




# EllipticForest: A Direct Solver Library for Elliptic Partial Differential Equations on Adaptive Meshes

Damyn Chipman <sup>1¶</sup>

<sup>1</sup> Boise State University, USA ¶ Corresponding author

DOI: [10.xxxxxx/draft](https://doi.org/10.xxxxxx/draft)

## Software

- [Review](#) 
- [Repository](#) 
- [Archive](#) 

Editor: [Vissarion Fisikopoulos](#) 

## Reviewers:

- [@sandeshkatakam](#)
- [@lukeolson](#)

Submitted: 24 January 2024

Published: unpublished

## License

Authors of papers retain copyright and release the work under a Creative Commons Attribution 4.0 International License ([CC BY 4.0](#))

## Summary

EllipticForest is a software library with utilities to solve elliptic partial differential equations (PDEs) with adaptive mesh refinement (AMR) using a direct matrix factorization. It implements a quadtree-adaptive variation of the Hierarchical Poincaré-Steklov (HPS) method ([Gillman & Martinsson, 2014](#)). The HPS method is a direct method for solving elliptic PDEs based on the recursive merging of Poincaré-Steklov operators ([Quarteroni & Valli, 1991](#)). EllipticForest is built on top of the parallel and highly efficient mesh library p4est ([Burstedde et al., 2011](#)) for mesh adaptivity and mesh management. Distributed memory parallelism is implemented through the Message Passing Interface (MPI) ([Walker & Dongarra, 1996](#)). EllipticForest wraps the fast, cyclic-reduction methods found the FISHPACK ([Swarztrauber et al., 1999](#)) library and updated in the FISHPACK90 ([Adams et al., 2016](#)) library at the lowest grid level (called leaf patches). In addition, for more general elliptic problems, EllipticForest wraps solvers from the PDE solver library PETSc ([Balay et al., 2023](#)). The numerical methods used in EllipticForest are detailed in ([Chipman et al., 2024a](#)) and the parallel counterpart in ([Chipman et al., 2024b](#)). A key feature of EllipticForest is the ability for users to extend the solver interface classes to implement custom solvers on leaf patches. EllipticForest is an implementation of the HPS method to be used as a software library, either as a standalone to solve elliptic PDEs or for coupling with other scientific libraries for broader applications.

## Statement of Need

Elliptic PDEs arise in a wide-range of physics and engineering applications, including fluid modeling, electromagnetism, astrophysics, heat transfer, and more. Solving elliptic PDEs is often one of the most computationally expensive steps in numerical algorithms due to the need to solve large systems of equations. Parallel algorithms are desirable in order solve larger systems at scale on small to large computing clusters. Communication patterns for elliptic solvers makes implementing parallel solvers difficult due to the global nature of the underlying mathematics. Further complicating implementations of effective solvers, adaptive mesh refinement adds coarse-fine interfaces and more complex meshes that make development and scalability difficult. The solvers implemented in EllipticForest address these complexities through proven numerical methods and efficient software implementations.

The general form of elliptic PDEs that EllipticForest is tailored to solve is the following:

$$\alpha(x, y) \nabla \cdot [\beta(x, y) \nabla u(x, y)] + \lambda(x, y) u(x, y) = f(x, y)$$

where  $\alpha(x, y)$ ,  $\beta(x, y)$ ,  $\lambda(x, y)$ , and  $f(x, y)$  are known functions in  $x$  and  $y$  and the goal is to solve for  $u(x, y)$ . Currently, EllipticForest solves the above problem in a rectangular domain  $\Omega = [x_L, x_U] \times [y_L, y_U]$ . The above PDE is discretized using a finite-volume approach using a

38 standard five-point stencil yielding a second-order accurate solution. This leads to a standard  
39 linear system of equations of the form

$$\mathbf{A}\mathbf{u} = \mathbf{f}$$

40 which is solved via the HPS method, a direct matrix factorization method.

41 Similar to other direct methods, the HPS method is comprised of two stages: a build stage  
42 and a solve stage. In the build stage, a set of solution operators are formed that act as the  
43 factorization of the system matrix corresponding to the discretization stencil. This is done  
44 with  $\mathcal{O}(N^{3/2})$  complexity, where  $N$  is the size of the system matrix. In the solve stage, the  
45 factorization is applied to boundary and non-homogeneous data to solve for the solution vector  
46 with linear complexity  $\mathcal{O}(N)$ . The build and the solve stages are recursive applications of  
47 a merge and a split algorithm, respectively. The advantages of this approach over iterative  
48 methods such as conjugate gradient and multi-grid methods include the ability to apply the  
49 factorization to multiple right-hand side vectors.

50 In addition, another advantage of the quadtree-adaptive HPS method as implemented in  
51 EllipticForest is the ability to adapt the matrix factorization to a changing grid. When the  
52 mesh changes due to a refining/coarsening criteria, traditional matrix factorizations must be  
53 tossed and recomputed. The quadtree-adaptive HPS method can adapt the factorization  
54 locally, eliminating the need to recompute the factorization. This works as the HPS method  
55 builds a set of solution operators that act like a global solution operator. When the mesh  
56 changes, the set can be updated by locally applying the merging and splitting algorithms. This  
57 is especially practical for implicit time-dependent problems that require a full linear solve each  
58 time step.

59 EllipticForest builds upon the p4est mesh library (Burstedde et al., 2011). The quadtree-  
60 adaptive HPS method is uniquely suited for quadtree meshes. p4est, as a parallel and highly  
61 efficient mesh library, provides routines for creating, adapting, and iterating over quadtree  
62 meshes. The routines in EllipticForest wrap or extend the capabilities in p4est. A primary  
63 extension is the development of a *path-indexed* quadtree. This is in contrast to the *leaf-indexed*  
64 quadtree implemented in p4est. A *path-indexed* quadtree is a data structure that stores data  
65 at all nodes in a quadtree, as opposed to just the leaf nodes (see Figure 1). The *path-indexed*  
66 quadtree data structure is designed to store the various data and operators required in the  
67 quadtree-adaptive HPS method.

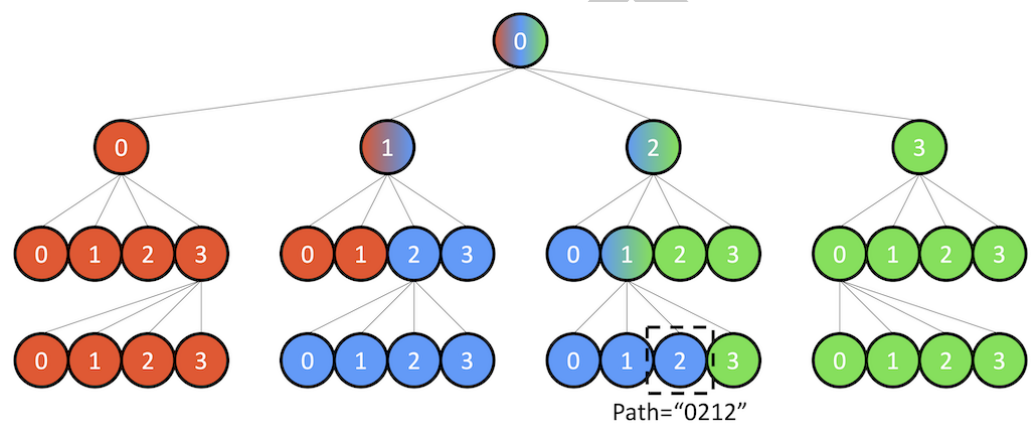
68 The novelty of EllipticForest as software is the implementation of the HPS method for coupling  
69 with other scientific software as well as user extension. Currently, other implementations of the  
70 HPS method are MATLAB or Python codes designed by research groups and used in-house for  
71 solving specific problems (Fortunato et al., 2022; Gillman, 2023; Semenov, 2023). EllipticForest  
72 is designed to be extended and coupled with external libraries. This paper highlights the  
73 software details including the user-friendly interface to the HPS method and the ability for users  
74 to extend the solver interface using modern object-oriented programming (OOP) paradigms.

## 75 Software Overview

76 Below, we outline various components of the software implemented in EllipticForest. These  
77 classes and utilities allow the user to create and refine meshes tailored for their use case, initialize  
78 the solver for the elliptic PDE, and visualize the output solution. A user may also extend the  
79 functionality of EllipticForest through inheritance of the Patch classes for user-defined solvers  
80 at the leaf level.

## Quadtree

The underlying data structure that encodes the mesh is a path-indexed quadtree. The Quadtree object is a class that implements a path-indexed quadtree using a NodeMap, which is equivalent to `std::map<std::string, Node<T>*>`. The template parameter `T` refers to the type of data that is stored on quadtree nodes. The Quadtree implemented in EllipticForest wraps the p4est leaf-indexed quadtree to create, iterate, and operate on the path-indexed quadtree. Functions to iterate over the quadtree include `traversePreOrder`, `traversePostOrder`, `merge`, and `split`. The `traversePreOrder` and `traversePostOrder` functions iterate over the tree in a pre- and post-order fashion, respectively, and provide the user with access to the node or node data via a provided callback function. The `merge` and `split` functions iterate over the tree in a post- and pre-order fashion, respectively, and provide the user with access to a family of nodes, or a group of four siblings and their parent node.



**Figure 1:** A path-indexed quadtree representation of a mesh. Colors indicate which rank owns that node. The nodes colored by gradient indicate they are owned by multiple ranks.

## Mesh

The user interfaces with the domain discretization through the Mesh class. The Mesh class has an instance of the Quadtree detailed above. Mesh provides functions to iterate over patches or cells via `iteratePatches` or `iterateCells`.

Mesh also provides the user with an interface to the visualization features of EllipticForest. A user may add mesh functions via `addMeshFunction`, which are functions in  $x$  and  $y$  that are defined over the entire mesh. This can either be a mathematical function  $f(x, y)$  that is provided via a `std::function<double(double x, double y)>`, or as a `Vector<double>` that has the value of  $f(x, y)$  at each cell in the domain, ordered by patch and then by the ordering of patch grid. Once a mesh function is added to the mesh, the user may call `toVTK`, which writes the mesh to a parallel, unstructured VTK file format. See the section below on output and visualization for more information.

## Patches

The fundamental building block of the mesh and quadtree structures are the patches. A Patch is a class that contains data matrices and vectors that store the solution data and operators needed in the HPS method. A Patch also has an instance of a PatchGrid which represents the discretization of the problem. Each node in the path-indexed quadtree stores a pointer to a Patch.

In EllipticForest, the patch, patch grid, patch solver, and patch node factory interfaces are implemented as a pure virtual interface for the user to extend. Internally, EllipticForest uses

these interfaces to call the implemented solver or discretization. By default, EllipticForest implements a 2nd-order, finite volume discretization and solver. This implementation is found under `src/Patches/FiniteVolume` and each class therein implements the pure virtual interface of the `Patch`, `PatchGrid`, `PatchSolver`, and `AbstractNodeFactory` classes. Users may use the finite volume implementation shipped with EllipticForest, or they may implement different solvers to be used in the HPS method.

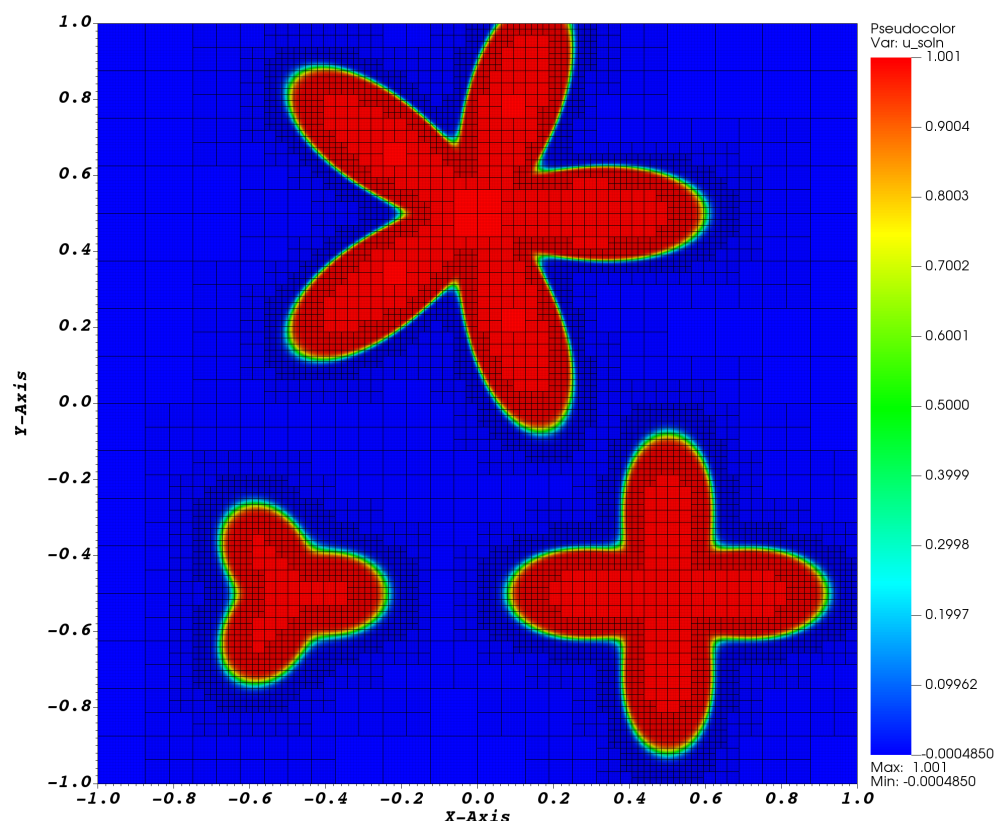
## HPS Solver

Once the mesh has been created and refined, and the patch solver has been initialized, solving the elliptic problem on the input mesh is done by creating an instance of the `HPSAlgorithm` class. The `HPSAlgorithm` class has member functions that perform the setup, build, upwards, and solve stages of the HPS method. As the HPS method is a direct method, once the build stage has been completed, the upwards and solve stages can be called without rebuilding the matrix factorization.

## Output and Visualization

Once the problem has been solved over the entire mesh, each leaf patch in the mesh has the solution stored in one of its data vectors, `vectorU`. This is a discrete representation of the solution to the PDE.

The user may choose to output the mesh and solution in an unstructured PVTK format using the VTK functionality built-in. To output to VTK files, the user first adds mesh functions to the mesh. This includes the solution stored in `vectorU` after the HPS solve. Then, the user calls the `toVTK` member function of the `Mesh` class. This will write a `.pvtu` file for the mesh and a `.pvtu` file for the quadtree. An example of this output for a Poisson equation is shown in [Figure 2](#).



**Figure 2:** Solution of Poisson equation on a quadtree mesh using EllipticForest. The mesh and data are output in an unstructured PVTK format and visualized with VisIt (Childs et al., 2012).

## Acknowledgements

The development of EllipticForest has been funded by the National Science Foundation (NSF-DMS #1819257) and the Boise State University School of Computing. The author acknowledges the assistance and guidance of Dr. Donna Calhoun and Dr. Carsten Burstedde through discussions and direction.

## References

- Adams, J. C., Swarztrauber, P., & Sweet, R. (2016). FISHPACK90: Efficient fortran sub-programs for the solution of separable elliptic partial differential equations. *Astrophysics Source Code Library*, ascl-1609.
- Balay, S., Abhyankar, S., Adams, M. F., Benson, S., Brown, J., Brune, P., Buschelman, K., Constantinescu, E., Dalcin, L., Dener, A., Eijkhout, V., Faibussowitsch, J., Gropp, W. D., Hapla, V., Isaac, T., Jolivet, P., Karpeev, D., Kaushik, D., Knepley, M. G., ... Zhang, J. (2023). *PETSc/TAO users manual* (ANL-21/39 - Revision 3.20). Argonne National Laboratory.
- Burstedde, C., Wilcox, L., & Ghattas, O. (2011). p4est: Scalable algorithms for parallel adaptive mesh refinement on forests of octrees. *SIAM Journal on Scientific Computing*,

- 152 33(3), 1103–1133.
- 153 Childs, H., Brugger, E., Whitlock, B., Meredith, J., Ahern, S., Pugmire, D., Biagas, K.,  
154 Miller, M., Harrison, C., Weber, G. H., Krishnan, H., Fogal, T., Sanderson, A., Garth, C.,  
155 Bethel, E. W., Camp, D., Rübel, O., Durant, M., Favre, J. M., & Navrátil, P. (2012).  
156 VisIt: An end-user tool for visualizing and analyzing very large data. In *High performance*  
157 *visualization—enabling extreme-scale scientific insight* (pp. 357–372).
- 158 Chipman, D., Calhoun, D., & Burstedde, C. (2024a). A fast direct solver for elliptic PDEs on  
159 a hierarchy of adaptively refined quadtrees. *[Forthcoming]*.
- 160 Chipman, D., Calhoun, D., & Burstedde, C. (2024b). Parallel and adaptive optimizations to  
161 the quadtree-adaptive hierarchical poincaré-steklov method. *[Forthcoming]*.
- 162 Fortunato, D., Hale, N., & Townsend, A. (2022). ultraSEM: The ultraspherical spectral element  
163 method. In *GitHub repository*. GitHub. <https://github.com/danfortunato/ultraSEM>
- 164 Gillman, A. (2023). HPS\_demos: A collection of codes applying the HPS method. In *GitHub*  
165 *repository*. GitHub. [https://github.com/agillman20/HPS\\_Demos](https://github.com/agillman20/HPS_Demos)
- 166 Gillman, A., & Martinsson, P. (2014). A direct solver with  $O(N)$  complexity for variable  
167 coefficient elliptic PDEs discretized via a high-order composite spectral collocation method.  
168 *Siam\_sc*, 36(4), A2023–A2046. <https://doi.org/10.1137/130918988>
- 169 Quarteroni, A., & Valli, A. (1991). Theory and application of steklov-poincaré operators for  
170 boundary-value problems. In *Applied and industrial mathematics* (pp. 179–203). Springer.  
171 [https://doi.org/10.1007/978-94-009-1908-2\\_14](https://doi.org/10.1007/978-94-009-1908-2_14)
- 172 Semenov, I. (2023). Streamer\_HPS\_DGSEM: Implementation of the spectral element method  
173 for modelling streamer discharges. In *GitHub repository*. GitHub. [https://github.com/igsemenov/Streamer\\_HPS\\_DGSEM](https://github.com/igsemenov/Streamer_HPS_DGSEM)
- 174 Swarztrauber, P., Sweet, R., & Adams, J. (1999). FISHPACK: Efficient FORTRAN subprograms  
175 for the solution of elliptic partial differential equations. *UCAR Publication*, July.
- 176 Walker, D., & Dongarra, J. (1996). MPI: A standard message passing interface. *Supercomputer*,  
177 12, 56–68.  
178