T8CODE V. 1.0 - MODULAR ADAPTIVE MESH REFINEMENT IN THE EXASCALE ERA

Johannes Holke* Carsten Burstedde† David Knapp* Lukas Dreyer*
Sandro Elsweijer* Veli Ünlü* Johannes Markert* Ioannis Lilikakis* Niklas Böing*
Prasanna Ponnusamy* Achim Basermann*

January 2023

ABSTRACT

In this note we present version 1.0 of our software library `t8code` for scalable dynamic adaptive mesh refinement (AMR) officially released in 2022 [1]. `t8code` is written in C/C++, open source, and readily available at www.dlr-amr.github.io/t8code. The library provides fast and memory efficient parallel algorithms for dynamic AMR to handle tasks such as mesh adaptation, load-balancing, ghost computation, feature search and more. `t8code` can manage meshes with over one *trillion* mesh elements [2] and scales up to one *million* parallel processes [3]. It is intended to be used as mesh management back end in scientific and engineering simulation codes paving the way towards high-performance applications of the upcoming exascale era.

Keywords: mesh management, adaptive mesh refinement, numerical simulation, high performance computing, exascale

1. INTRODUCTION

AMR has been established as a successful approach for scientific and engineering simulations over the past decades [4–7]. By modifying the mesh resolution locally according to problem specific indicators, the computational power is efficiently concentrated where needed and the overall memory usage is reduced by orders of magnitude. However, managing adaptive meshes and associated data is a very challenging task, especially for parallel codes. Implementing fast and scalable AMR routines generally leads to a large development overhead motivating the need for external mesh management libraries like `t8code`.

`t8code` is written in C/C++, open source, and version 1.0 can be obtained at www.dlr-amr.github.io/t8code [1]. It uses efficient space-filling curves (SFC) to manage the data in structured refinement trees. While in the past being successfully applied to quadrilateral and hexahedral meshes [8, 9], `t8code` extends these SFC techniques in a modular fashion, such that arbitrary element shapes are supported. We achieve this modularity through a novel decoupling approach that separates high-level (mesh global) algorithms from low-level (element local) implementations. All high-level algorithms can then be applied to different implementations of element shapes and refinement patterns. A mix of different element shapes in the same mesh is also supported.

In version 1.0, `t8code` provides implementations of Morton type SFCs with $1 : 2^d$ refinement for vertices ($d = 0$), lines ($d = 1$), quadrilaterals, triangles ($d = 2$), hexahedra, tetrahedra, prisms, and pyramids ($d = 3$). The latter having a $1 : 10$ refinement rule with tetrahedra emerging as child elements [10]. Additionally, implementation of other refinement patterns and SFCs is possible according to the specific requirements of the application.

The purpose of this note is to provide a brief overview and a first point of entrance for software developers working on codes storing data on (distributed) meshes. The structure is as follows: Sec. 2 gives a brief outline of the fundamental algorithms, Sec. 3 presents the interface, Sec. 4 emphasizes the modularity of `t8code` while Sec. 5 shows some performance

*German Aerospace Center (DLR), Institute for Software Technology, Cologne, Germany, johannes.holke@dlr.de, corresponding author
†University of Bonn

results. Finally, in Sec. 6 we draw a conclusion and give a brief outlook.

For further information beyond this short note and also for code examples, we refer to our Documentation and Wiki [1] and our other technical papers on `t8code` [2, 3, 10–16]

2. FUNDAMENTAL CONCEPTS

`t8code` is based on the concept of tree-based adaptive mesh refinement. Starting point is an unstructured input mesh, which we call coarse mesh that describes the geometry of the computational domain. The coarse mesh elements are refined recursively in a structured pattern, resulting in refinement trees of which we store only minimal information of the finest elements (the leafs of the tree). We call this resulting fine mesh the `forest`.

By enumerating the children in the refinement pattern we obtain a space-filling curve logic. Via these SFCs, all elements in a refinement tree are assigned an index and are stored in linear order of these indices. Information such as coordinates or element neighbors do not need to be stored explicitly, but can be recovered from the index and the appropriate information of the coarse elements. The less elements the input mesh has, the more memory and runtime are saved through the SFC logic. `t8code` supports distributed coarse meshes of arbitrary size and complexity, which we tested for up to 370 million input elements [11].

The `forest` mesh is distributed, that is, at any time, each parallel process only stores a unique portion of the `forest` mesh, the boundaries of which are calculated from the SFC indices; see Fig. 1.

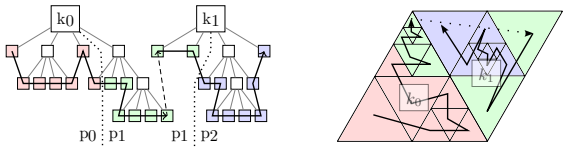

Figure 1: Left: Quad-tree of an exemplary `forest` mesh consisting of two trees (k_0, k_1) distributed over three parallel processes P0 to P2. The SFC is represented by a black curve tracing only the finest elements (leaf nodes) of each tree. Right: Sketch of the associated triangular mesh refined up to level three.

3. INTERFACING WITH `t8code`

In this section we discuss the main interface of `t8code` and how an application would use it. While `t8code` offers various ways to interact with meshes and data, we restrict ourselves to the most important functionality here.

Every application is different and comes with their own requirements, data, and adaptation criteria. In order to support a wide variety of use cases, our core philosophy for `t8code` is to impose as few assumptions and to offer as much freedom as possible. We cater for this by applying the Hollywood principle: "Don't call us, we'll call you!". Whenever an application needs to interact with the mesh, e.g., adapting the mesh, interpolating data, etc., we offer suitable callback handlers.

The application developer implements custom callback functions and registers them via the `t8code` application programming interface (API). Any mesh specific details on how to access individual elements in the forest is opaque to the application and internally handled by `t8code` in an efficient manner. Of course, any typical application using hierarchical meshes needs to store data on the elements of a forest. This data might correspond to some simulated state variables, e.g., fluid velocity and temperature in a CFD simulation. In accordance to our core philosophy, the data is only loosely coupled with `t8code`'s data structures. In order to properly access the application data in the callbacks, the data simply needs to be provided as a consecutive array with one entry per element enumerated in SFC order. For parallel applications, access to neighboring elements across parallel zones (ghost layer) is provided in a similar fashion.

3.1 An example application

In the following section, we want to discuss the most important high-level operations implemented in `t8code`. For this, consider a 3D numerical solver application that traces a flow bubble moving around a rotating cylinder. The application runs in parallel and the mesh is dynamically adapted in (almost) every time step resolving the moving bubble with higher resolution than the surrounding domain. These perpetual mesh changes constantly require the flow state data to be interpolated from one adaption step to the next. A visualization of such a setup might look like Fig. 2.

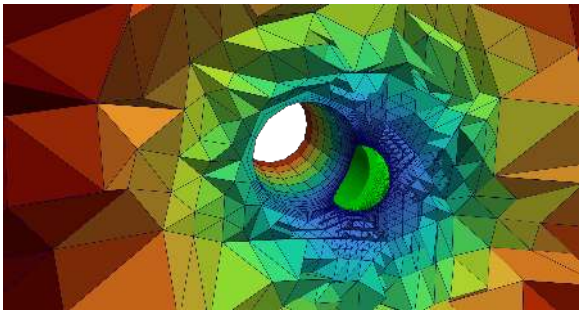

Figure 2: Meshed region of fluid flow around a rotating cylinder. The green blob corresponds to a bubble that is transported within the moving fluid. The mesh is particularly refined along the boundary of the bubble. Colors encode the element's distance from the bubble.

The standard way to implement such an application is to use the following high-level `t8code` operations: ***New, Adapt,***

Balance, Interpolate, Partition, Ghost, Iterate. This is also illustrated in the flowchart in Fig. 3. Next, we give more details about the different operations:

New: Constructs a new, uniformly refined mesh from a coarse geometry mesh. This mesh is already distributed across the parallel processes. This step is usually only carried out once during the preprocessing phase.

Adapt: Decides for each element whether to refine, coarsen, or pass according to the results of a criterion provided by a custom adaption callback.

Balance: Establishes a 2:1 balance condition, meaning that afterwards the refinement levels of neighboring elements are either the same or differ by at most ± 1. Note, this operation only refines elements, never coarsens them. Applications are free to decide whether they require the balance condition or not.

Interpolate: Interpolates data from one forest mesh to another. For each element that was refined, coarsened or remained the same, an application provided callback is executed deciding how to map the data onto the new mesh.

Partition: Re-partitions the mesh across all parallel processes, such that each process has the same computational load (e.g. element count). Due to the SFC logic, this operation is very efficient and may be carried out in each time step.

Partition Data redistributes any user defined data from the original mesh to the re-partitioned one. Input is an array with one entry for each element of the original forest containing the application data, output is an array with one entry for each element of the re-partitioned forest, containing the same data (that may previously have been on a different process).

Ghost: Computes a list of all ghost elements of the current process. Ghosts are elements that are neighbors to elements of the process, but do not belong to the process itself.

Ghost exchange transfers application specific data across all ghost elements. Input is an array of application data with one filled entry for each local element and one unfilled entry for each ghost. On output the entries at the ghost elements will be filled with the corresponding values from the neighbor processes.

Iterate: Iterates through the mesh, providing face neighbor information for each element passed as an argument to the callback. In our example application, it is used to carry out the advection step of the bubble.

Search may be used additionally for extra tasks, such as searching for particles, or identifying flow features. It hierarchically iterates through the mesh and executes a callback function on all elements that match a given criterion. Leveraging the SFC tree logic, `Search` omits large chunks of the mesh if they do not match the criterion. Hence, it does not necessarily inspect each individual element and therefore performs much faster than a linear search [2, 17].

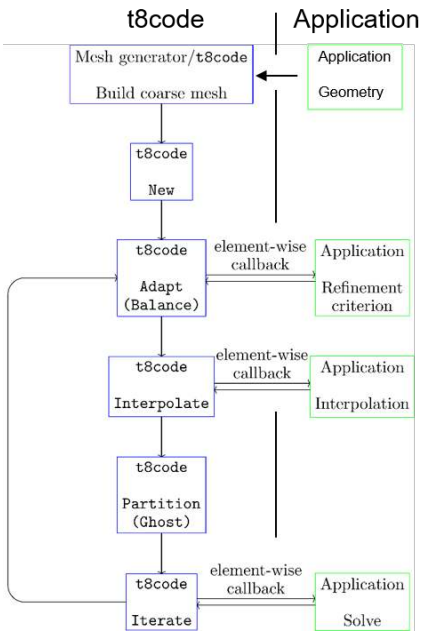

Figure 3: Flowchart of a typical simulation code which interacts with `t8code`. Information about the different operations can be found in the text.

4. MODULARITY & EXTENSIBILITY

A distinct feature of `t8code` compared to similar AMR libraries is its high modularity achieved by decoupling high-level from low-level algorithms and coming along with it the support for arbitrary element shapes and refinement patterns. It also allows to combine different element shapes within the same mesh (hybrid meshes).

All high-level operations use the low-level algorithms only as a black box. For example, ***Adapt*** iterates through the mesh and when necessary calls the low-level algorithms `element_children` or `element_parent` to refine or coarsen an element. In order to implement the logic of ***Adapt***, however, no knowledge of the implementation details of these low-level functions is required.

Thus, for each individual tree we can simply replace the underlying implementation of the low-level algorithms (e.g. from tetrahedra to hexahedra) without affecting the high-level functionality. We achieve this by encapsulating all shape-specific element operations such as parent/child computation, face-neighbor computation, SFC index computation and more in an abstract `C++` base class. The different element shapes and refinement patterns are then specializations of this base class. Hence, `t8code` can be easily extended - also by application developers - to support other refinement patterns and SFCs.

Moreover, this very high degree of modularity allows us to support an even wider range of non-standard additions. For example, the insertion of sub-elements[1] to resolve hanging

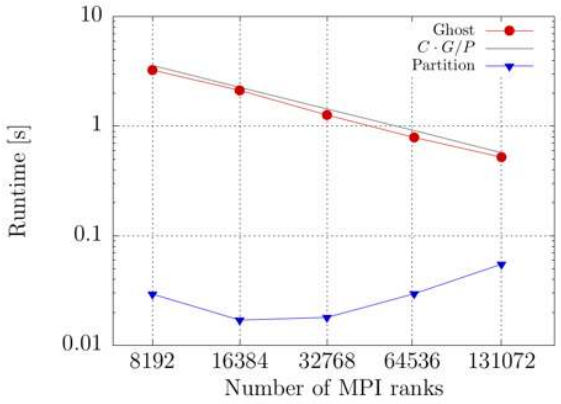

Figure 4: Strong scaling on JUWELS with tetrahedral elements. We plot the runtimes of ***Ghost*** and ***Partition*** with a refinement band from levels 8 to 10 after four time steps. Hence, the forest mesh consists of approximately 1.91 billion tetrahedra. As observed in the plot, we achieve perfect scaling for ***Ghost*** in the number G/P of ghosts per process. The runtime of Partition is below 0.1 seconds even for the largest run. More details can also be found in [2].

nodes [13] in quadrilateral meshes. Each quad element that has a hanging node is subdivided into a set of several triangles eliminating the hanging node.

Furthermore, we added support for holes[2]in the mesh by selectively deleting elements [16]. This feature can be used to incorporate additional geometry information into the mesh. Similar to marking elements as getting refined or coarsened, we can additionally mark elements as getting removed. These elements will be eliminated completely from the SFC reducing the overall memory footprint.

Addionally, we support curved hexahedra with geometry-informed AMR [14]. Thus, information such as element volumes, face areas, or positions of interpolation/quadrature points in high order meshes can be calculated exactly with respect to the actual geometry. Another use case is to start with a very coarse input mesh and geometrically refine the mesh maxing out the performance benefits of tree-based AMR.

5. PERFORMANCE

In this section we present some of our benchmark results from various performance studies conducted on the JUQUEEN [18] and the JUWELS [19] supercomputers at the Jülich Supercomputing Center.
Ghost and ***Partition*** are exceptionally fast with proper scaling of up to 1.1 *trillion* mesh elements; see Tab. 1, [2]. In Fig.4 we show a strong scaling result for a tetrahedral mesh achieving ideal strong scaling efficiency for ***Ghost***. Furthermore, in a

[2]Subelements and holes are not part of version 1.0 but will be integrated soon.

# process	# elements	# elem. / process	Ghost	Partition
49,152	1,099,511,627,776	22,369,621	2.08 s	0.73 s
98,304	1,099,511,627,776	11,184,811	1.43 s	0.33 s

Table 1: Runtimes on JUQUEEN for the ghost layer and partitioning operations for a distributed mesh consisting of 1.1 trillion elements.

prototype code [15] implementing a high-order discontinuous Galerkin method (DG) for advection-diffusion equations on dynamically adaptive hexahedral meshes we obverve a 12 times speed-up compared to non-AMR meshes with only an overall 10 to 15% runtime contribution of `t8code`; see Fig. 5.

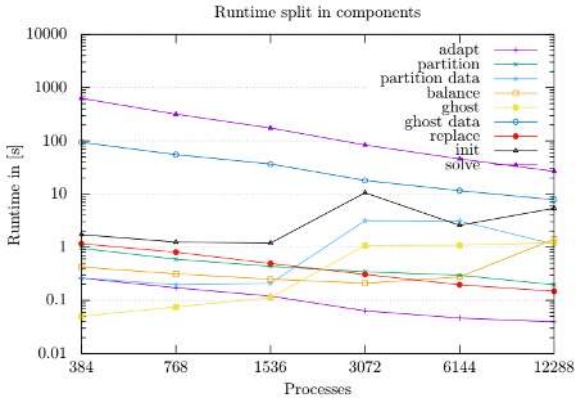

Figure 5: Runtimes on JUQUEEN of the different components of our DG prototype code coupled with `t8code`. Note that all features associated with dynamical mesh adaptation utilize only around 15% of the total runtime largely independent of the number of processes.

6. CONCLUSION

In this note we introduce our open source AMR library `t8code`. We give a brief overview of the fundamental algorithms and data structures, namely our modular SFC, and outline a general usage pattern when an application interacts with the library. Due to its high modularity, `t8code` can be easily extended for a wide range of use cases. Performance results confirm that `t8code` is a solid choice for mesh management in high-performance applications in the upcoming exascale era.

Future efforts will include an integration of our techniques into simulation use cases with in-depth performance and accuracy evaluations. Additionally, we strive to extend all presented features to all element shapes and space dimensions. Other possible extensions that we plan to research in the near future are mesh adaption of prism boundary layers and the support for an-isotropic refinement.

References

[1] Holke J., Burstedde C., Knapp D., Dreyer L., Elsweijer S., Uenlue V., Markert J., Lilikakis I., Boeing N. "t8code.", 9 2022. URL https://github.com/dlr-amr/t8code

[2] Holke J., Knapp D., Burstedde C. "An Optimized, Parallel Computation of the Ghost Layer for Adaptive Hybrid Forest Meshes." *SIAM Journal on Scientific Computing*, pp. C359–C385, Jan. 2021. URL https://epubs.siam.org/doi/abs/10.1137/20M1383033

[3] Holke J. *Scalable algorithms for parallel tree-based adaptive mesh refinement with general element types*. PhD thesis, Rheinische Friedrich-Wilhelms-Universität Bonn, 2018

[4] Teunissen J., Keppens R. "A geometric multigrid library for quadtree/octree AMR grids coupled to MPI-AMRVAC." *Computer Physics Communications*, vol. 245, 106866, 2019. URL https://www.sciencedirect.com/science/article/pii/S001046551930253X

[5] Bangerth W., Hartmann R., Kanschat G. "Deal.II—A General-Purpose Object-Oriented Finite Element Library." *ACM Trans. Math. Softw.*, vol. 33, no. 4, 24–es, aug 2007. URL https://doi.org/10.1145/1268776.1268779

[6] Dörfler W. "A Convergent Adaptive Algorithm for Poisson's Equation." *SIAM Journal on Numerical Analysis*, vol. 33, no. 3, 1106–1124, 1996. URL https://doi.org/10.1137/0733054

[7] Babuvška I., Rheinboldt W.C. "Error Estimates for Adaptive Finite Element Computations." *SIAM Journal on Numerical Analysis*, vol. 15, no. 4, 736–754, 1978. URL https://doi.org/10.1137/0715049

[8] Burstedde C., Wilcox L.C., Ghattas O. "p4est: Scalable Algorithms for Parallel Adaptive Mesh Refinement on Forests of Octrees." *SIAM Journal on Scientific Computing*, vol. 33, no. 3, 1103–1133, 2011

[9] Weinzierl T. "The Peano Software-Parallel, Automaton-based, Dynamically Adaptive Grid Traversals." *ACM Transactions on Mathematical Software*, vol. 45, no. 2, 1–41, 2019

[10] Knapp D. *A space-filling curve for pyramidal adaptive mesh refinement*. Master's thesis, Rheinische Friedrich-Wilhelms-Universität Bonn, 2020

[11] Burstedde C., Holke J. "Coarse Mesh Partitioning for Tree-Based AMR." *SIAM Journal on Scientific Computing*, vol. Vol. 39, C364–C392, 2017

[12] Burstedde C., Holke J. "A tetrahedral space-filling curve for nonconforming adaptive meshes." *SIAM Journal on Scientific Computing*, vol. 38, C471–C503, 2016

[13] Becker F. *Removing hanging faces from tree-based adaptive meshes for numerical simulations*. Master's thesis, Universität zu Köln, Dezember 2021

[14] Elsweijer S. "Curved Domain Adaptive Mesh Refinement with Hexahedra." Tech. rep., Hochschule Bonn-Rhein-Sieg, Jul. 2021. URL https://elib.dlr.de/143537/

[15] Dreyer L. *The local discontinuous galerkin method for the advection-diffusion equation on adaptive meshes*. Master's thesis, Rheinische Friedrich-Wilhems-Universität Bonn, Februar 2021. URL https://elib.dlr.de/143969/. Erstgutachter: Prof. Dr. Carsten Burstedde, Zweitgutachter: Dr. Johannes Holke

[16] Lilikakis I. *Algorithms for tree-based adaptive meshes with incomplete trees*. Master's thesis, Universität zu Köln, 2022. URL https://elib.dlr.de/191968/

[17] Burstedde C. "Parallel Tree Algorithms for AMR and Non-Standard Data Access." *ACM Trans. Math. Softw.*, vol. 46, no. 4, nov 2020. URL https://doi.org/10.1145/3401990

[18] "JUQUEEN Supercomputer." URL https://hbp-hpc-platform.fz-juelich.de/?page_id=34

[19] "JUWELS Supercomputer." URL https://www.fz-juelich.de/en/ias/jsc/systems/supercomputers/juwels