydrodynamics and radiation. To work with different physics processes, the AMR capabilities are also extended to handle free, periodic, symmetric, wall, and specified boundary conditions. Figure 4.15 shows a multi-level AMR simulation of Taylor-Green with a periodic boundary condition, following high vorticity regions.

Sharp gradients in a simulation shall be well sampled within the bandwidth of numerical grids; otherwise ringing noises due to the Gibbs phenomenon may grow near the coarse-fine mesh interfaces. Figure 4.16 shows a Taylor blast simulation in which the resolution is enhanced by two AMR levels. Compared to the full-scale simulations, the AMR refined results match closely with the full-scale and are free of ringing noises. As a result of down-sampling from the top resolution, the fidelity of the coarsened fields at lower levels is also enhanced in terms of both amplitude and phase.

Figure 4.17 shows a multi-material, multi-physics simulation of a radiative blast over a surface. The adaptive mesh follows the expanding shock and covers both thermal energy variation inside the shock and a pressure gradient in the ground, through tracking both material and pressure thresholds. The refined mesh provides a high-resolution solution to the interaction between the impinging radiative shock and the surface.

By following sparse active features in a simulation, adaptive mesh refinement can boost computational efficiency significantly by avoiding computing at the highest resolution over the entire simulation domain. Performance profiling shows relatively a low-to-moderate overhead incurred by the AMR to overall simulation since mesh refinement, involving both interpolation and

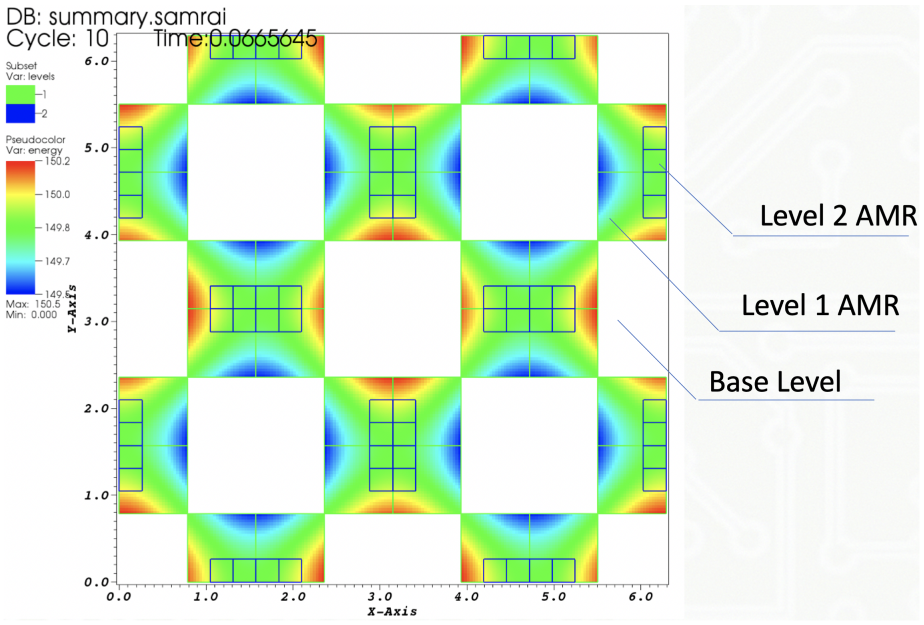


Figure 4.15: Taylor-Green simulation with two levels of AMR tracking high vorticity.

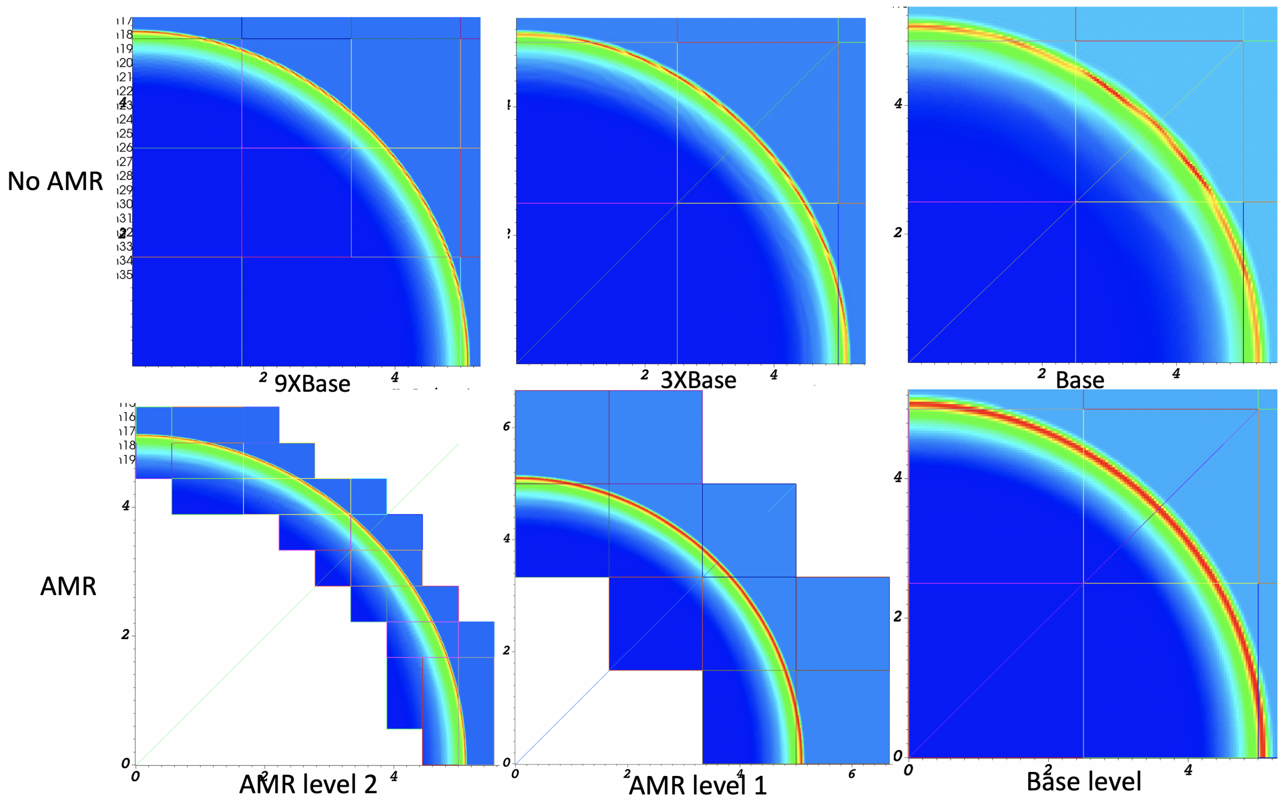


Figure 4.16: Comparison of full-scale and AMR simulations at different levels of resolutions. Note the AMR results are free of truncation-related ringing noises for such a sharp pulse.

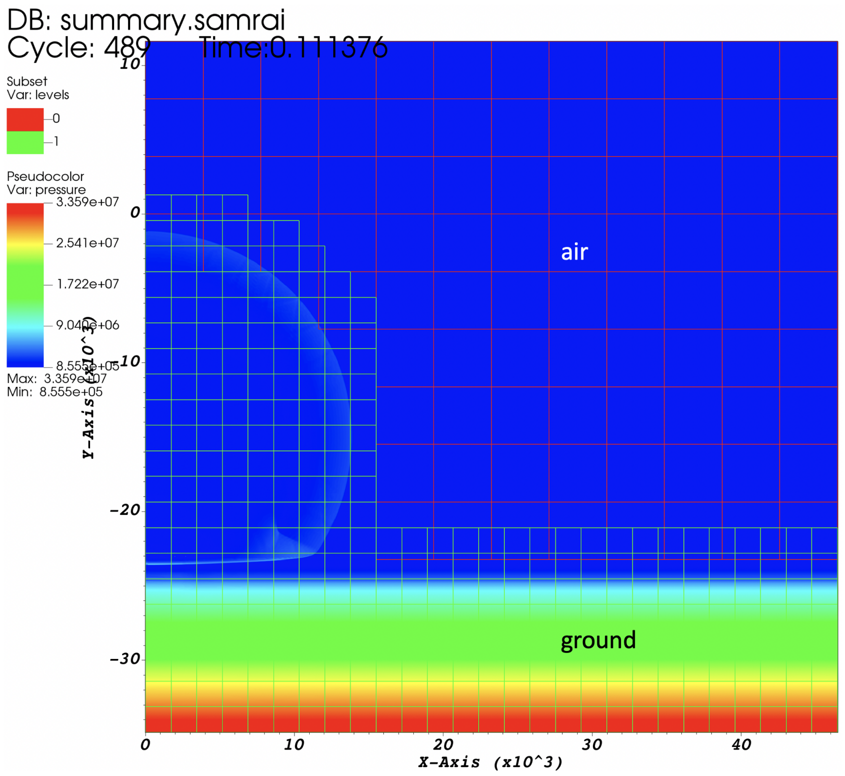


Figure 4.17: Simulation of a radiative blast over a surface with adaptive mesh refinement.

coarsening, occurs only intermittently before state update goes beyond a single mesh cell. Overall wall-clock time and memory usage is usually a small fraction of the full-scale cost, which typically grows exponentially with increasing spatial resolutions.

To troubleshoot AMR related issues, users can check numerical artifacts from the last visualization output at the location where the updated time step drops significantly. Ringing noises along the coarse-fine mesh interfaces tend to indicate errors from truncating energetic features. Users usually begin with checking whether the initial condition including the source specification are well resolved by the mesh resolutions and whether different materials and properties are in agreement with the geometry specifications. In addition, users can check if the AMR mesh hierarchy properly covers the extent of active features, such as pressure and materials. If the coverage is incomplete or too tight, AMR meshes shall be extended so that coarse-fine mesh interfaces are away from energetic features. To diagnose efficiently, it is recommended to begin with configuring and testing the base level followed by the first AMR level to verify the setup parameters.

Chapter 5

MARBL Applications

As previously stated, success of MAPP will ultimately be determined by the degree of adoption of its simulation tools by the LLNL user community and beyond. We believe user engagement as soon as possible is essential to provide the necessary feedback to generate a useful product (i.e. customer focus [2]). As such, we have been periodically releasing public "alpha" versions of the code to an ever increasing set of early adopters since the beginning of FY17 and incorporating their feedback into future development. In this section, we review a collection of user applications of MARBL for modeling high-energy density physics experiments.

5.1 ICF Capsules

One of the first multiphysics applications for MARBL was ICF capsule modeling. In practice, ICF capsule simulations require a multi-group treatment of radiation diffusion (or transport) for physics accuracy. However, in 2017 MARBL had a only basic 2T single group radiation diffusion capability. LLNL physicist Robert Tipton proposed a simple ICF capsule design which is intended to be simulated using radiation-hydrodynamics codes with single-group radiation diffusion methods [130]. The goal was to create an ICF Test problem that was close to the NIC Rev. 5 ICF point design, but simple enough to be run by codes which didn't yet have the full physics capabilities needed for typical ICF design simulations. The problem consists of four materials: DT gas, DT ice, a plastic (CH) ablator and a surrounding Helium filled region representing the space inside of a typical ICF hohlraum. The problem is driven by a time dependent radiation temperature profile representing a 4-shock laser pulse that was specifically tuned for 1D capsules with single-group radiation diffusion as shown in Figure 5.1.

This problem can be run in 1,2 and 3 dimensions and is therefore highly valuable for testing and comparisons with other codes. We have been using this problem as a testbed for many aspects of the MARBL code, including testing hydrodynamics, tabular equations of sate and opacities, matter-radiation coupling and TN burn as described in Section 2.4.6. It has also proven effective as a stress test for ALE remesh algorithms in 2D and 3D. As an example, in Figure 5.2 we show 2D axisymmetric high-order ALE results of the Tipton capsule benchmark where we have prescribed a single 10% P8 Legendre mode density perturbation to the DT ice material at the initial time, thereby introducing a 2D symmetry breaking mechanism. This configuration makes the use of ALE remesh/remap essential to capture the Rayleigh-Taylor (RT) and Richtmyer-Meshkov (RM) hydrodynamic instabilities that will form as a result of this initial perturbation. As the DT gas is compressed, fusion reactions begin to occur, producing TN neutrons. MARBL stores time histories for integrated quantities like TN neutron production and material volumes directly in the Sidre datastore object which are accessible via the user interface for plotting / checkpointing. In Figure 5.3 we show the computed material volumes as a function of time as well as the computed TN neutron yield.

In 2017, LLNL physicist Steve Langer, in collaboration with summer intern Trevor Parker, started using an early version of MARBL for simplified ICF capsule modeling. One of the goals of their studies was to understand if the high-order hydrodynamic capabilities of MARBL could provide a benefit for simulating ICF capsules with a fill-tube feature which can pose a challenge for

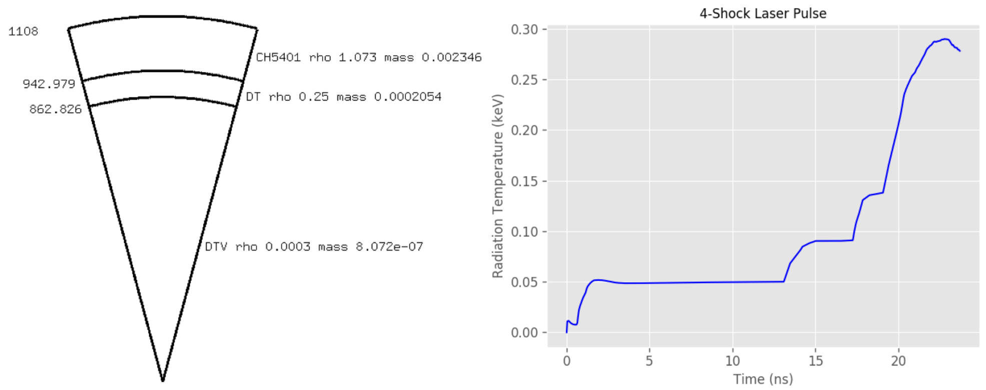


Figure 5.1: Pie-diagram for the Tipton capsule; a single-group radiation diffusion ICF benchmark consisting of DT gas, DT ice and a CH ablator material (*left*). The problem is driven with a radiation temperature vs. time profile applied to the outer boundary of the problem to model a 4-shock laser pulse (*right*).

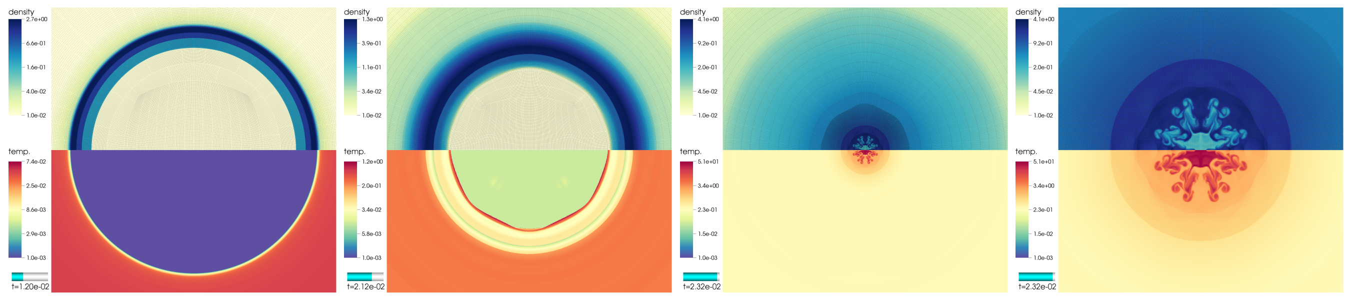


Figure 5.2: Snapshots of MARBL high-order ALE rad-hydro simulation of Tipton ICF capsule benchmark with a single 10% P8 Legendre mode density perturbation applied to the DT ice material at initial time. Left to right: density (*top-half*) and temperature (*bottom-half*)]) at t = 12ns, 21.2ns and 23.2ns and a zoomed in view at 23.2ns.

existing ALE codes due to the hydrodynamic instabilities such a feature can introduce. Performing a realistic fill tube simulation requires full radiation transport capability, since a diffusion approximation is not sufficient to model the flow of radiation in the presence of the fill tube. Nevertheless, a simplified model was developed to test the ALE hydrodynamics of a radiation diffusion driven ICF capsule implosion with an axial feature to stress test the high-order ALE hydrodynamics in MARBL as shown in Figure 5.4. In this example, we utilize the TMOP non-linear mesh optimization technology which offers a practical advantage for handling complex hydrodynamic flows of this type on a moving, high-order mesh.

5.2 Radiation Driven Kelvin-Helmholtz Instability

This work started in the summer of 2017 and was performed originally by Steve Langer (LLNL) and summer intern Trevor Parker [131]. The goal was to apply the recently developed high-order 2T ALE rad-hydro capabilities of MARBL to the modeling of a laser driven Kevin Helmholtz instability experiment of [132]. The experimental setup as well as the initial 3D MARBL model are shown in Figure 5.5. The experimental target package is a laser driven shock tube with a layer of foam next to a layer of plastic with a sinusoidal ripple machined on the interface between the two materials. The purpose of this prescribed interface perturbation is to initiate a shock driven Kelvin-Helmholtz “roll-up“ which can then be radiographed and compared with simulations. The laser heats an ablator that drives a shock into the foam. The shock causes shear flow at the interface between the foam and the plastic. The growth of the perturbation is tracked using backlit x-ray images.

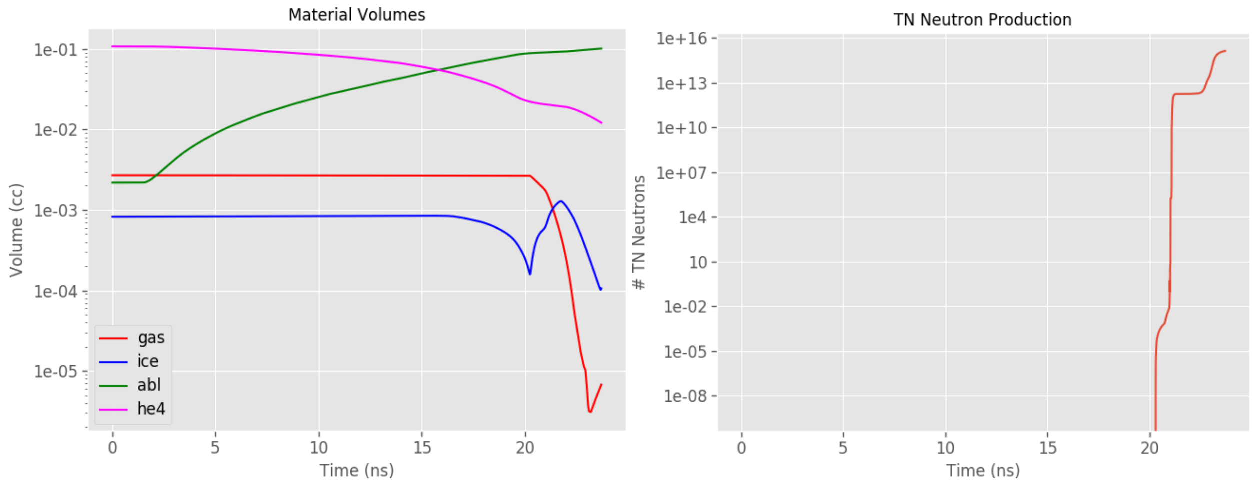


Figure 5.3: Material volumes vs. time (*left*) and total TN neutrons vs. time (*right*) computed by MARBL for the Tipton capsule benchmark in 2D axisymmetric geometry.

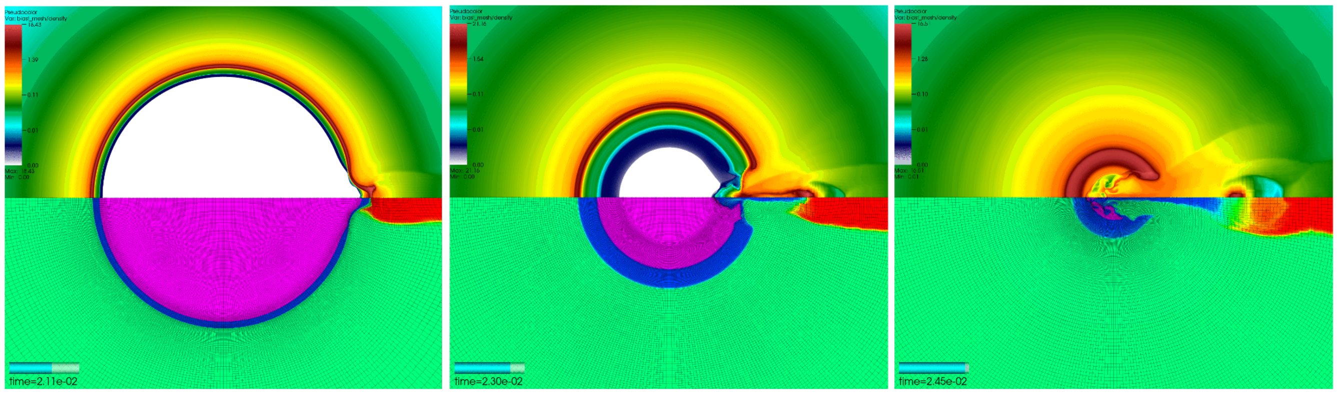


Figure 5.4: Snapshots of MARBL high-order ALE rad-hydro simulation of Tipton ICF capsule benchmark with a simplified axial fill tube perturbation, which later forms a jet.

The initial MARBL model of the K-H simulation used a radiation energy source with the strength adjusted to match the observed shock speed. Eventually, when a laser ray trace package is added to MARBL, it will be possible to directly simulate the pressure generated by the laser and the need for adjustments to match the experimental shock velocity will disappear. Figure 5.6 shows the 2017 3D MARBL simulation results (*top-row*) compared to experimental x-ray images (*bottom row*) at 25 ns, 45 ns, and 75 ns (*left to right*) where there is well-developed K-H rollup throughout the sample at the latest time. In the simulation images, the grey scale is inverted relative to the experiment. Qualitatively speaking, the MARBL simulated shock positions match the experiment and the growth of the perturbations is similar. One of the features that we are interested in is the appearance of ”bubbles” (lighter gray regions) above the vortices late in the experiment. Simulations with other rad-hydro codes suggest that these are due to excess pressure in the vortices pushing sideways on the beryllium that surrounds the target. These are intrinsically 3D features.

Since the initial 2017 work, we have continued to evolve the MARBL model of this experiment, including extensions of the 3D model to account for the shock tube side wall expansion into vacuum (the original model treated these as symmetric “wall” boundary conditions which would nonphysically tamp the 3D expansion), and use of new code capabilities like TMOP non-linear mesh optimization, utilization of multirate IMEX time-stepping and matrix free-partial assembly methods to improve runtime and

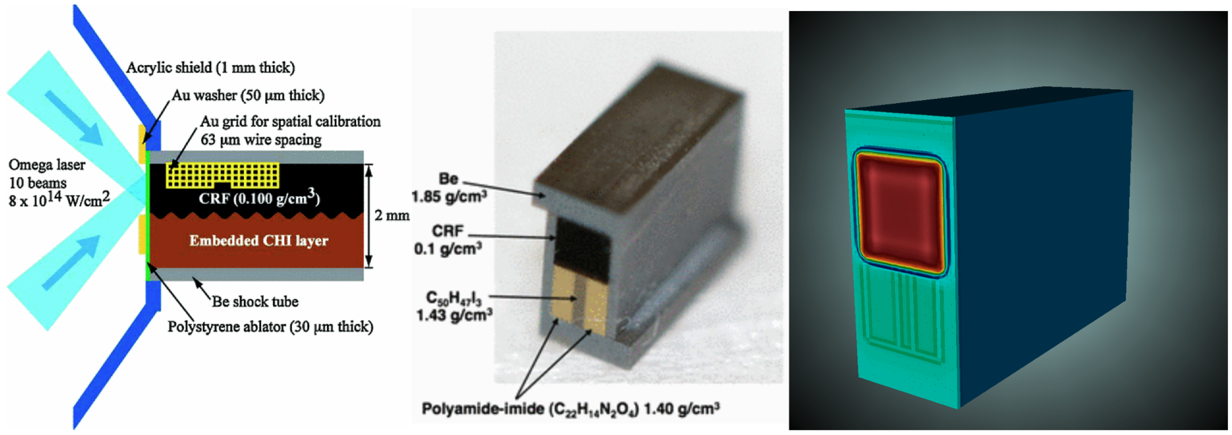


Figure 5.5: Schematic depiction of the K-H target (*right*), consisting of foam next to plastic with a ripple machined on the interface between the two materials; a 3D view of the experimental target package (*middle*) and the initial 2017 MARBL 3D model of Langer and Parker (*right*).

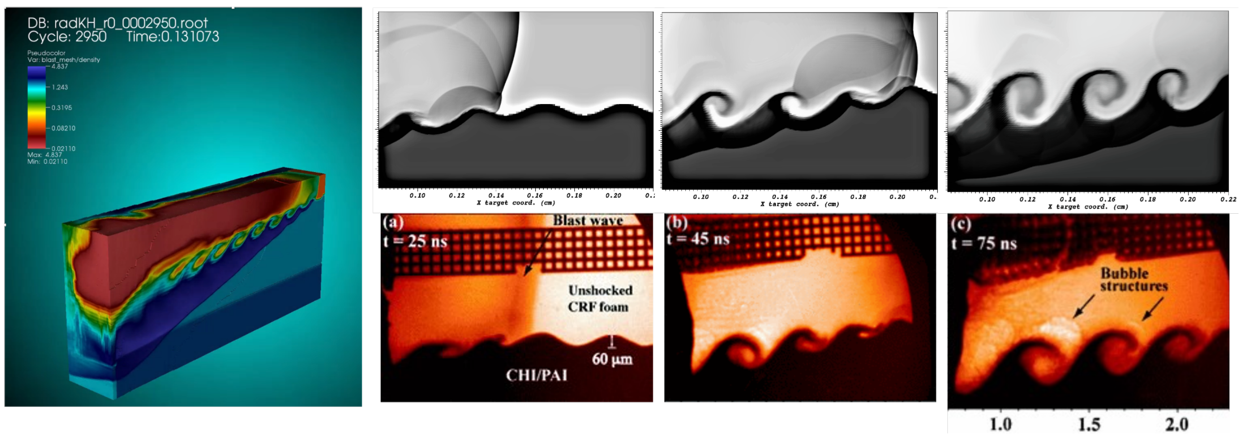


Figure 5.6: MARBL 3D simulation results (*top-row*) compared to experimental x-ray images (*bottom-row*) at 25 ns, 45 ns, and 75 ns (*left to right*) from the initial 2017 model.

use of the Ascent high-order mesh/field in-situ visualization capability for generating on-the-fly volume renderings, 3D images and simulated radiographs. In Figure 5.7 we MARBL results for a simplified 2D version of this model where we compare our standard implicit-explicit (IMEX) time stepping approach to the new multirate IMEX approach which enables super-cycling of the expensive implicit radiation diffusion solve relative the inexpensive explicit hydrodynamics. This enables a substantial run time speedup as shown in Figure 5.7 with no impact on simulation results. Results for the updated 2020 3D model are shown in Figure 5.8 along with in-situ volume rendering, pseudo-color plots and simulated radiographs produced by the high-order visualization library Ascent shown in Figure 5.9 which has been fully integrated into MARBL.

5.3 Surface Perturbations in Shock-Driven Metals

In 2019, MARBL was used by LLNL physicists Fady Najjar and Leo Kirsch to model a set of shocked liquid metal experiments for studying ejecta formation [133]. Specifically, MARBL models of experiments studying the dynamic evolution of microjets created

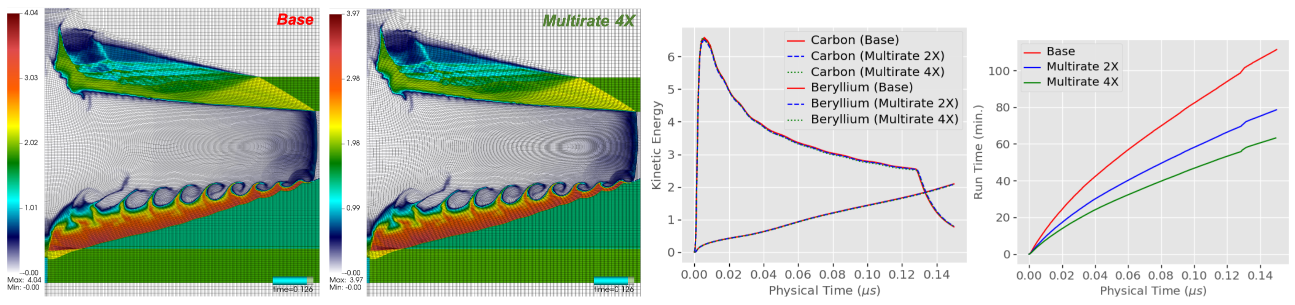


Figure 5.7: 2D MARBL results for radiating Kelvin-Helmholtz experiment simulation using new multirate IMEX for radiation-hydrodynamics which results in a runtime speedup with little to no impact to simulation results

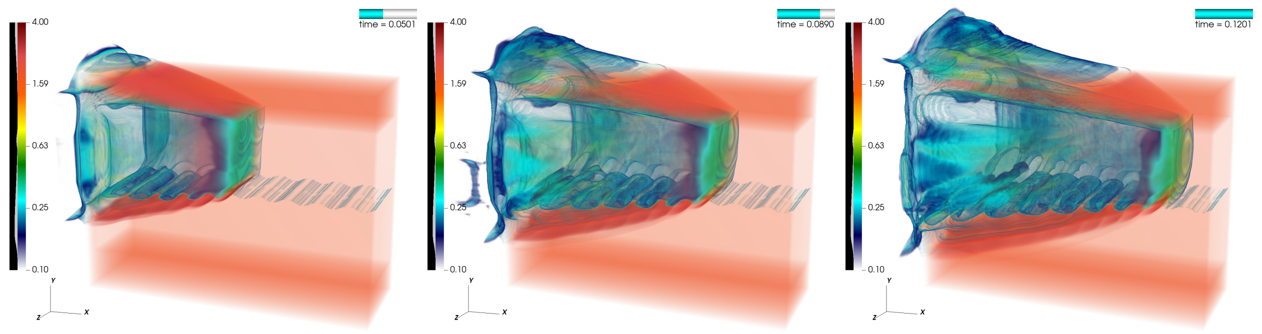


Figure 5.8: Snapshots of density for the updated 2020 MARBL 3D model of the radiating Kelvin-Helmholtz experiment. This improved model includes side wall expansion into vacuum, use of TMOP non-linear mesh optimization, multirate IMEX time-stepping and matrix-free partial assembly methods for improved run time and in-situ plotting, volume rendering and simulated radiography using the Ascent library. The model consists of 172,800,000 quadrature points and ran on 2304 MPI ranks of a CTS1 system for 70 hours.

by two distinct material types of Tin and Copper, as shown in the schematic of Figure 5.10. Ejecta represents a cloud of particles being emitted from a free material surface when impacted by a shock wave. Such ejecta particulates play a key role in a wide variety of natural and engineering applications, including supernovae explosions, asteroid strikes, inertial confinement fusion, and hazards on spacecrafts and satellites due to debris impact. Detailed computations were performed using MARBL's high-order arbitrary Lagrangian-Eulerian (ALE) hydrodynamics package along with a the non-conforming mesh refinement capability to understand the generation and evolution of ejecta formed from shock driven flow over imposed surface perturbations. These simulations are based on a recent experimental campaign for studying laser-driven ejecta where micron-sized divots have been fielded. The novel high-order mesh optimization algorithms of MARBL proved valuable for these simulations, enabling robust calculations of high-velocity jets on a moving mesh without the need for manual user intervention as shown in Figure 5.11.

5.4 Micro-structure Modeling of High Density Carbon Ablators

In the summer of 2020, LLNL physicist Luc Peterson and summer intern Jeremy Binagia used the v0.7 alpha release of MARBL to model experiments performed at the Omega laser facility to study the details of shock formation in high density Carbon (HDC) microstructure. These experimental results, described in [134], suggest that HDC microstructure may induce fluid instabilities. This is important to understand as HDC is a candidate material for ICF ablators for NIF. The HDC material heterogeneity can seed instabilities, caused by shock speed variation as it passes through grains of differing orientations. This can be measured

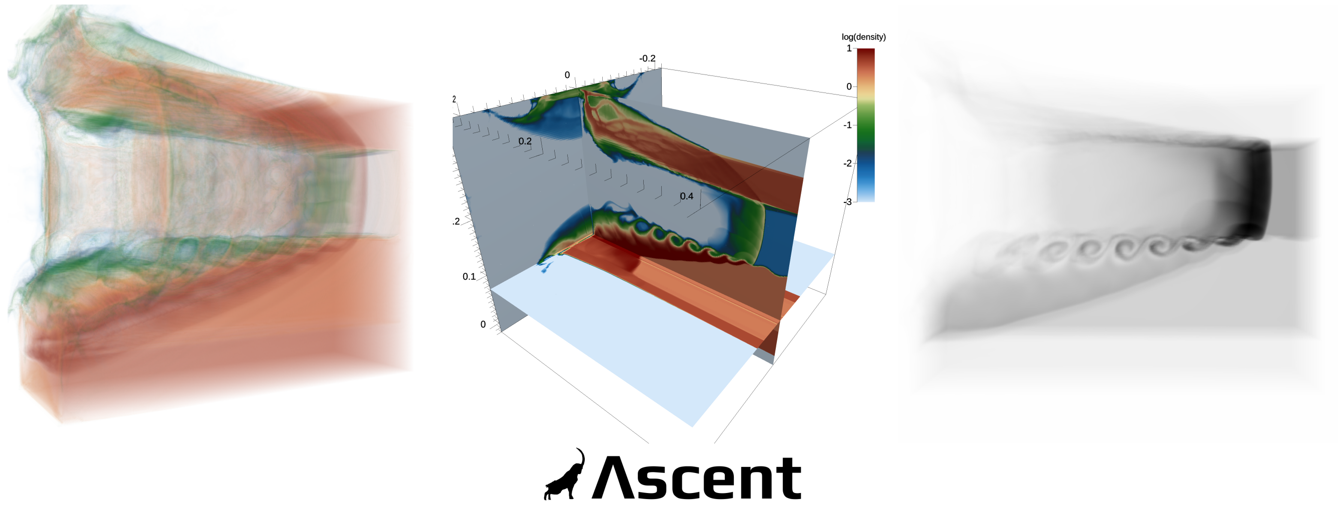


Figure 5.9: In-situ generated volume rendering (*left*,) pseudo-color plot (*middle*) and simulated radiograph (*right*) produced by the high-order visualization library Ascent which has been fully integrated into MARBL

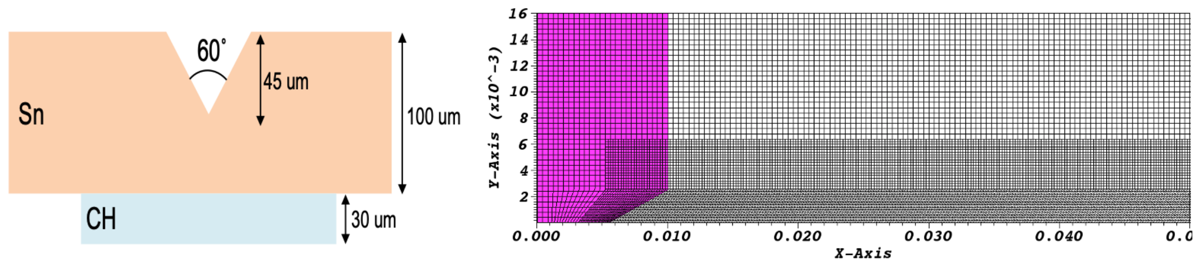


Figure 5.10: Schematic of target configuration (*left*) and 2D MARBL computational domain with high-order, non-conforming mesh showing the metallic foil (magenta) and the planar divot (*right*).

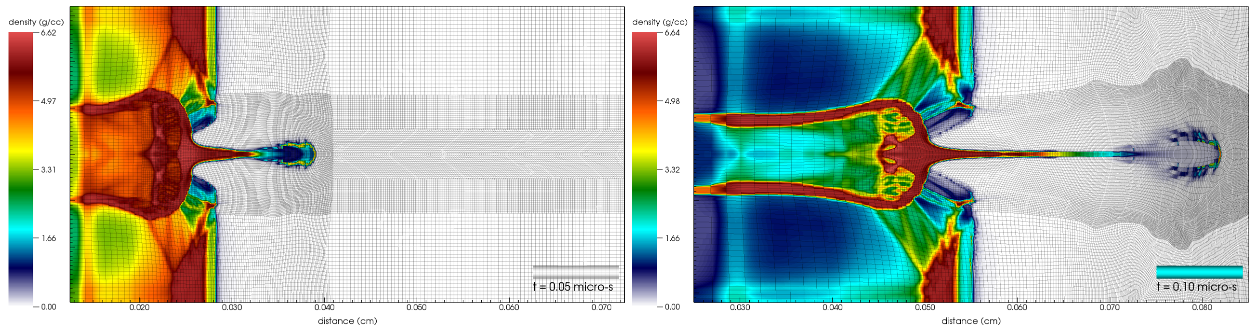


Figure 5.11: Snapshots at two different times of MARBL high-order ALE calculation of shocked liquid metal experiment showing material density and high-order ALE mesh

experimentally using a 2D VISAR diagnostic to measure the velocity non-uniformity at the shock front after it passes through the ablator. The goal of this summer project was to simulate, for the first time, a realistic 3D HDC microstructure via a Voronoi tessellation. Previous computational studies had been performed in 2D only.

The experimental schematic and a 2D view of the 3D MARBL model is shown in Figure 5.12. This example took advantage of MARBL capabilities in computational geometry and problem setup provided by the Axom library, which allowed us to quickly define a workflow for shaping 3D Voronoi microstructures into a 3D MARBL simulation as shown in Figure 5.13. This allowed us to setup and perform 3D high-order ALE rad-hydro simulations to confirm that the presence of the microstructure has a clear impact on the dynamics of the shock as shown in Figure 5.14. In this simulation, the shock is driven by a radiation temperature ($T_{rad} = 135eV$) set at the left boundary with a pulse duration of 3 ns. The computational mesh size is chosen to capture multiple HDC grain boundaries; HDC has a polycrystalline microstructure with grains ranging from 1 to 20 microns in size.

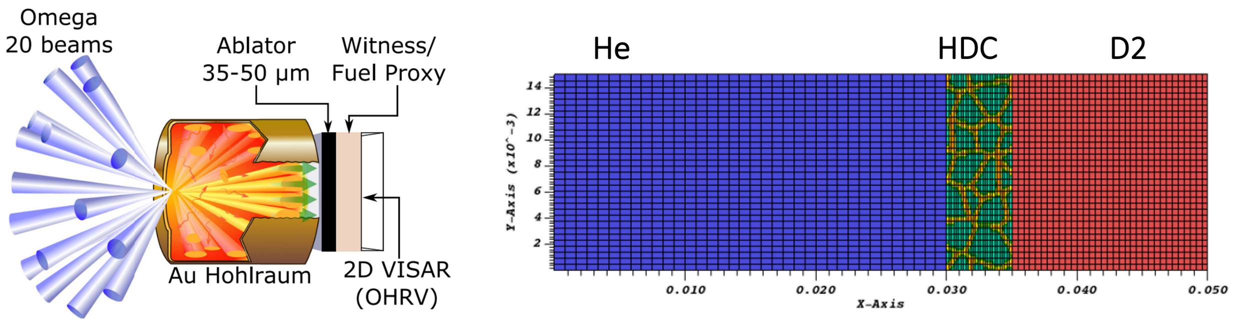


Figure 5.12: HDC microstruture experimental schematic (*left*) and a 2D view of the 3D MARBL model (*right*).

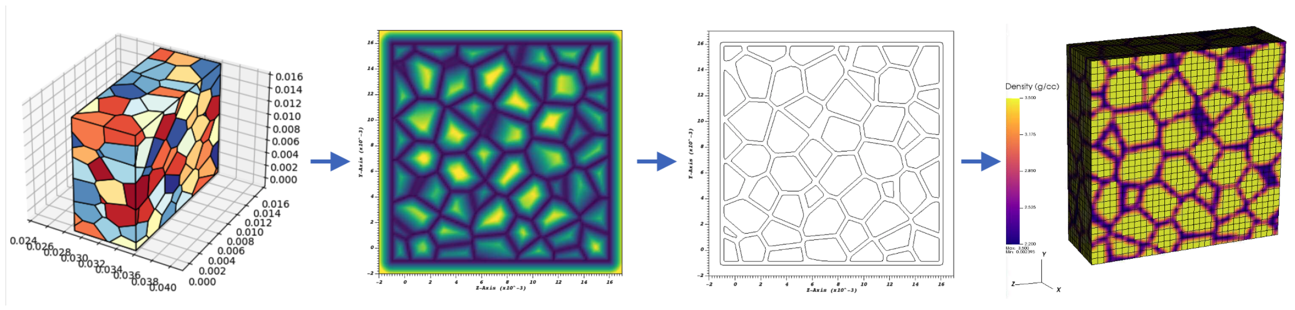


Figure 5.13: Workflow for shaping in 3D Voronoi structure into a MARBL simulation (*left-to-right*): Creation of polygon mesh using MicroStructPy open source code, generation of distance field using Axom, extract isosurface to define grain boundaries and finally, generate initial density for MARBL using Axom in/out spatial query at all quadrature points in the 3D mesh.

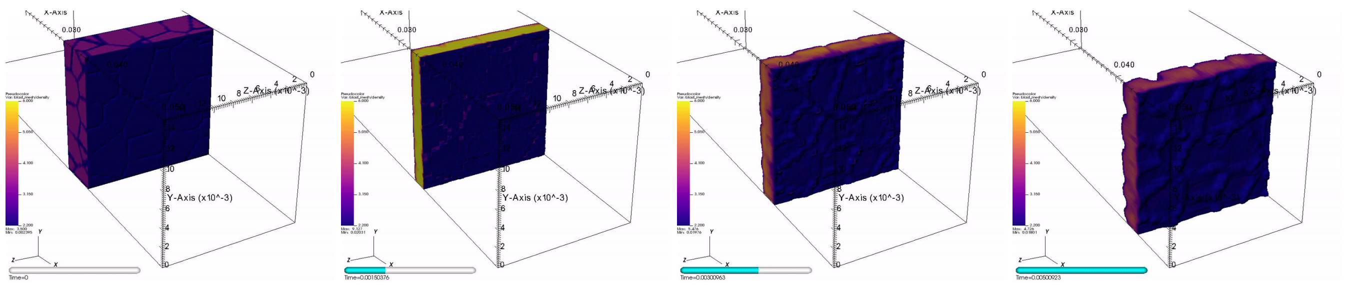


Figure 5.14: Snapshots of 3D MARBL simlation of HDC region undergoing shock compression from radiation ablation.

5.5 High Velocity Impact of Metal Lattice Structures

The recently developed GPU capability for the high-order ALE hydrodynamics in MARBL was used to model the high-velocity impact of a geometrically complex lattice structure made of Steel. The ultimate goal of this work is to perform many simulations to find optimal lattice structures for mitigating damage from high-velocity impacts. In the example shown in Figure 5.15, we model a lattice structure known as an ”octet-truss” which was readily “shaped” into a mesh by using the functional spatial querying capability and a recently developed python interface. In this case, the user defines a complex geometric inside/outside query in python which is then called at run time from MARBL to define the high-order volume fraction fields for the Steel and the (background) Air materials of the mesh.

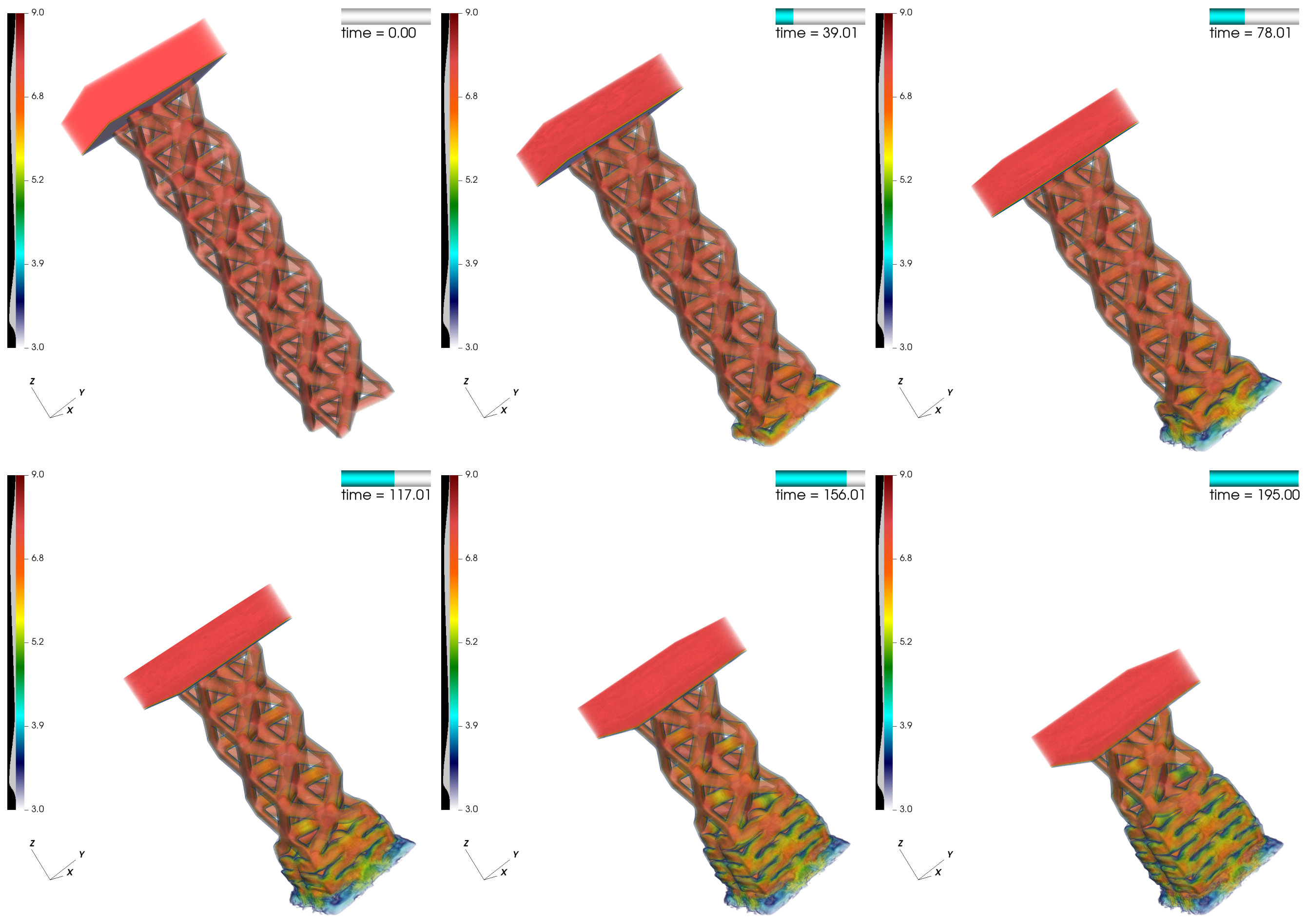


Figure 5.15: Snapshots of density for 3D MARBL high-order ALE simulations of a high-velocity impact of an octet-truss lattice made of Steel.

5.6 Shaped Charges

5.6.1 TOW2 Shaped Charge modeling

In this section we will discuss modeling of shaped charges in MARBL, which involve the use of a high explosive (HE) acting on a metal "liner" to create a fast moving jet. These calculations served as a way to familiarize early adopters with using the next generation code; in particular how to build a complex mesh, port it into MARBL and how to run a hydro simulation (ALE management etc...). Here we will show results of modeling the TOW2 shaped charge which is based on a Copper liner and has been extensively studied both computationally and experimentally (see for example [135]). We created the PMesh TOW2

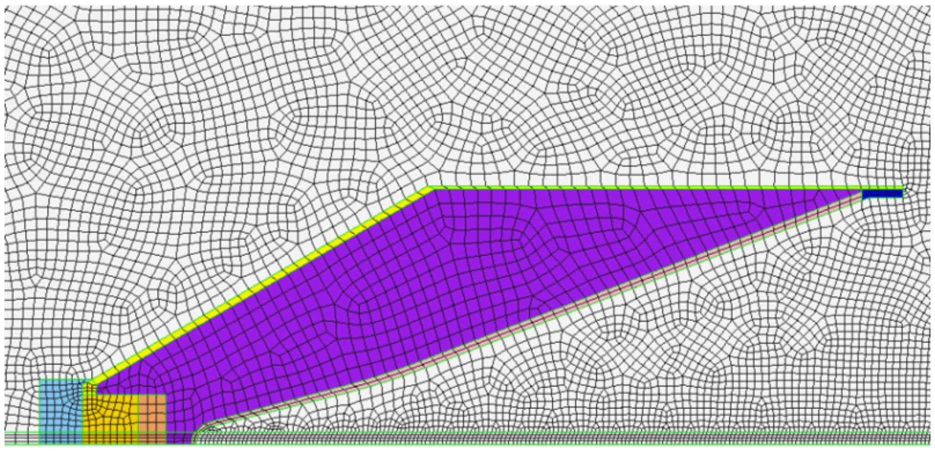


Figure 5.16: PMesh model of the TOW2 shaped charge. The mesh is a fully unstructured NURBS mesh.

.

model starting from an ARES silo restart file and utilizing the PMesh geometry-extract functionality. This capability provided the 2Drz spline pairs that characterize the geometry of the TOW2. The model is shown in Figure 5.16. Once the PMesh model is created it can be ported to MARBL where the user needs to define the materials with their EOS and strength model definitions, the initial conditions and also define the Lagrange surfaces to perform ALE during the hydro simulation. All the above are common practices within the MARBL deck repo. For the TOW2 for example, the user started with an existing BRL81 shaped charge deck, swaped the BRL81 materials with the TOW2 materials and after a few tweaks the model was ready to run. The MARBL TOW2 model is shown in Figure 5.17 for t = 0 μs. The user can interact with the mesh resolution in several ways. At the PMesh level the mesh resolution can be readily changed for a surface of interest. Once the PMesh model is ported into MARBL the user can either uniformly refine the whole model or refine a specific material or refine a specific region or all of the above. This is shown in Figure 5.17. At each level of refinement the chosen order of the mesh (Q_2-Q_1 etc) remains unchanged. In Figure 5.18 we show the jet evolution of the TOW2 shaped charge and also a comparison with a calculation made in the ARES code. The two codes are in good agreement and reproduce the experimental TOW2 speed of about 0.95 μs. We used the same LX14 JWL EOS in both calculations and the same EOS and material strength models. For this run we used a Lagrangian surface constraint to keep the inside of the Copper liner Lagrangian and in the near future we will also start utilizing the material adaptivity (similar to relaxation mesh weighting in ARES) to concentrate zones in the regions of interest. Finally in Figure 5.19 we are showing preliminary calculations of a half-symmetry 3D TOW2 model. Work is in progress in this arena to perform large 3D calculations of TOW2 shaped charges on various targets.

5.6.2 Rod impact on HE

The motivation to study cylindrical rods in MARBL is twofold. First, it is a simple problem that can test the code capabilities (e.g HE Reactive Flow) and also serve as a test problem for comparisons with other codes. Secondly, we know experimentally

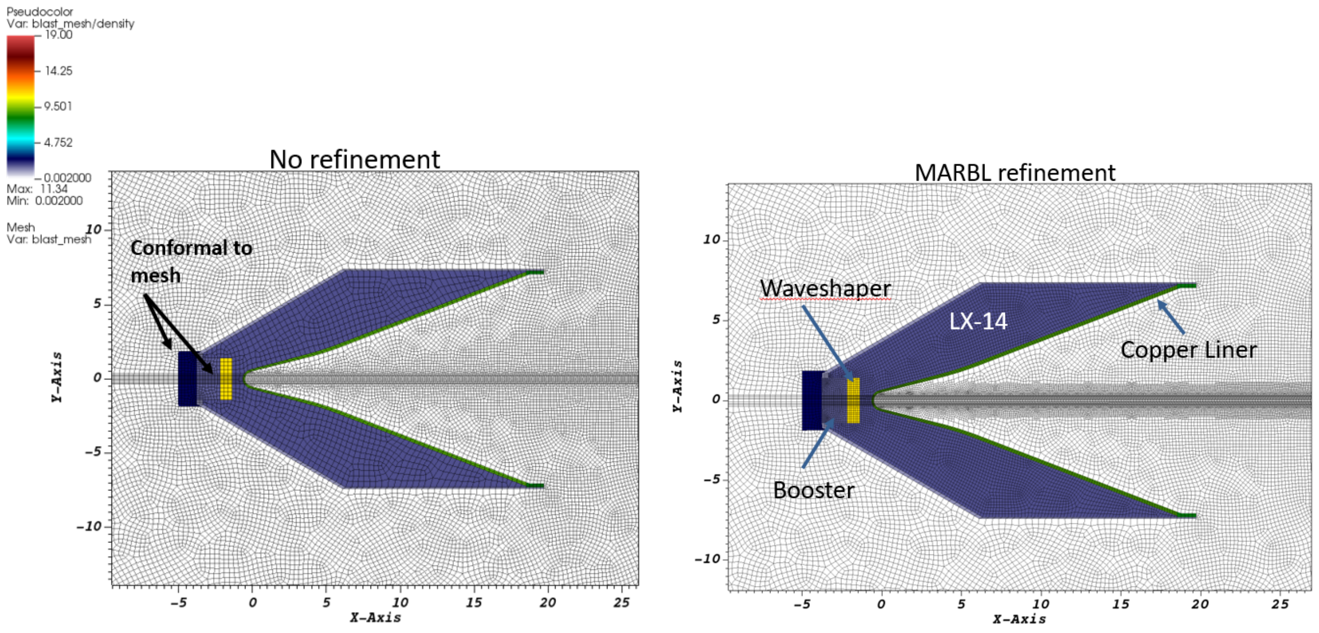


Figure 5.17: (left) MARBL model with the same resolution as the PMesh model. (Right) MARBL refined regions (HE, Copper liner, Booster) and also the path that the jet will be created.

.

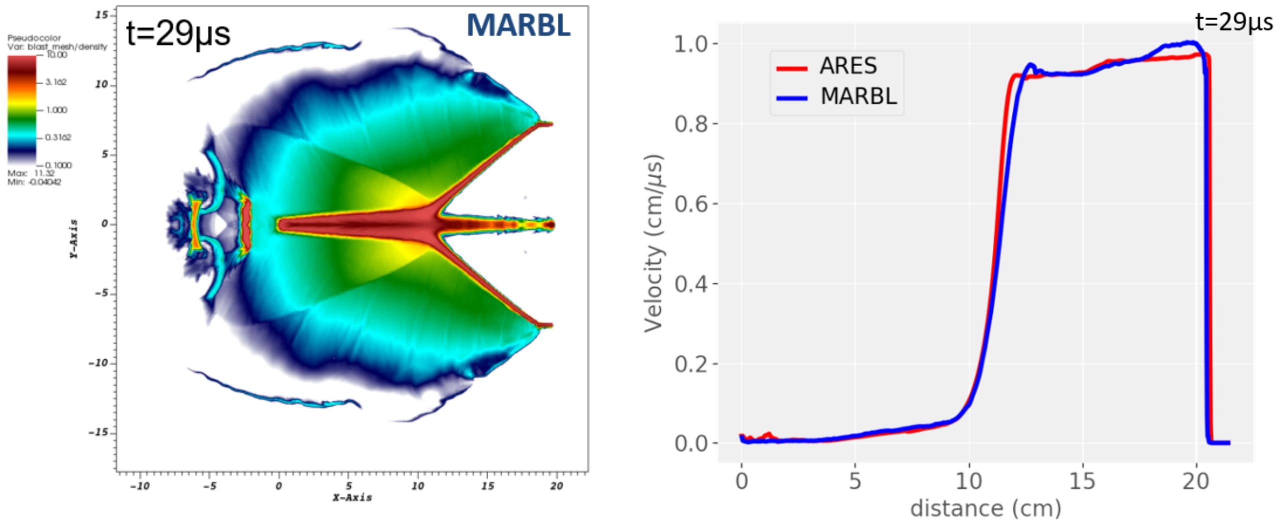


Figure 5.18: (left) MARBL TOW2 simulation at 29 μs. (Right) MARBL vs ARES speed lineout comparison.

.

that when a Copper jet travels through thick HE targets, its speed was observed to be reduced at a higher rate after about 6cm in the target. These are the Poulsen experiments as it is shown in Figure 5.20 Using MARBL we are studying the interaction of the Copper rod with an “infinite” target of LX14 HE. The HE initiation by the rod will be modeled with a Reactive Flow Ignition and Growth model.

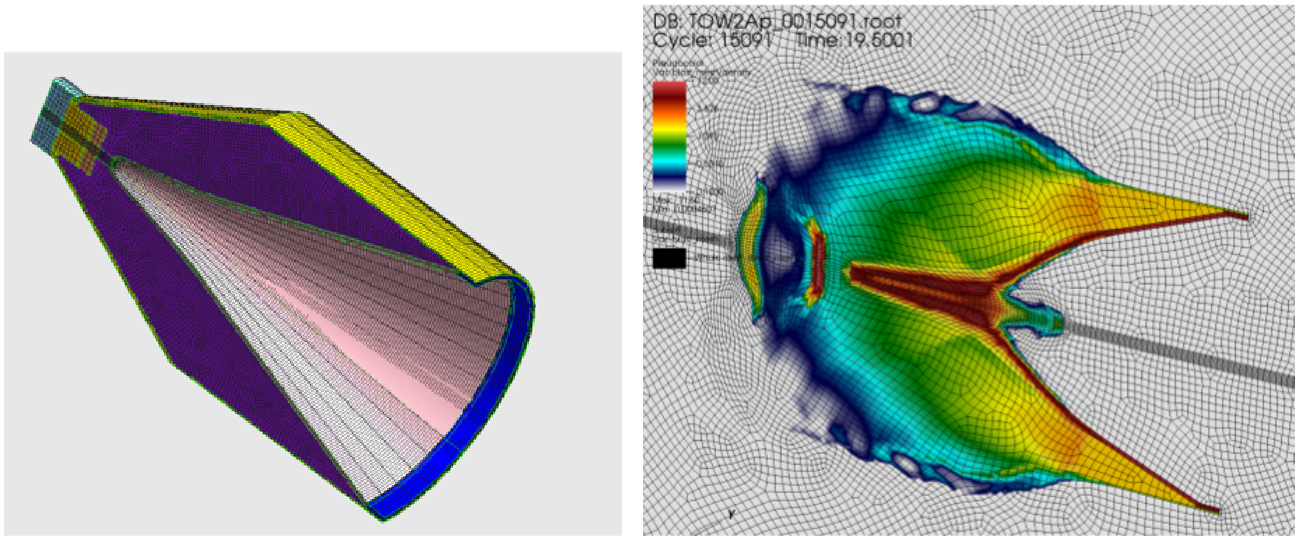


Figure 5.19: (left) 3D TOW2 PMesh model. (Right) MARBL 3D simulation (in progress) at 19.5 μs and early jet formation.
.

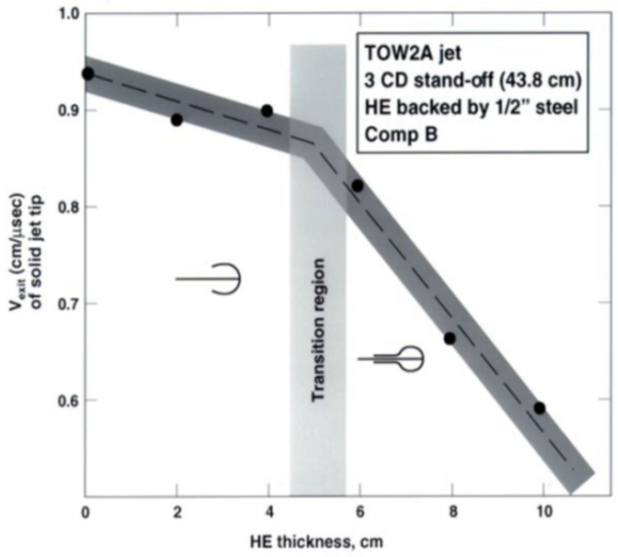


Figure 5.20: The Pulsen experimental data suggest that there is a transitional region when the jet is inside the HE, after which its speed is reduced at a higher rate. This reduction is attributed to the interaction of the front of the jet, also known as "backsplash", with its main body.
.

We created the rod-HE problem in PMesh and ported to MARBL as shown in Figure 5.22 We performed MARBL Q_2-Q_1 (2nd order) calculations of the rod/HE test problem for different resolutions. At 40 zones/cm the backsplash has been resolved and the next step would be to compare results with other codes (ARES, ALE3D) and also compare with analytic formula. This problem can serve as a MARBL testbed for studies of HE Reactive Flow models and studies of high-order mesh behavior. High-order curvilinear elements can offer a new insight in the jet/HE interaction study. and the test is perfect problem for designers that want

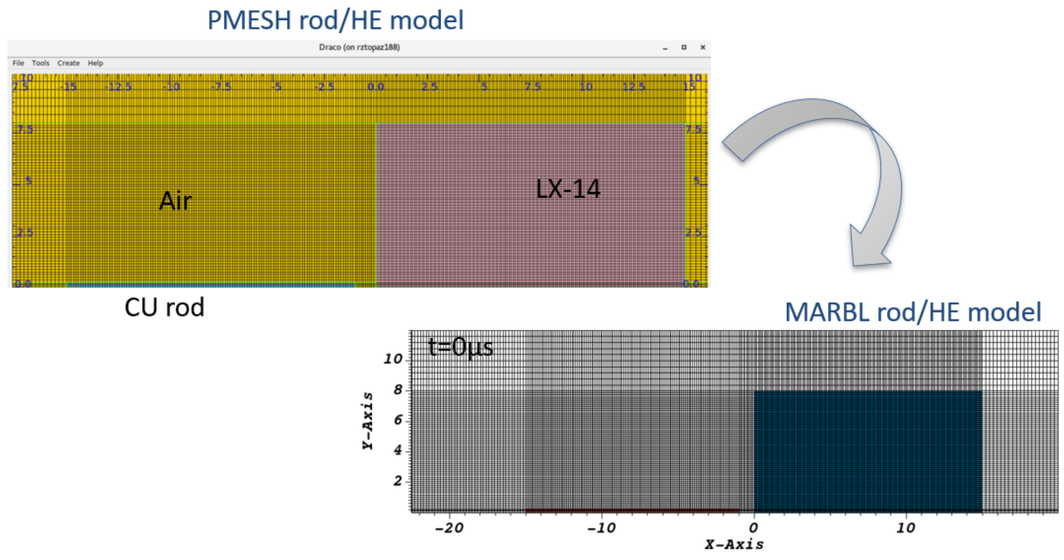


Figure 5.21: PMesh and MARBL models of the rod-HE problem.

.

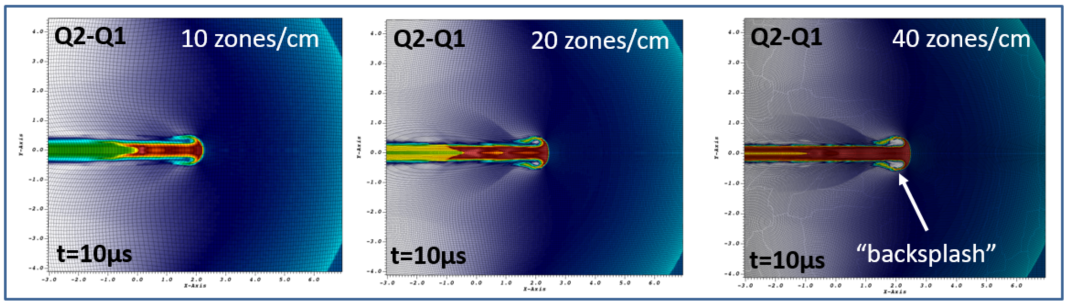


Figure 5.22: MARBL calculation resolution study of a 0.5 cmμs constant velocity Copper rod impinging on a large LX14 target.

.

to learn PMesh and MARBL.

5.7 Reacting Turbulent Jets

5.7.1 The temporally developing jet

The turbulent, temporally developing jet is a classical problem in fundamental turbulence research, due to the fact that physically realistic turbulence can be simulated in a relatively simple computational domain. As opposed to a spatially developing jet that issues from a nozzle and develops in space, the temporal jet develops in time and remains homogeneous in two directions. While a one-to-one comparison is not possible between the spatial and temporal jets, temporal jets maintain several key dynamics that exist in spatial jets, such as a breakdown to turbulence through the Kelvin-Helmholtz instability, and the existence of a fully-developed region where the statistics of the jet decay in a self-similar manner.

Indeed, turbulence simulation data which is fully developed and self-similar is useful in establishing a validation baselines for under-writing lower fidelity turbulence models such as Reynolds-Averaged Navier-Stokes (RANS) models. Since the simulation data discussed here are from a high-Reynolds number Direct Numerical Simulation (DNS) averaged quantities can directly inform model coefficients and settings. The turbulent jet also provides a platform for validating multi-physics RANS models, which may include chemical reactions and/or energy deposition. Details on these physical models are discussed below.

The turbulent jet is typically initialized with a turbulent core of fluid traveling at a mean velocity between two irrotational regions on either side. This results in two regions of shear where the velocity transitions from the jet to the irrotational region outside the jet. Additionally, it is necessary to slightly perturb the velocity field, or else the jet will not transition to turbulence. This is done by introducing isotropic fluctuations in the velocity field, such that the most unstable modes in the shear layer grow and eventually break down into turbulence.

One of the benefits of simulating a temporal jet is the boundary conditions; because the jet is homogeneous in the x and z directions, periodic boundary conditions can be used. In the cross-stream direction, y, it is common to use a free slip or periodic boundary conditions for incompressible jets. In the case of exothermic chemical reactions, an outflow boundary condition is required to allow the fluid in the computational domain to expand as the temperature increases.

The temporal jet is characterized by the jet Reynolds number,

$$Re_{jet} = \frac{\rho V_{jet} H}{\mu}, \tag{5.1}$$

where ρ and μ are the fluid density and dynamic viscosity, respectively, V_{jet} is the velocity difference between the jet and the outer flow, and H is the thickness of the jet.

The temporal development of the jet occurs in several stages. At early times, the initial perturbations inside the shear layer cause the most unstable modes to grow exponentially. This results in the onset of the Kelvin-Helmholtz instability, which further destabilizes the jet. As the instability increases, the flow becomes three-dimensional and quickly breaks down into turbulence. Once the turbulence has become fully developed, the statistics of the jet, i.e., the mean velocities and Reynolds stresses, evolve in a self-similar manner. Figure 5.23 visualizes the development of a temporally-developing jet through the stages discussed above.

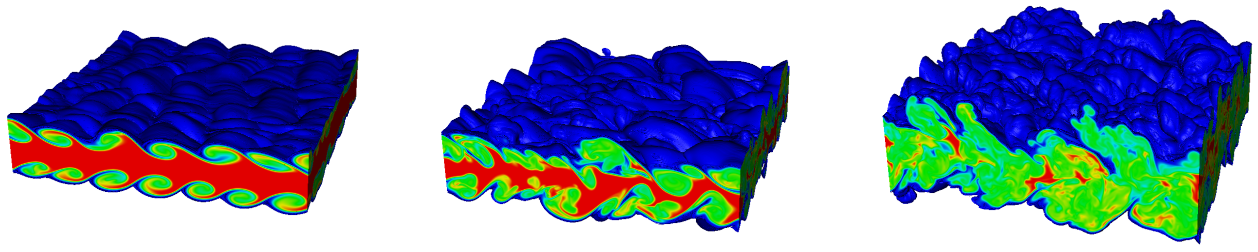


Figure 5.23: Snapshots of a temporally-developing jet during different stages of development.

5.7.2 Code validation

Validation of the code was performed by comparing the statistics in the self-similar region of the jet to established results from the literature. A constant density jet was simulated in a domain with 256x384x256 grid points (∼25 million) using eight GPUs and eight CPUs, distributed across two nodes. The jet Reynolds number was set to 3200, with a width of 1/6th of the domain size in the y direction. The computational setup was designed to match the conditions from a simulation performed by da Silva and Pereira [136].

Results from the temporally developing jet show good agreement with established results from the literature. The mean velocity, as a function of the cross-stream direction, is plotted in Figure 5.24a for the current GPU simulations as well as several previous studies. The mean velocity is normalized by the mean centerline velocity, and the cross-stream coordinate is normalized by the half-width of the jet. The rms of the fluctuating component of the streamwise velocity is plotted in Figure 5.24b. The axes are normalized in the same manner as the previous plot, and the results from previous studies are overlaid. Qualitatively, results from the current GPU simulations agree with previous simulations, despite there being some variance in the results. Importantly, our results agree very well with the simulations from daSilva and Pereira [136], which are the basis for the current conditions. This indicates that the GPU version of the eulerian hydrodynamics code is accurately simulating the dynamics of the temporally developing jet.

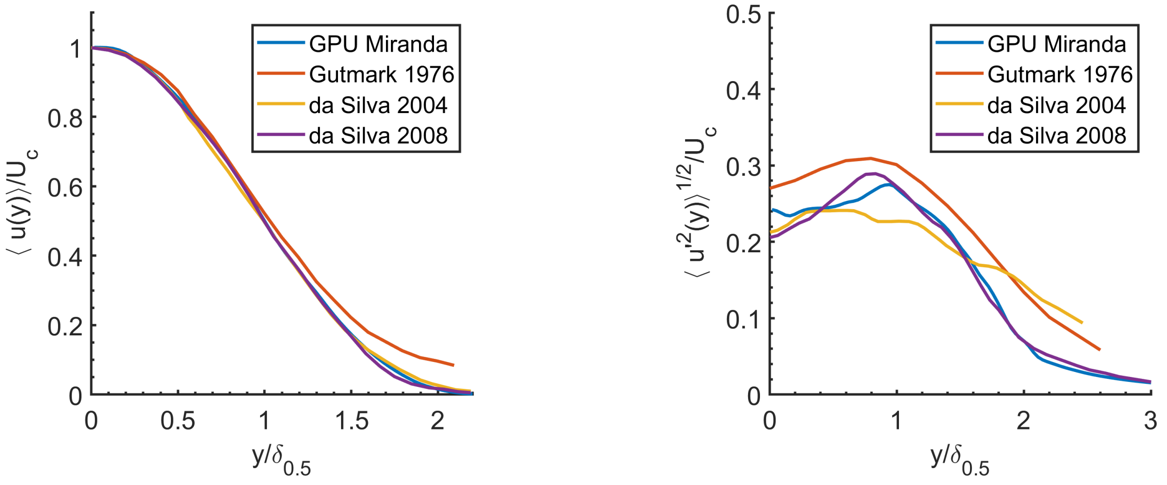


Figure 5.24: Plots of the a) mean velocity and b) rms of fluctuating streamwise velocity component in the self-similar region of the jet versus the streamwise coordinate. The velocities are normalized by the mean velocity at the centerline of the jet, and the streamwise coordinate is normalized by the half-width of the jet.

5.7.3 Algorithm development and code validation for reacting jets

A simple algorithm for calculating chemical source terms based on global reaction-rate kinetics was implemented into the Eulerian hydro solver. Because chemical reactions do not rely on spatial gradients, the chemical reaction mechanism can be validated with a zero-dimensional approximation, commonly referred to as 'reactor' models. Here we will assume a constant volume, fixed mass reactor that is perfectly stirred and adiabatic (see e.g., Ref. [137], Chapter 6). This reduces the governing equations to the following forms:

$$\frac{\mathrm{d}\rho}{\mathrm{d}t} = 0, \tag{5.2}$$

$$\frac{\mathrm{d}\mathbf{u}}{\mathrm{d}t} = 0, \tag{5.3}$$

$$\frac{\mathrm{d}E}{\mathrm{d}t} = \dot{\omega}_T, \tag{5.4}$$

$$\frac{\mathrm{d}Y_k}{\mathrm{d}t} = \dot{\omega}_k, \tag{5.5}$$

Here we see that for a fixed-mass, fixed-volume container, the density ρ must be constant. We can further simplify the equations by noting that $E = \rho(e + \mathbf{u}\cdot\mathbf{u}/2) = \rho e$. Assuming ideal gas law with a single adiabatic index, $e = RT/(\gamma - 1)$, where γ is the (constant) adiabatic index for all the gases. The value of R is given by $R = R_u/\bar{W}$, where the mixture averaged molecular weight is

$$\bar{W} = \left(\sum_{k=1}^{K} \frac{Y_k}{W_k} \right)^{-1}, \tag{5.6}$$

and R_u is the universal gas constant. Using these assumptions, we can develop a set of coupled equations for the species temperature and mass fractions:

$$\frac{\mathrm{d}T}{\mathrm{d}t} = \frac{\gamma - 1}{R\rho} \dot{\omega}_T. \tag{5.7}$$

$$\frac{\mathrm{d}Y_k}{\mathrm{d}t} = \frac{\dot{\omega}_k}{\rho}. \tag{5.8}$$

The reaction terms, $\dot{\omega}_k$ and $\dot{\omega}_T$ are obtained from the chemical reaction mechanism. For each species k, the source term is calculated by

$$\dot{\omega}_k = \sum_{j=1}^{M} W_k K_j \nu_{kj}, \tag{5.9}$$

where W_k is the molecular weight of species k, K_j is the rate of the j^{th} reaction, and ν_{kj} is the stoichiometric coefficient of the k^{th} species reacting in the j^{th} equation. The source term for species k is the sum of these parameters for all M number of reactions. All reactions are defined by the reaction rate, K_j, which is modeled with an Arrhenius exponential,

$$K_j = A_j \exp\left(\frac{E_j}{RT} \right) \prod_{k=1}^{N} [X_k]^{n_{kj}}. \tag{5.10}$$

A_j is the pre-exponential constant (determined by the chemical mechanism), E_j is the activation energy, $[X_k]$ is the molar concentration of the k^{th} species, N is the total number of species being tracked, and n_{kj} is the "concentration exponent," which is given on a per-species basis, for each reaction, by the chemical mechanism in use.

The source term for the energy equation, $\dot{\omega}_T$, is given by the species production (destruction) rates:

$$\dot{\omega}_T = -\sum_{k=1}^{N} \Delta h^{\circ}_{f,k} \dot{\omega}_k, \tag{5.11}$$

where $\Delta h^{\circ}_{f,k}$ is the enthalpy of formation of species k.

To validate the chemistry solver, the output from a zero-dimensional simulation in the eulerian hydro code was compared to the output from a Matlab routine that implements equations 5.7 and 5.8 above, using a built-in stiff ODE solver. For this test case, a single-step reaction between methane and air was used, based on tabulated values for the pre-exponential constant and activation energy [137]. Figure 5.25 shows the evolution of the mass fractions of methane, oxygen, carbon dioxide, and water vapor as the species react. Current results are shown as solid lines, and results from the Matlab ODE solver are shown as circles. The two simulations are in perfect agreement, which suggests that the chemical mechanism was implemented correctly.

5.7.4 High-Reynolds number scalability study

In addition to adding a chemical reaction model, it is also desirable to explore the Reynolds number dependence of the turbulent jet. Turbulence theory predicts that when the flow Reynolds number is sufficiently large, that the large-scale mean features of the flow become Reynolds number independent. That is to say, that even though the small, Komogorov length scales will directly depend on the Reynolds number, that the separation of length scales is so vast (High Reynolds number) that the large scales no longer directly depend on the smallest of scales.

The requirement for a separation of physical length-scales directly maps to the requirement for the separation of computational length-scales and computing flows at high Reynolds number becomes increasingly expensive. Indeed, Pope predicts that the computational cost as a function of Reynolds number scales as:

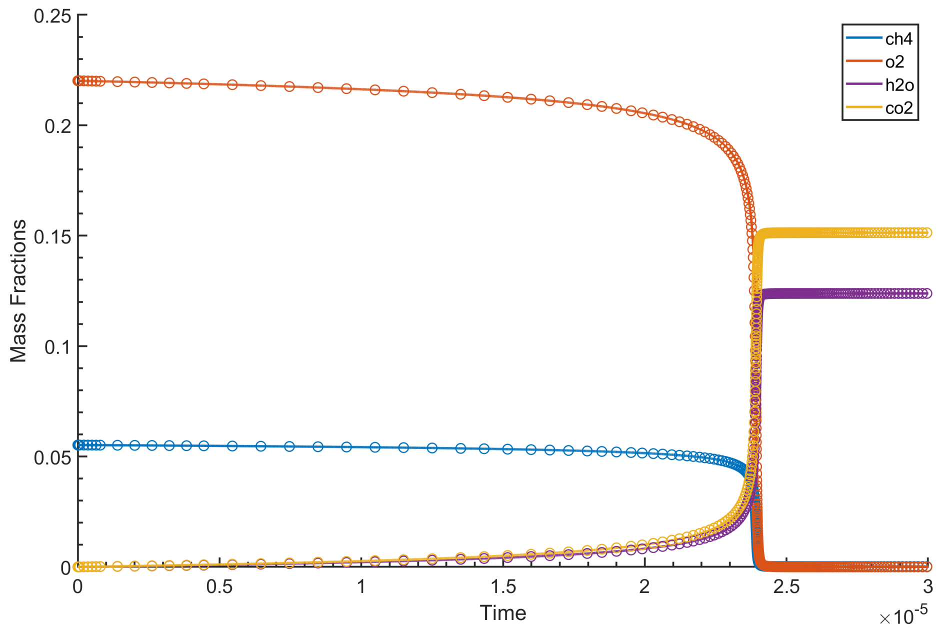


Figure 5.25: Plot of species mass fractions in a zero-dimensional reactor model. Species concentrations are shown for methane, oxygen, water vapor, and carbon dioxide. Solid lines belong to Miranda, while circles belong to the Matlab ODE solver.

$$Cost = N_{ops} = N_{cells} \times N_{t-steps} = (Re_L^{3/4})^3 \times Re_L^{1/2} \approx Re_L^{2.75}. \tag{5.12}$$

Therefore, to double the Reynolds number of the flow, the computation costs increase by a factor of 8.

The computational acceleration in throughput provided by the GPU offload will, therefore, allow for higher-Reynolds number flows to be achieved for a give number of computational nodes. In the case of the turbulent jet problem, we have been able to scale the problem out to over 8 billion computational zones and achieve DNS and LES results at very high Reynolds numbers.

As part of this physics application, a grid convergence study was conducted. Typical grid convergence studies double the spatial resolution while the number of computational nodes are increased a commensurate amount in order to keep the total number of degrees of freedom the same. This numerical study is useful for determining grid independence but when its done as described here, can also be used to evaluate the weak scaling of the code. Weak scaling is useful in determining how the scalability worsens as the total problem size grows. A more in-depth study of weak scaling is provided in previous sections for a different problem, but we include a scaling study for the turbulent jet for this problem here.

Below are time snapshots of 1 billion computational zone results, computed on the Lassen system (Sierra). The computational mesh size was 1024^3 and was computing using 128 nodes (512 GPUs) of Lassen. The calculation required approximately 48 hours of walltime to take 120k time steps to compute this late time calculation. The walltime required on an equivalent number of compute nodes on CTS1 type machines (Intel Xenon, eg.) would have taken roughly 300 hours.

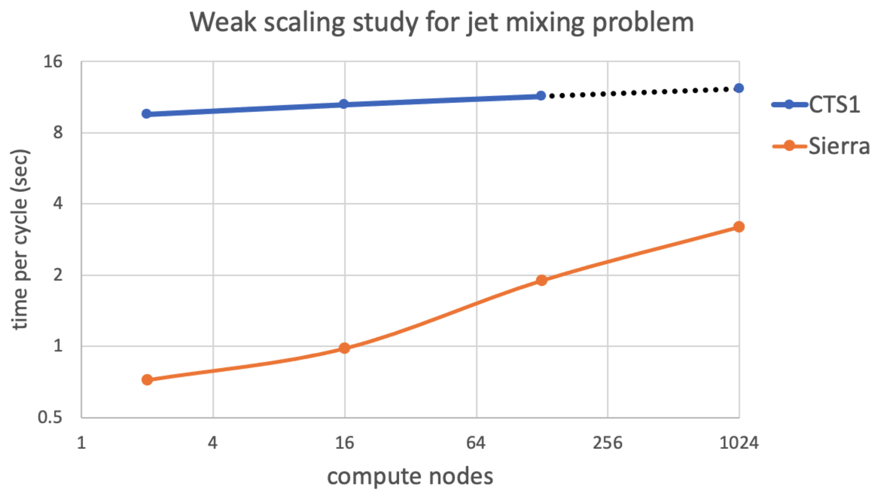


Figure 5.26: Weak scaling data from grid convergence study. Data compares the walltime per timestep vs nodes used on the Sierra and the CTS1 platforms. The study used approximately 8 million grid points per node on both platforms. The dashed black line and corresponding data point denotes a performance model for CTS1.

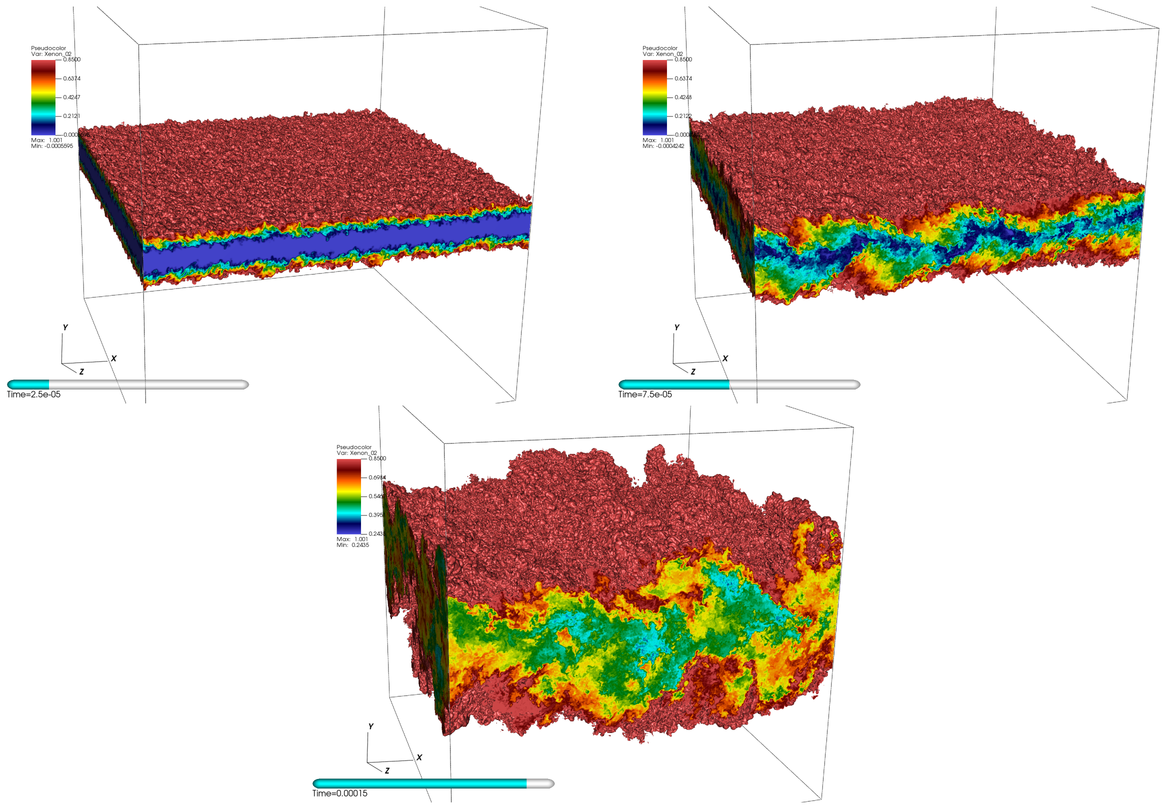


Figure 5.27: Plots of mass fraction isovolumes over time (left to right) of the heavy gas of the mixing shear layer. Evolution shows the growing instability and transition to broad-band turbulence.

Bibliography

[1] A. Black, R. Hornung, M. Kumbera, R. Neely, and Rieben R. Computer science recommendations for LLNL ASC Next-Gen code. Technical Report LLNL-TR-658622, Lawrence Livermore National Laboratory, 2014.

[2] D. Post and R. Kendall. Software project management and quality engineering for complex, coupled multiphysics, massively parallel computational simulations: Lessons learned from ASCI. *Int. J. High Perf. Computing Applications*, (4):399–416, 2004.

[3] D. J. Benson. Computational methods in Lagrangian and Eulerian hydrocodes. *Comput. Methods Appl. Mech. Engrg.*, 99:235–394, 1992.

[4] E. J. Caramana, D. E. Burton, M. J. Shashkov, and P. P. Whalen. The construction of compatible hydrodynamics algorithms utilizing conservation of total energy. *J. Comput. Phys.*, 146:227–262, 1998.

[5] A. J. Barlow, P-H. Maire, W. J. Rider, R. N. Rieben, and M. J. Shashkov. Arbitrary Lagrangian-Eulerian methods for modeling high-speed compressible multimaterial flows. *J. Comput. Phys.*, 322(C):603–665, 2016.

[6] R. N. Rieben. Prototype mixed finite element hydrodynamics capability in ARES. Technical Report LLNL-TR-405350, Lawrence Livermore National Laboratory.

[7] T.V. Kolev and R.N. R. N. Rieben. A tensor artificial viscosity using a finite element approach. *J. Comput. Phys.*, 228(22):8336–8366, 2009.

[8] V.A. Dobrev, T.E. Ellis, T.V. Kolev, and R.N. Rieben. Curvilinear finite elements for Lagrangian hydrodyanmics. *Internat. J. Numer. Methods Fluids*, 65(11-12):1295–1310, 2010.

[9] V.A. Dobrev, T.V. Kolev, and R.N. Rieben. High-order curvilinear finite element methods for Lagrangian hydrodynamics. *SIAM J. Sci. Comp.*, 34(5):B606–B641, 2012.

[10] V. A. Dobrev, T. E. Ellis, Tz. V. Kolev, and R. N. Rieben. High order curvilinear finite elements for axisymmetric Lagrangian hydrodynamics. *Computers and Fluids*, 83:58–69, 2013.

[11] V.A. Dobrev, T.V. Kolev, and R.N. Rieben. High-order curvilinear finite elements for elastic-plastic Lagrangian dynamics. *J. Comput. Phys.*, 257(B):1062–1080, 2014.

[12] R.W. Anderson, V.A. Dobrev, T.V. Kolev, and R.N. Rieben. Monotonicity in high-order curvilinear finite element arbitrary Lagrangian-eulerian remap. *Internat. J. Numer. Methods Fluids*, 77(5):249–273, 2014.

[13] V.A. Dobrev, T.V. Kolev, R.N. Rieben, and V.Z. Tomov. Multi-material closure model for high-order finite element Lagrangian hydrodynamics. *Internat. J. Numer. Methods Fluids*, 2016. in press.

[14] Robert W. Anderson, Veselin A. Dobrev, Tzanio V. Kolev, Robert N. Rieben, and Vladimir Z. Tomov. High-order multi-material ALE hydrodynamics. *SIAM J. Sci. Comp.*, 40(1):B32–B58, 2018.

[15] P. D. Bello-Maldonado, V. Z. Tomov, Tz. V. Kolev, and R. N. Rieben. A matrix-free hyperviscosity formulation for high-order ALE hydrodynamics. *Computers and Fluids*, 205(15), 2020.

[16] BLAST: High-order finite element Lagrangian hydrocode. `www.llnl.gov/CASC/blast`.

[17] H. Ockendon and J. Ockendon. *Waves and Compressible Flow*, volume 47 of *Texts in Applied Mathematics*. Springer-Verlag, 2004.

[18] J.A. Cottrell, T.J.R. Hughes, and Y. Bazilevs. *Isogeometric analysis: toward integration of CAD and FEA*. Wiley, 2009.

[19] E. J. Caramana and M. J. Shashkov. Elimination of artificial grid distortion and hourglass-type motions by means of Lagrangian subzonal masses and pressures. *J. Comput. Phys.*, 142(2):521–561, 1998.

[20] A. W. Cook and W. H. Cabot. Hyperviscosity for shock-turbulence interactions. *J. Comput. Phys.*, 203(2):379–385, 2005.

[21] Veselin A. Dobrev, Patrick Knupp, Tzanio V. Kolev, Ketan Mittal, and Vladimir Z. Tomov. The Target-Matrix Optimization Paradigm for high-order meshes. *SIAM J. Sci. Comp.*, 41(1):B50–B68, 2019.

[22] Veselin A. Dobrev, Patrick Knupp, Tzanio V. Kolev, Ketan Mittal, and Robert N. Rieben ad Vladimir Z. Tomov. Simulation-driven optimization of high-order meshes in ALE hydrodynamics. *Computers and Fluids*, 208(15), 2020.

[23] M. Berndt and N. Carlson. Using polynomial filtering for rezoing in ALE. Number LA-UR 11-05015. Presented at the 2011 MultiMat conference, Arcachon, France, 2011.

[24] Robert E. Tipton. Eulerian mixed zone pressure relaxation. private communication, August 1989.

[25] A.W. Cook. Artificial fluid properties for large-eddy simulation of compressible turbulent mixing. 19(055103), 2007.

[26] R. M. Lewis C. A. Kennedy, M. H. Carpenter. Low-storage, explicitrunge-kutta schemes for the compressible navier-stokes equations. *Appl. Numer. Math*, 35(177), 2007.

[27] R. Managan, J. Grondalski, and D. Stevens. Selene physics manual. Technical report, Lawrence Livermore National Laboratory, 2019.

[28] P. Castillo, R. N. Rieben, and D. A. White. FEMSTER: An object-oriented class library of high-order discrete differential forms. *ACM Trans. Math. Soft.*, 31, 2005.

[29] D. A. White, J. M. Koning, P. Castillo, and R. N. Rieben. Development and application of compatible discretizations of maxwell's equations. *The IMA Volumes in Mathematics and its Applications*, 142, 2006.

[30] Verification of high-order mixed FEM solution of transient magnetic diffusion problems. *IEEE Trans. Mag.*, 42(1):25–39, 2006.

[31] D. White. Nonphysical reverse currents in transient finite-element magnetics simulation. *IEEE Transactions on Magnetics*, 45:1973 – 1989, 2009.

[32] A. Sandu. A class of multirate GARK methods. *SIAM J. Num. Anal.*, 57:2300–2327, 2019.

[33] S. Roberts, A. Sarshar, and A. Sandu. Coupled multirate infinitesimal GARK schemes for stiff systems with multiple timescales. *SIAM J. Sci. Comp.*, 42:1609–1638, 2020.

[34] Axom. `https://github.com/LLNL/axom`.

[35] Shroud. `https://github.com/LLNL/shroud`.

[36] Conduit. `https://github.com/LLNL/comduit`.

[37] Introducing json. `https://www.json.org/json-en.html`.

[38] Yaml. `https://en.wikipedia.org/wiki/YAML`.

[39] Numpy. `https://numpy.org`.

[40] K. Weiss, G. Zagaris, R. Rieben, and A. Cook. Spatially accelerated shape embedding in multimaterial simulations. In S. Canann, editor, *Proceedings 25th International Meshing Roundtable*, IMR '16, Washington, D.C., September 27–30 2016.

[41] J. Grandy. Conservative remapping and region overlays by intersecting arbitrary polyhedra. *J. Comput. Phys.*, 148(2):433–466, 2009.

[42] A. Kunen C. Harrison, B. Ryujin. A scientific data exchange library for HPC simulations. `https://llnl-conduit.readthedocs.io/en/latest/_downloads/74e2e3f5ada90a7aa46727de2d8ecc5c/2016_07_13_scipy2016_conduit_slides.pdf`. Accessed: 2020-10-14.

[43] M. Miller. Silo user's guide (revision: July 2014). `https://wci.llnl.gov/content/assets/docs/simulation/computer-codes/silo/LLNL-SM-654357.pdf`. Accessed: 2020-10-14.

[44] Stuart I. Feldman. *Make — A Program for Maintaining Computer Programs. Software: Practice and Experience*, 9(4):255–265, 1979. `https://onlinelibrary.wiley.com/doi/epdf/10.1002/spe.4380090402`.

[45] Make (software). `https://en.wikipedia.org/wiki/Make_(software)`. Accessed: 2019-12-26.

[46] Richard M. Stallman, Roland McGrath, and Paul D. Smith. *GNU Make: A Program for Directing Recompilation*. Free Software Foundation, 2016. `https://www.gnu.org/software/make/manual/`.

[47] Paul D. Smith. *Metaprogramming Make I — Evaluation and Expansion*. `http://make.mad-scientist.net/evaluation-and-expansion/`. Accessed: 2019-12-13.

[48] How to use variables. `https://www.gnu.org/software/make/manual/make.html#Using-Variables`. Accessed: 2019-12-13.

[49] Computed variable names. `https://www.gnu.org/software/make/manual/html_node/Computed-Names.html`. Accessed: 2019-12-13.

[50] Graph visualization software. `https://www.graphviz.org/`. Accessed: 2019-12-13.

[51] Roberto Ierusalimschy, Luiz Henrique de Figueiredo, and Waldemar Celes. *Lua 5.1 Reference Manual*. Lua.Org, 2006. `https://www.lua.org`.

[52] Thomas Williams, Colin Kelley, and many others. *Gnuplot 5.2.8: an interactive plotting program*. `http://www.gnuplot.info/`, December 2019.

[53] Fortran-modules. `https://www.tutorialspoint.com/fortran/fortran_modules.htm`. Accessed: 2019-12-19.

[54] Eric Melski. Rules with multiple outputs in gnu make. `https://www.cmcrossroads.com/article/rules-multiple-outputs-gnu-make`. Accessed: 2019-12-19.

[55] L.E. Busby. *A Note on Compiling Fortran*. `https://e-reports-int.llnl.gov/pdf/891106.pdf`. Technical Report LLNL-TR-738243, September, 2017.

[56] Brian Foote and Joseph Yoder. Big ball of mud. `http://www.laputan.org/mud/`. Accessed: 2019-12-23.

[57] A. Mokhov, N. Mitchell, and S. Jones. *Build Systems à la Carte*. `https://www.microsoft.com/en-us/research/uploads/prod/2018/03/build-systems-final.pdf`. Accessed: 2019-12-20.

[58] A. Mokhov, N. Mitchell, S. Jones, and S. Marlow. *Non-recursive Make Considered Harmful*. `https://www.microsoft.com/en-us/research/wp-content/uploads/2016/03/hadrian.pdf`. Accessed: 2019-12-20.

[59] Matt Craighead. What's wrong with gnu make? `http://www.conifersystems.com/whitepapers/gnu-make/`. Accessed: 2019-12-23.

[60] Adrian Neagu. What is wrong with make? `http://freshmeat.sourceforge.net/articles/what-is-wrong-with-make`. Accessed: 2019-12-23.

[61] Rocky Bernstein. *Remake – GNU Make with comprehensible tracing and a debugger*. `http://bashdb.sourceforge.net/remake/`. Accessed: 2019-12-20.

[62] John Graham-Cumming. *The GNU Make Book*. No Starch Press, 2015. `https://nostarch.com/gnumake`.

[63] John Graham-Cumming. Updated list of my gnu make articles. `https://blog.jgc.org/2013/02/updated-list-of-my-gnu-make-articles.html`. Accessed: 2019-12-20.

[64] Paul D. Smith. Collection of gnu make papers. `http://make.mad-scientist.net/`. Accessed: 2019-12-20.

[65] Address sanitizer. `https://clang.llvm.org/docs/AddressSanitizer.html`.

[66] Leak sanitizer. `https://clang.llvm.org/docs/LeakSanitizer.html`.

[67] Clang coverage. `https://clang.llvm.org/docs/SourceBasedCodeCoverage.html`.

[68] Memory sanitizer. `https://clang.llvm.org/docs/MemorySanitizer.html`.

[69] Clang format. `https://clang.llvm.org/docs/ClangFormat.html`.

[70] Clang tidy. `https://clang.llvm.org/extra/clang-tidy`.

[71] Sphinx. `https://www.sphinx-doc.org/en/master/index.html`.

[72] Raja. `https://github.com/LLNL/RAJA`.

[73] D. A. Beckingsale, J. Burmark, R. Hornung, H. Jones, W. Killian, A. J. Kunen, O. Pearce, P. Robinson, B. S. Ryujin, and T. R. W. Scogland. RAJA: Portable performance for large-scale scientific applications. *2019 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*, pages 71–81, 2019.

[74] Umpire. `https://github.com/LLNL/Umpire`.

[75] D. Beckingsale, M. McFadden, J. Dahm, R. Pankajakshan, and R. Hornung. Umpire: Application-focused management and coordination of complex hierarchical memory. *IBM Journal of Research and Development*, pages 1–10, 2019.

[76] Cuda c++ programming guide. `https://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf`.

[77] Hip programming guide. `https://rocmdocs.amd.com/en/latest/Programming_Guides/HIP-GUIDE.html`.

[78] David Boehme, Todd Gamblin, David Beckingsale, Peer-Timo Bremer, Alfredo Gimenez, Matthew LeGendre, Olga Pearce, and Martin Schulz. Caliper: Performance Introspection for HPC Software Stacks. In *The International Conference for High Performance Computing, Networking, Storage, and Analysis (SC'16)*, Salt Lake City, Utah, USA, November 13-18, 2016.

[79] David Boehme. Ubiquitous Performance Analysis. In *Scalable Tools Workshop*, Tahoe, California, USA, July, 2019.

[80] Abhinav Bhatele, Stephanie Brink, and Todd Gamblin. Hatchet: Pruning the overgrowth in parallel profiles. In *The International Conference for High Performance Computing, Networking, Storage and Analysis (SC'19)*, Denver, Colorado, November 17-22, 2019.

[81] LLNL/IREP. `https://github.com/LLNL/irep`. Accessed: 2019-12-26.

[82] Fortran iso c binding. `http://fortranwiki.org/fortran/show/iso_c_binding`. Accessed: 2019-12-26.

[83] Jonathan Corney and Theodore Lim. *3D Modeling with ACIS*. Saxe-Coburg Publications, 2002.

[84] C. Blanc and C. Shlick. Accurate parameterization of conics by NURBS. *IEEE Computer Graphics and Applications*, 16(6):64–71, 1996.

[85] Knut Mørken, Martin Reimers, and Christian Schulz. Computing intersections of planar spline curves using knot insertion. *Computer Aided Geometric Design*, 26:351–366, 03 2009.

[86] Pseudonym. Splitting of NURBS curves. `http://computergraphics.stackexchange.com/a/352`. Accessed: 2016-09-15.

[87] Benjamin Marussig, Juergen Zechner, Gernot Beer, and Thomas-Peter Fries. Stable isogeometric analysis of trimmed geometries. *Computer Methods in Applied Mechanics and Engineering*, 316:497–521, 04 2017.

[88] Robert Schmidt, Roland Wüchner, and Kai-Uwe Bletzinger. Isogeometric analysis of trimmed NURBS geometries. *Computer Methods in Applied Mechanics and Engineering*, 241-–244:93—-111, 10 2012.

[89] Les A. Piegl and Wayne Tiller. *The NURBS Book (2nd ed.)*. Springer, 1997.

[90] emscripten. `https://emscripten.org`.

[91] Kenneth Moreland, Christopher Sewell, William Usher, Lita Lo, Jeremy Meredith, David Pugmire, James Kress, Hendrik Schroots, Kwan-Liu Ma, Hank Childs, Matthew Larsen, Chun-Ming Chen, Robert Maynard, and Berk Geveci. VTK-m: Accelerating the visualization toolkit for massively threaded architectures. *IEEE Computer Graphics and Applications (CG&A)*, 36(3):48–58, May/June 2016.

[92] Lawrence Livermore National Laboratory. Ascent documentation, 2020.

[93] Matthew Larsen, James Ahrens, Utkarsh Ayachit, Eric Brugger, Hank Childs, Berk Geveci, and Cyrus Harrison. The ALPINE in situ infrastructure: Ascending from the ashes of strawman. In *Proceedings of the In Situ Infrastructures on Enabling Extreme-Scale Analysis and Visualization*, ISAV'17, pages 42–46, New York, NY, USA, 2017. ACM.

[94] Matthew Larsen, Eric Brugger, Hank Childs, Jim Eliot, Kevin Griffin, and Cyrus Harrison. Strawman: A batch in situ visualization and analysis infrastructure for multi-physics simulation codes. In *Proceedings of the First Workshop on In Situ Infrastructures for Enabling Extreme-Scale Analysis and Visualization*, ISAV2015, pages 30–35, New York, NY, USA, 2015. ACM.

[95] ALPINE ECP project, 2020.

[96] P. G. Ciarlet. *The Finite Element Method for Elliptic Problems*. North-Holland, 1978.

[97] Claes Johnson. *Numerical solution of partial differential equations by the finite element method*. Cambridge University Press, 1987.

[98] L. Demkowicz. *Computing with hp-Adaptive Finite Elements. Volume I: One and Two Dimensional Elliptic and Maxwell Problems*. Chapman Hall CRC, 2006.

[99] Susanne C. Brenner and L. Ridgway Scott. *The Mathematical Theory of Finite Element Methods*. Springer New York, 2008.

[100] P. Solin, K. Segeth, and I. Dolezel. *Higher-Order Finite Element Methods*. Chapman Hall CRC, 2002.

[101] Michel Deville, Paul Fischer, and Ernest Mund. *High-order methods for incompressible fluid flow*. Cambridge University Press, 2002.

[102] Douglas N. Arnold, Franco Brezzi, Bernardo Cockburn, and L. Donatella Marini. Unified analysis of discontinuous Galerkin methods for elliptic problems. *SIAM Journal on Numerical Analysis*, 39(5):1749–1779, May 2001.

[103] R. Anderson, J. Andrej, A. Barker, J. Bramwell, J.-S. Camier, J. Cerveny V. Dobrev, Y. Dudouit, A. Fisher, Tz. Kolev, W. Pazner, M. Stowell, V. Tomov, I. Akkerman, J. Dahm, D. Medina, and S. Zampini. MFEM: A modular finite element library. *Computers & Mathematics with Applications*, 2020.

[104] MFEM: Modular finite element methods library. `mfem.org`.

[105] Douglas N. Arnold, Richard S. Falk, and Ragnar Winther. Differential complexes and stability of finite element methods I. The de Rham complex. In *Compatible Spatial Discretizations*, pages 23–46. Springer New York, 2006.

[106] CEED: Center for Efficient Exascale Discretizations in the U.S. Department of Energy's Exascale Computing Project. `https://ceed.exascaleproject.org`.

[107] Steven A Orszag. Spectral methods for problems in complex geometries. *Journal of Computational Physics*, 37(1):70 – 92, 1980.

[108] Steffen Müthing, Marian Piatkowski, and Peter Bastian. High-performance implementation of matrix-free high-order discontinuous galerkin methods. 11 2017.

[109] Martin Kronbichler and Katharina Kormann. Fast matrix-free evaluation of discontinuous galerkin finite element operators. *ACM Trans. Math. Softw.*, 45(3), August 2019.

[110] Mark Ainsworth, Gaelle Andriamaro, and Oleg Davydov. Bernstein–Bézier finite elements of arbitrary order and optimal assembly procedures. *SIAM Journal on Scientific Computing*, 33(6):3087–3109, January 2011.

[111] Douglas N. Arnold, Richard S. Falk, and Ragnar Winther. Finite element exterior calculus, homological techniques, and applications. *Acta Numerica*, 15:1–155, 2006.

[112] J. C. Nedelec. Mixed finite elements in $\mathbb{R}^3$. *Numerische Mathematik*, 35(3):315–341, September 1980.

[113] O Sahni, XJ Luo, KE Jansen, and MS Shephard. Curved boundary layer meshing for adaptive viscous flow simulations. *Finite Elements in Analysis and Design*, 46(1):132–139, 2010.

[114] Veselin A. Dobrev, Tzanio V. Kolev, and Robert N. Rieben. High-order curvilinear finite element methods for Lagrangian hydrodynamics. *SIAM Journal on Scientific Computing*, 34(5):B606–B641, 2012.

[115] Xiao-Juan Luo, MarkS. Shephard, Lie-Quan Lee, Lixin Ge, and Cho Ng. Moving curved mesh adaptation for higher-order finite element simulations. *Engineering with Computers*, 27(1):41–50, 2011.

[116] Patrick Knupp. Introducing the target-matrix paradigm for mesh optimization by node movement. *Engineering with Computers*, 28(4):419–429, 2012.

[117] Veselin Dobrev, Patrick Knupp, Tzanio Kolev, Ketan Mittal, and Vladimir Tomov. The Target-Matrix Optimization Paradigm for high-order meshes. *SIAM Journal on Scientific Computing*, 41(1):B50–B68, 2019.

[118] Veselin Dobrev, Patrick Knupp, Tzanio Kolev, and Vladimir Tomov. Towards simulation-driven optimization of high-order meshes by the Target-Matrix Optimization Paradigm. In Xevi Roca and Adrien Loseille, editors, *27th International Meshing Roundtable*, volume 127 of *Lecture Notes in Computational Science and Engineering*, pages 285–302. Springer International Publishing, Cham, 2019.

[119] A. Barlow, R. Hill, and M. J. Shashkov. Constrained optimization framework for interface-aware sub-scale dynamics closure model for multimaterial cells in Lagrangian and arbitrary Lagrangian-Eulerian hydrodynamics. *Journal of Computational Physics*, 276(0):92–135, 2014.

[120] L. Demkowicz, J.T. Oden, W. Rachowicz, and O. Hardy. Toward a universal h-p adaptive finite element strategy, part 1. constrained approximation and data structure. *Computer Methods in Applied Mechanics and Engineering*, 77(1):79 – 112, 1989.

[121] T. Schönfeld. Adaptive mesh refinement methods for three-dimensional inviscid flow problems. *International Journal of Computational Fluid Dynamics*, 4(3-4):363–391, 1995.

[122] V. Heuveline and F. Schieweck. H1-interpolation on quadrilateral and hexahedral meshes with hanging nodes. *Computing*, 80(3):203–220, Jul 2007.

[123] Jakub Cerveny, Veselin Dobrev, and Tzanio Kolev. Non-conforming mesh refinement for high-order finite elements. *SIAM Journal on Scientific Computing*, 41(4):C367–C392, 2019.

[124] S. Aluru and F.E. Sevilgen. Parallel domain decomposition and load balancing using space-filling curves. In *Proceedings Fourth International Conference on High-Performance Computing*. IEEE Comput. Soc, 1997.

[125] E. G. Boman, U. V. Catalyurek, C. Chevalier, and K. D. Devine. The Zoltan and Isorropia parallel toolkits for combinatorial scientific computing: Partitioning, ordering, and coloring. *Scientific Programming*, 20(2):129–150, 2012.

[126] S.K. Lele. Compact finite difference schemes with spectral-like resolution. *J. Comput. Phys.*, 103:16–42, 1992.

[127] W. Cabot A.W. Cook. The mixing transition in rayleigh-taylor instability. 511:333–362, 2004.

[128] pyranda. `https://github.com/LLNL/pyranda`.

[129] R. D. Hornung and S. R. Kohn. Managing application complexity in the samrai object-oriented framework. *Concurrency and Computation: Practice and Experience*, 14:347–368, 2002.

[130] R. Tipton. An icf test problem for single group radiation-hydrodynamics codes. (Unpublished report).

[131] S. H. Langer, T. Parker, and R. N. Rieben. Simulating high energy density experiments using marbl. Technical Report LLNL-PROC-746726, Lawrence Livermore National Laboratory.

[132] O. A. Hurricane, J. F. Hansen, H. F. Robey, B. A. Remington, M. J. Bono, E. C. Harding, R. P. Drake, and C. C. Kuranz. A high energy density shock driven kelvin–helmholtz shear layer experiment. *Physics of Plasmas*, 16(5), 2009.

[133] F. Najjar, R. Rieben, and L. Kirsch. High-order lagrangian hydrodynamics computations of surface perturbations in shock-driven metals. Number https://doi.org/10.1063/12.0000915. AIP Conference Proceedings, 2020.

[134] S. J. Ali et. al. Probing the seeding of hydrodynamic instabilities from nonuniformities in ablator materials using 2d velocimetry. *Physics of Plasmas*, 25, 2018.

[135] W. Walters. A brief history of shaped charges. volume 1, pages 3–10. 24th International Symposium on Ballistics, New Orleans, LA, 2008.

[136] Carlos B da Silva and José CF Pereira. Invariants of the velocity-gradient, rate-of-strain, and rate-of-rotation tensors across the turbulent/nonturbulent interface in jets. *Physics of Fluids*, 20(5), 2008.

[137] Stephen R Turns. *Introduction to Combustion*. McGraw-Hill Companies, 1996.

Appendix A

Developer survey

A.1 Full responses to developer survey

This appendix contains the full responses to our developer survey (presented in summary form in Section 3.13). We have anonymized the developers, who we refer to as 'Dev 01', 'Dev02', etc... and have lightly edited the responses for clarity.[1]

A.1.1 Survey Questions

We asked the following five questions to our developers:

1. Was the documentation for cloning/building/running and testing the code sufficient to accomplish your tasks?
2. Was the build system sufficiently flexible to integrate your package into the code?
3. Was it clear how to test / run your development code to verify its functionality?
4. How many code team members did you need to consult with in order to accomplish your work?
5. What recommendations do you have about things that are unclear or need improvement?

A.1.2 Survey Responses

Dev 01 Developer 1 is a computer scientist who worked on the code from April 2017 until May 2020 to integrate the Overlink package and Conduit-Carter functionality into MARBL.

1. The documentation for building, running, and testing the MAPP codes has always been very complete; I don't believe I've ever had any issues outside of synchronizing the constituent code repositories (which the project has made very easy through the availability of utility scripts).
2. Integrating my particular code (i.e. Overlink) was a bit clumsy at first since all updated versions had to be deployed as a new third-party archive, but the more recent introduction of "second-party" libraries has smoothed this process over considerably.

[1]We've used the following initials for specific MAPP developers mentioned in the survey responses: L.B. – Lee Busby; J.G. – John Grondalski; R.R. – Rob Rieben; A.S. – Aaron Skinner; T.S. – Tom Stitt; K.W. – Kenny Weiss;

3. Yes, the documentation on running/testing the code has always made it very easy to run regression tests.
4. At the start of my time with MAPP, I needed to communicate with 2 or more people (i.e. L.B., K.W., and occasionally also T.S.) in order to work through build processes/integrations, though mine was a frontier use case. Since this use case has become better supported, I've generally had to consult with one person (i.e. K.W.) or fewer.
5. More documentation on how to build the codes more quickly using pre-built libraries (or "package farms," as I believe they're called) would be a welcome addition.

Dev 02 Developer 2 is a physicist who worked on the code from February 2018 through June 2019 to integrate the Selene package into MARBL.

1. I must admit that I just asked you (K.W.) how to do this. The human interaction was much more direct and efficient than going through the documentation. I also had J.G. as a conduit for performing the direct integration.
2. I liked how 3rd party libraries were built into the code design upfront. The common use of CMAKE and BLT was wonderful as I could just offload 90% of building my library into the code without having to worry about compiler options and other details being inconsistent.
3. Again J.G. managed this for me. My experience supporting at least 3 independent code projects is that it was invaluable to have a staff member that was familiar with my library as one of the core developers. J.G. setup a subset of tests for me to run and I knew that the core developer could then push things more completely through their paces on multiple platforms.
4. The great thing about the code was that it was a team with interlocking expertise. Sometimes that was inconvenient, but it ensured multiple perspectives were applied to the integration. Each of them provided unique improvements to the overall project. I would say that it encompassed about 5-6 people.
5. My only issue was having to build multiple third-party libraries from scratch every time that I did a fresh build. Several of the very slowly evolving libraries could probably have cached builds. I have a feeling that this has been fixed by now since I worked on the project.

Dev 03 Developer 3 is a computer scientist who worked on the code from September 2018 through November 2020 to integrate the Ascent and Devil Ray in-situ visualization packages into MARBL.

1. Almost. The only item that I thought could be documented a little better was the `package.status=auto` and how it works. That said I was easily able to proceed with a little help.
2. Yes. The flexibility made the integration pretty easy.
3. Yes.
4. 1-2.
5. I think could be a little more detail about how the entire build system works under the covers. When something goes wrong, it would allow developers to reason better about what might be happening without consulting a MAPP developer. Ultimately, the MAPP build system is one of the most complete and well tested build systems I have used in my experience with 8 other simulation codes.

Dev 04 Developer 4 is a computer scientist who worked on the code from November 2018 through May 2019 to integrate Caliper and related packages into MARBL.

1. The docs were really good, and just written documentation got us 95% of the way there. For the small corner cases that popped up, I just email you and you know what to do.

2. Yes. I hazily remember some confusion over which flags should be used to enable which packages, but a few emails sorted that out.
3. Yeah, though I was given a simple test problem to run, I think you (K.W.) really did the more sophisticated testing.
4. Two. K.W. for technical stuff; K.W. and R.R. for "is this doing what you want?"
5. This is based on some pretty hazy memories, but there were really extensive docs for building at the time I tried it, but what proved most helpful was just using some other package as a model. I could see having an example package that exercises a lot of the build system being a handy tool.

Dev 05 Developer 5 is a computer scientist who worked on the code from July 2019 through November 2019 in support of the Spot and Hatchet packages.

1. Yes.
2. Yes.
3. Yes.
4. One (not many questions).
5. Everything was clear.

Dev 06 Developer 6 is a computer scientist who worked on the code from July 2019 through November 2019 to integrate the Adiak package and improve support for Caliper and Spot within MARBL.

1. For my initial build, yes the documentation was generally sufficient. I only needed a couple hours of effort to start from scratch and get to a build of MAPP/MARBL. I vaguely recollect that I had some compiler-specific speedbumps (it might have been an incorrect Intel compiler version?), but I was able to work through them. Documentation on recommended compilers would have helped with that.
2. Yes. I mostly just grep'd MAPP for the names of other packages that had been added, and duplicated that infrastructure for Caliper and Adiak. I recollect that mostly worked. There wasn't anything so special about my packages to make that difficult.
3. I didn't figure out running/testing from the online documentation, and I had to send an email to you (K.W.) for recommendations. You pointed me at a small problem I could use for quick iteration testing, and you helped me with the right command line for running tests. With that email help I was able to figure it out.
4. Two. I worked with you (K.W.), during development. Then I iterated with T.S. a bit during the Pull Request [to merge the code].
5. I'd overall rate the build system a 'B' experience. A more standardized and familiar build system would have helped a lot. When everything worked it was fine, but when things didn't work it was tough to debug. There were a few specific things:
   (a) I had specific challenges around Shroud. I needed to change a parameter's type, and it wasn't intuitive how to rebuild MAPP after the change. I could build my own shroud, but couldn't get it to apply to MAPP. I recollect that I punted on this and didn't change the type, even though it would have been a cleaner design. Better shroud documentation would have helped.
   (b) I recollect having repeated problems around the `*.d` dependency files. If I changed a header, the build system wouldn't auto-update the `*.d` files. I found myself frequently deleting all `*.d` files in the build area, which made for larger than necessary rebuilds.
   (c) After several iterations of changes and rebuilds, I somehow corrupted some of the configuration behind the bootstrap file – I kept getting warnings about it. The bootstrap system was complex, and I never figured out how to fix it. But I still managed to keep it building.

Dev 07 Developer 7 is a code physicist who was working with our code to add support for a physics package.

1. Yes and no. I was able to clone/build/run/test in the default manner relatively easy, but I needed to contact the developers directly to figure out how to do a build with a custom library.
2. Yes, after some help from L.B. and R.R..
3. Sort of? At the time, there was a test problem that I was trying and it had some out of date settings. R.R. helped get things in order.
4. Two (I think).
5. I had some recommendations back when I last touched the code, but you may have gotten to these already:
   (a) Make sure all the test problems in the repository work without modifications
   (b) One or more detailed examples in which you describe how to visualize the result of the test problem

Dev 08 Developer 8 is an applied mathematician who worked on the code in 2019–2020 in support of GPU acceleration of our Lagrangian Hydrodynamics kernels.

1. The website has been definitely useful to get started and I was able to try/modify and test the modifications with the documentation.
2. Absolutely: it is straight forward and flexible enough to pick just was is needed to run, which really facilitates the process.
3. All the commands are given, which again allow it to be straight forward.
4. One member (at the beginning) was usually enough to make sure that the integration was working.
5. One table for the different machines with the default command/compiler/restriction could be useful. For example some target (asan) might need a specific compiler on some machines.

Dev 09 Developer 9 is an applied mathematician who started working on the code in 2020 in support of the Teton package. As of November 2020, this development is still "In Progress"

1. Yes.
2. An enthusiastic yes: the ability to build with specified branches from different libraries was exceedingly helpful.
3. Yes.
4. Typically only two (Initially L.G. the most and then A.S.).
5. n/a.

Dev 10 Developer 10 is an applied mathematician who worked on the code in 2020 in support of the Tribol interface physics library.

1. More often than not, I found the MAPP documentation helpful for whatever task I wanted to accomplish. Getting up and running with the code was straightforward following the guide and there was sufficient detail to customize my build to my liking.
2. I haven't done this too much, but I was able to get the latest version of Tribol up and running with your help.

3. The documentation was clear on how to build MAPP in debug mode, but there wasn't much guidance after that. It would be nice to have additional documentation on how to effectively debug in MAPP. For instance, if I'm just writing code for a single package, I might only want to build the debuggable version of that package so the rest of the code is optimized. It's not clear how to do this using the build system from the documentation. Also, I've run into issues with running make after modifying the code. There seems to be some issue with BLT when I do this, so I end up rebuilding entire packages unnecessarily.

4. Just you.

5. As I mentioned above in 3, maybe a guide for debugging code in MAPP?

Dev 11 Developer 11 is a computer scientist who worked on the code in Spring 2020 to add support for an LLNL research debug tool, fppchecker. This feature was being considered to help debug a floating point error that we were seeing when running the code on device (GPU), but not on the host (CPU). fppchecker made some assumptions about how the code should be built that were incompatible with those of the MARBL Build System, which will require some additional development within MBS before this integration can be successful. We were able to resolve our floating point bugs before being able to resolve these issues and deferred the integration of fppchecker until those changes could be made to MBS.

1. Yes.

2. Not quite; building the code with a not-before-tried compiler was not straightforward, and we are yet to resolve the compilation errors.

3. Didn't get to it, but probably yes.

4. Two (and they did most of the work as I was mostly stuck).

5. Recommendations:

   (a) Code

   i. Minimize compiler-specific `ifdef` guards in the code (e.g., `CUDA_ARCH`, `NVCC`); RAJA policies (or OpenMP) should be used instead.

   ii. Current 'runtime' capability to choose to run on the CPU vs. GPU via `IFDEF`s should be replaced by the RAJA multi-policy where possible.

   iii. Clean up `__host__`, `__host_device__`, and `__device__` annotations. NVCC is the least restrictive compiler about this; Clang prohibits mismatches/overloading of `__host_device__` functions by host or device functions with the same signature. Cray compilers are Clang-based so this will be an immediate issue when starting to port the code to El Capitan.

   iv. For portability, `__host__`, `__host_device__`, and `__device__` annotations should be replaced by macros that can then be defined in a single place (or compiled out entirely).

   (b) Compiler instructions

   i. Add instructions on how to disable linking, and how to do compilation and linking separately

   ii. While it was clear how to set the host compiler, instructions on setting the device compiler are missing

Dev 12 Developer 12 is a computer scientist who worked on the code in Summer 2020 in support of the Scalable Checkpoint and Restart (SCR) package. This effort is still "In Progress" as of November 2020.

1. The docs did a fine job to get me going. There were two items I remember working through. To understand a new code, I like to step through a simple test case under control of a debugger. For that, I like to build and run in full debug mode. Figuring how to get a debug build took some back-and-forth discussion. I ran into a problem that seemed to show up while building

in debug but not while building the optimized code. I also needed to switch compilers at some point, and I had to ask for help with that. For one, the intel compiler seemed to trip on some Miranda Fortran syntax that gcc did not, so there was a bit of hacking/debugging that code to get past it.

2. I was able to integrate calls to SCR into two components. The tricky part then was finding the commands I needed to fetch and build using those modified branches. As for adding SCR itself, I skipped that step by installing a copy external to the project, so that it's not fully integrated yet.
3. I wanted to explore the range of checkpoint/restart options in a test code to make sure I had SCR integration coverage as best as I could. The docs described some of this, but it was still helpful to get in touch with people to ask questions.
4. I ended up working with 3-4 people.
5. Building the full suite of packages can take a long time. Knowing the proper commands to just rebuild parts that have changed would help, but it took a while before I figured that out. I eventually got help from a developer via email and a PR while documenting my steps.

Dev 13 Developer 13 is an applied mathematician who has been working with the code for the past few years in support of adding Adaptive Mesh Refinement (AMR) capabilities to MARBL's Lagrangian hydrodynamics package. This is still a research effort within the MFEM library and Developer 13 has been adding incremental support for AMR, as new features become available through MFEM.

1. Yes. The documentation is very good for a simulation code under heavy development.
2. Yes.
3. Yes.
4. There are 3 team members I consult with frequently, probably have interacted with 5 or 6 more infrequently.
5. I think it might help decrease the fragility of the regression testing to improve the toolset that extracts and compares data between runs. T.S. started helping me with this a couple years ago, I should pick this thread back up and continue to work on it.