Structured Adaptive Mesh Refinement Adaptations to Retain Performance Portability with Increasing Heterogeneity

A. Dubey
Mathematics and Computer Science Division
Argonne National Laboratory, USA

M. Berzins
School of Computing, University of Utah, USA

C. Burstedde
Institut für Numerische Simulation, University of Bonn, Germany

M.L. Norman
San Diego Supercomputer Center
Department of Physics, University of California, San Diego, USA

D. Unat
Koç University, İstanbul, Turkey

M. Wahib
National Institute of Advanced Industrial Science and Technology AIST/TokyoTech
RIKEN Center for Computational Science,Japan

***Abstract*—Adaptive mesh refinement (AMR) is an important method that enables many mesh-based applications to run at effectively higher resolution within limited computing resources by allowing high resolution only where really needed. This advantage comes at a cost, however: greater complexity in the mesh management machinery and challenges with load distribution. With the current trend of increasing heterogeneity in hardware architecture, AMR presents an orthogonal axis of complexity. The usual techniques, such as asynchronous communication and hierarchy management for parallelism and memory that are necessary to obtain reasonable performance are very challenging to reason about with AMR. Different groups working with AMR are bringing different approaches to this challenge. Here, we examine the design choices of several AMR codes and also the degree to which demands placed on them by their users influence these choices.**

Introduction

In many science research domains mesh-based methods for solving partial differential equations (PDEs) play a crucial role in scientific discovery. In many of these applications a wide range of scales needs to be resolved, which can make simulations intractable without adaptivity in mesh resolution even on the largest available supercomputers. Adaptive mesh refinement (AMR) [2] is an important method that enables mesh-based applications to run effectively at higher resolution within available computing resources by allowing high resolution only where needed. This advantage comes with greater complexity in the mesh management machinery, however, which includes resolution of quantities at fine-coarse interfaces and load distribution that changes dynamically in parallel environments.

Many of the AMR packages and codes grew when the distributed-memory parallel model was dominant. The decomposition and distribution of work in this parallelization model can be very coarse grained, and this is reflected in the design of almost all early AMR libraries and AMR-based codes. The intermediate years of multicore platforms motivated the use of threading provided by OpenMP, which was not very invasive and, therefore, easy to adopt. The arrival of heterogeneity in the form of accelerators of different types on compute nodes of supercomputing platforms poses a challenge that has required varying degrees of intrusive changes in AMR libraries and applications codes. Because different codes are tackling this challenge in different ways, it is interesting and useful to understand the rationale behind the choices made by their developers.

An opportunity to do so occurred in the form of a minisymposium at "The Platform for Advanced Scientific Computing" conference in Zurich, Switzerland. The theme of the minisymposium was techniques and tools being developed and utilized by various AMR packages and AMR-based codes for heterogeneous platforms. It became obvious that there is considerable diversity in perceived challenges and in approaches in response to those challenges. The causation and correlations were intriguing enough that several of the participants proceeded to do a deeper analysis of their respective motivations and priorities that are presented in this article.

Adaptive Mesh Refinement

AMR is a method for reducing both the memory and compute footprint of partial differential equation (PDE) solvers. The method has been around for 40 years [2]. The commonality among the AMR codes and packages featured here is that they have logically Cartesian mesh cells on which the solution evolves and that the cells at the same level of refinement have identical spatial resolution. Within that commonality there are two distict flavors of refinement handling and AMR bookkeeping:

1) Cells are collected in equally-sized blocks in terms of number of cells per block. The blocks are organized in one or more octrees where the coarsest level forms the root of the trees. Usually an explicit parent-child relationship exists between blocks at two consecutive levels. We refer to this type as *octree AMR*.
2) Cells are collected in arbitrarily-sized blocks called patches. These blocks can be placed anywhere on the physical domain as long as finer blocks are fully contained within a region of the next coarse level. Unlike octrees, no explicit parent-child relationship exists among blocks. This flavor is referred to as *block-structured AMR* hereafter.

Of the codes discussed in this paper, Flash-X is a new incarnation of FLASH for exascale. FLASH was developed on top of Paramesh [9], an octree AMR library. In addition to Paramesh, Flash-X also supports the use AMReX [1], which is natively block-structured but mimics the behavior of octrees for Flash-X. Uintah implements block-structured AMR. Enzo transitioned from block-structured to octrees in its newest version, Enzo-E, using Cello [4] as its underlying AMR. `p4est` is an AMR software library managing a forest of octrees.

Design Choices

Flash-X

The Flash-X approach to design can be summarized as the distribution of compositional and performance handling between source and configuration tools so that none is too complex or difficult to maintain. The most influential design choice of composability in the earliest version of FLASH was driven by the need to handle a variety of physical situations that required inclusion of components in different permutations and combinations along with different sets of physical state variables. This was achieved by creating a domain-specific language (DSL) that encodes meta-information about various components and subcomponents of the code in "config" files. A "setup tool" parses the config files recursively to generate a self-consistent collection of components [7]. This fundamental design choice has remained the linchpin of all architectural enhancements including the most recent one.

True to the philosophy of distributed complexity handling, different abstractions and tools are being used to address different challenges posed by heterogeneity. For hierarchical domain decomposition, Flash-X uses AMReX's tiling as the base abstraction instead of blocks. Elimination of bulk synchronization is planned through a combination of using asynchronous collectives from the host overlapping with other local operations on the host or accelerators and combined with a domain-specific runtime system that manages all data movements.

Two modes of code transformation are used to handle platform-specific heterogeneity: one for physics operators, and another one for timestepping. Physics operators take a key-value dictionary approach where keys can have multiple alternative values, each one specific to a type of accelerator. The numerics of the code are decorated with keys to enable a single expression of the computation. The timestepping code transformation tool generates the code from a library of platform specific templates.

Uintah

Uintah opted for an asynchronous many-task (AMT) dataflow approach from the outset [3] that consists of tasks and a runtime system that uses a dynamic directed acyclic graph to guide task execution. This AMT runtime extracts the appropriate level of parallelism by automatically mapping tasks to available computational resources. The primary features of the design include: (1) A shared memory compute-node data warehouse that uses atomic operations to be lock-free. (2) Decentralized execution of the task graph, which is implemented by each CPU core or GPU requesting work itself. (3) Accelerator task execution on a node, which is implemented through an extension of the runtime system that uses data prefetching for efficient task execution. (4) Extensive scalability through exploiting asynchronous (including out-of-order) task execution, over-decomposition of tasks, overlapping of communication and computation, work stealing, and task graph prioritization, based on communication needs and dependencies.

An important aspect of the Uintah design is the choice of scheduler. Uintah uses two main schedulers. The Uintah MPI scheduler executes a fixed task graph on a core but with asynchronous communication. In contrast, the unified scheduler implements a completely asynchronous approach across all available cores or accelerators. In order to enable Uintah tasks to run without code changes across multiple types of CPUs and GPUs, the Kokkos approach was adopted over other portability approaches after early experiments and design studies. At scale, the use of MPI+Kokkos has allowed for good strong-scaling to 442,368 threads across 1,728 Knights Landing processors.

Enzo-E

Enzo-E [5] is a new, extreme-scale version of Enzo that addresses the extensibility and scalability limitations of Enzo through a completely new object-oriented software design and implementation and a new, more scalable AMR infrastructure layer called *Cello* inspired by `p4est`. Enzo-E uses Charm++ for parallelization rather than MPI. This provides an abstract interface to the parallel machine, asynchronous task execution, dynamic load balancing, and fault tolerance, among other benefits. The entire code is implemented in C++ except for certain Fortran physics kernels taken from the Enzo codebase.

Enzo, like other block-structured AMR, replicates metadata on every node. While that makes communication primitives easier to implement, the amount of metadata grows with the size of the problem and runs into a scaling limit eventually. The most significant design choice for Enzo-E was to abandon block-structured AMR in favor of a forest-of-octrees-style AMR because of its superior parallel scalability and simplicity [6]. The entire multilevel mesh is fully distributed across the parallel machine by using Charm++'s *chare-array* data structure, and each block in the array is assigned to a Charm++ task called a *chare*.

The second major design choice was to implement the code in C++ following object-oriented design principles. This leads to a clean separation of concerns between numerical methods in the Enzo-E layer, AMR mesh data structures and operations in the Cello layer, and task mapping and parallel execution in the Charm++ layer. The Enzo-E application layer consists of a collection of method objects that initialize and update field and particle data stored in the Cello blocks. Currently, Enzo-E runs only on CPU clusters with excellent weak and strong parallel scalability. Since the code is written mainly in C++, however, it can in principle use performance portability tools such as Kokkos to translate specific numerical method kernels to be executed on GPUs.

`p4est`

`p4est` is a software library that manages adaptive forest-of-octree meshes in the distributed memory parallel model. From the beginnings, the algorithms and data structures of `p4est` were designed with a strong focus on modularity and flexibility. To this day, these original structures have supported all newer algorithms and improvements. Extensions have been introduced to support new algorithmic features in a backwards-compatible way [8]. The observed longevity can be attributed to the logic minimalism of both local and global state.

`p4est` works with two first-class objects, the *connectivity* of tree roots and the *forest* storing the process-local leaves (and only them). The former is a read-only global replicated object (one per MPI shared-memory node is sufficient) that lists the number of trees and for each the face, edge, and corner neighbors and their relative orientation. One rule of thumb is to use the connectivity to represent the domain topology and to leave geometric representation and accuracy to the adaptive refinement.

The forest object stores only the leaves of the forest. The leaves satisfy a total order and are allocated strictly process-local: each leaf has one and only one owner process. Mesh refinement, coarsening, and 2:1 balancing have strictly process-local effects (even though the latter depends on one query-reply round of communication). Since these generally offset load balance, the leaves may be repartitioned at any time to re-establish a guaranteed ± 1 leaf distribution between processes. This is possible only by routinely allowing for partition boundaries inside trees and for tree boundaries inside a partition, and for empty processes as well.

The encoding of the parallel mesh partition requires some amount of global replicated metadata, which does not include per-leaf refinement or geometrical and numerical data. Similarly, when the metadata depends on the total number of MPI processes, the amount per rank is kept low. `p4est`, in effect, replicates 32 bytes per MPI rank on each rank. This data is sufficient to encode the shape of all partition boundaries without referencing a single leaf. This is a powerful property of pure octree approaches and allows for executing fast remote, general multi-objective searches. This memory is currently redundant on multi-rank nodes, but prototype code exists to reduce it further using MPI-3 shared memory. `p4est` currently is MPI-only, with scalability tested to 10^6+ parallel processes. Plans exist for using OpenMP for additional threading, multiplying the speed of internal algorithms where applicable. However, the implementation assures that threading is transparent and will not be exposed to the public API (beyond the passing of hints).

Analysis and Inferences

The codes cover a spectrum from relatively little change (`p4est`), adoption of technologies in place (Uintah), complete rewrite from scratch (Enzo-E), and ambitious architectural refactoring (Flash-X), in increasing order of complexity.

Uintah's asynchronous task-based design from the outset has proven to be prescient for hyperparallelism. Although its vintage is roughly the same as that of FLASH, it started as a new from-scratch code and opted for C++ as the language in which to code. The design included a separation of concerns so that communication and computation could evolve out of lockstep. Since a great deal of effort has been invested in C++ template-based tools for performance portability, Uintah has been in a strong position to leverage those tools.

The developers of Enzo-E took a different tack. They evaluated the limitations of their existing AMR, considered the forthcoming demands from science, and concluded that block-structured AMR is fundamentally flawed for scalability that will be demanded by their future science and machine directions. Instead of patching over their existing framework, they opted to leverage known scalable technologies, namely, Charm++ with built-in asynchrony and scalability, and built their capabilities on top of it. The key to success here was to have a long lead time for infrastructure development so that the project could begin on a small scale with modest resources. Since Enzo had been a community-developed and community-supported code for a while, such considerations for resource management and a long-lead transition time were critical in driving the directions of development.

Flash-X had the most challenging transition to make. Even when it first came into existence, FLASH did not start from scratch. It was an amalgamation of legacy codes, and its architecture was gradually imposed over it through an iterative process. That also set the stage for FLASH to remain a Fortran code. Over the past few years its growth has exploded in terms of physics solver support, but investment in infrastructure has suffered. It supports multiple science domains that need an operational code. The code clearly needed a fundamental invasive re-architecting, but there also had to be a gradual transition path for ongoing verification during transition. This is the reason that it had to find a way to take the separation of concerns several layers deep into the architecture and find a collection of tools to address different aspects of its computational challenges.

In contrast to Uintah, Enzo, and Flash-X, `p4est` is agnostic of the numerics of the solvers, and it is not authoritative to the orchestration of computation. While `p4est` does not directly address heavy-duty computation that might be offloaded to an accelerator and other special-purpose devices, it offers all the logic necessary to support applications in device-oriented data partitioning and assignment.

Conclusions

The codes included in this study represent a broad coverage of the spectrum of AMR technologies and their use in science domains. Study of their diverse approaches to tackling performance portability in the presence of heterogeneity proved to be an interesting exercise. The approaches ranged from mostly nonintrusive (in `p4est`) to moderate (in Uintah) to extremely intrusive (in Flash-X), to a complete rewrite of the infrastructure (in Enzo-E). Each code has its own targets and accompanying biases. One feature that emerges as a common theme is that sufficient attention must be paid to the basic design of the code structure with an eye to flexibility.

Acknowledgements

■ REFERENCES

1. Amrex. https://amrex-codes.github.io/, 2020.
2. M. Berger and J. Oliger. Adaptive mesh refinement for hyperbolic partial differential equations. *Journal of Computational Physics*, 53:484–512, 1984.
3. M. Berzins, J. Beckvermit, T. Harman, A. Bezdjian, A. Humphrey, Q. Meng, J. Schmidt, , and C. Wight. Extending the uintah framework through the petascale modeling of detonation in arrays of high explosive devices. *SIAM Journal on Scientific Computing*, 2016.
4. J. Bordner. Cello. https://cello-project.org/, 2020.
5. J. Bordner and M. L. Norman. Computational Cosmology and Astrophysics on Adaptive Meshes using Charm++. *arXiv e-prints*, page arXiv:1810.01319, Oct. 2018.
6. C. Burstedde, L. C. Wilcox, and O. Ghattas. `p4est`: Scalable algorithms for parallel adaptive mesh refinement on forests of octrees. *SIAM Journal on Scientific Computing*, 33(3):1103–1133, 2011.
7. A. Dubey, K. Antypas, M. Ganapathy, L. Reid, K. Riley, D. Sheeler, A. Siegel, and K. Weide. Extensible component based architecture for FLASH, a massively parallel, multiphysics simulation code. *Parallel Computing*, 35:512–522, 2009.
8. T. Isaac, C. Burstedde, L. C. Wilcox, and O. Ghattas. Recursive algorithms for distributed forests of octrees. *SIAM Journal on Scientific Computing*, 37(5):C497–C531, 2015.
9. P. MacNeice, K. Olson, C. Mobarry, R. de Fainchtein, and C. Packer. PARAMESH: A parallel adaptive mesh refinement community toolkit. *Computer Physics Communications*, 126(3):330–354, 2000.

Anshu Dubey is a Computational Scientist in the Mathematics and Computer Science Division at Argonne National Laboratory. She is the software architect for Flash-X. Contact her at adubeyanl.gov

Martin Berzins is a multi-disciplinary Computational Science researcher whose research cuts across Applied Mathematics, Computer Science and Engineering. He is a Professor of Computer Science in the School of Computing and in the Scientific Computing Imaging Institute at the University of Utah and a Visiting Professor at the University of Leeds.

Carsten Burstedde is a physicist with a doctorate degree in applied mathematics from Bonn University, Germany. Together with Lucas C. Wilcox, he founded the `p4est` software in 2007 during his postdoc at the University of Texas at Austin. He is now a Professor for Scientific Computing at the Institute for Numerical Simulation in Bonn.

Michael L. Norman is Director of the San Diego Supercomputer Center and Distinguished Professor of Physics at the University of California, San Diego. There, he also directs the Laboratory for Computational Astrophysics which develops community application software for astrophysical simulation, including the ZEUS and Enzo codes.

Didem Unat is a Professor at Ko University, Istanbul, Turkey. She is the recipient of the Marie Sklodowska-Curie Individual Fellowship from the European Commission in 2015 and the Young Scientists Award in 2019 from the Science Academy of Turkey.

Mohammed Wahib is a senior scientist at AIST/TokyoTech Open Innovation Laboratory, Tokyo, Japan. Prior to that he worked as a researcher in RIKEN Center for Computational Science (RIKEN-CCS).