Brief p4est interface schematics

Carsten Burstedde

May 29, 2013

Abstract

We describe the general procedure of using p4est from application codes. p4est is a software library that stores and modifies a forest-of-octrees refinement structure using distributed (MPI) parallelism. It expects the description of the domain as a coarse mesh of conforming hexahedra. Non-conforming adaptive mesh refinement (AMR), coarsening, and other operations that modify the forest are implemented as p4est API functions. To inform the application about the refinement structure, several methods are provided that encode this information.

1 Starting point

We generally separate the adaptive mesh refinement (AMR) topology from any numerical information. The former is stored and modified internally by the p4est software library, while an application is free to define the way it uses this information and arranges numerical and other data. This document is inteded to describe the interface by which p4est relates mesh information to the application.

The general, modular AMR pipeline is described in [3], which is not specific to p4est but can in principle be applied to any AMR provider. The p4est algorithms and main interface routines are destribed in [4]. An example usage of p4est as scalable mesh backend for the general-purpose finite element software deal.II is described in [1]. A reference implementation of p4est in C can be freely downloaded [2] and used and extended under the GNU General Public License. This code contains commented header files and simple examples and use cases. This software is best installed standalone into a dedicated directory, where an application code can then find the header and library files to compile and link against, respectively.

In this document, we document the three distinct tasks to

A create a coarse mesh (Figure 1),

B modify the refinement and partition structure internal to p4est (Figure 2),

C and to relate the mesh information to an application (Figure 3).

Unless indicated otherwise, all operations described below are understood as MPI collectives, that is, they are called on all processors simultaneously. Currently, part A needs to be performed redundantly on all processors, which is acceptable for up to 10^5–10^6 coarse cells (octree roots). Parts B and C strictly implement distributed parallelism, where runtime and per-processor memory are roughly proportional to the number of local elements (octree leaves) on a given processor, independent of the number of octrees, the total number of elements, or the refinement depth. Partitioning the forest into equal numbers of elements per processor is a core p4est operation and

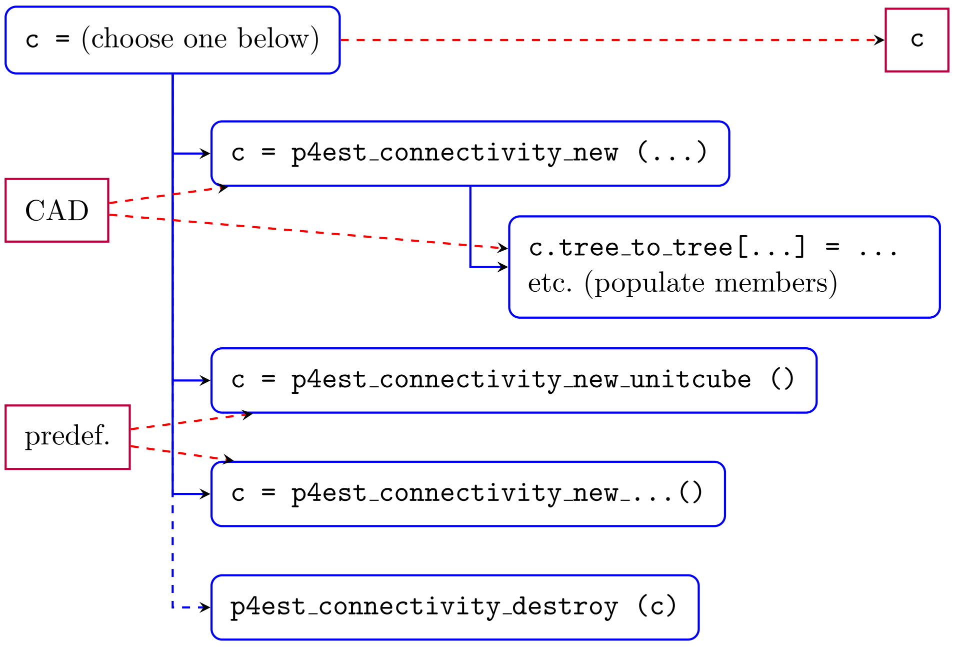

c = (choose one below)
c
c = p4est_connectivity_new (...)
CAD
c.tree_to_tree[...] = ...
etc. (populate members)
c = p4est_connectivity_new_unitcube ()
predef.
c = p4est_connectivity_new_...()
p4est_connectivity_destroy (c)

Figure 1: Part A, creating the coarse mesh connectivity. The `p4est` connectivity `c` is a `C struct` that contains numbers and orientations of neighboring coarse cells. It can be created by translating CAD or mesh data file formats or by using one of several predefined `p4est` functions. The data format is documented in the extensive comment blocks in `p4est_connectivity.h` (2D) and `p8est_connectivity.h` (3D); see also [4]. In the following, `p4est` always refers to both 2D and 3D.

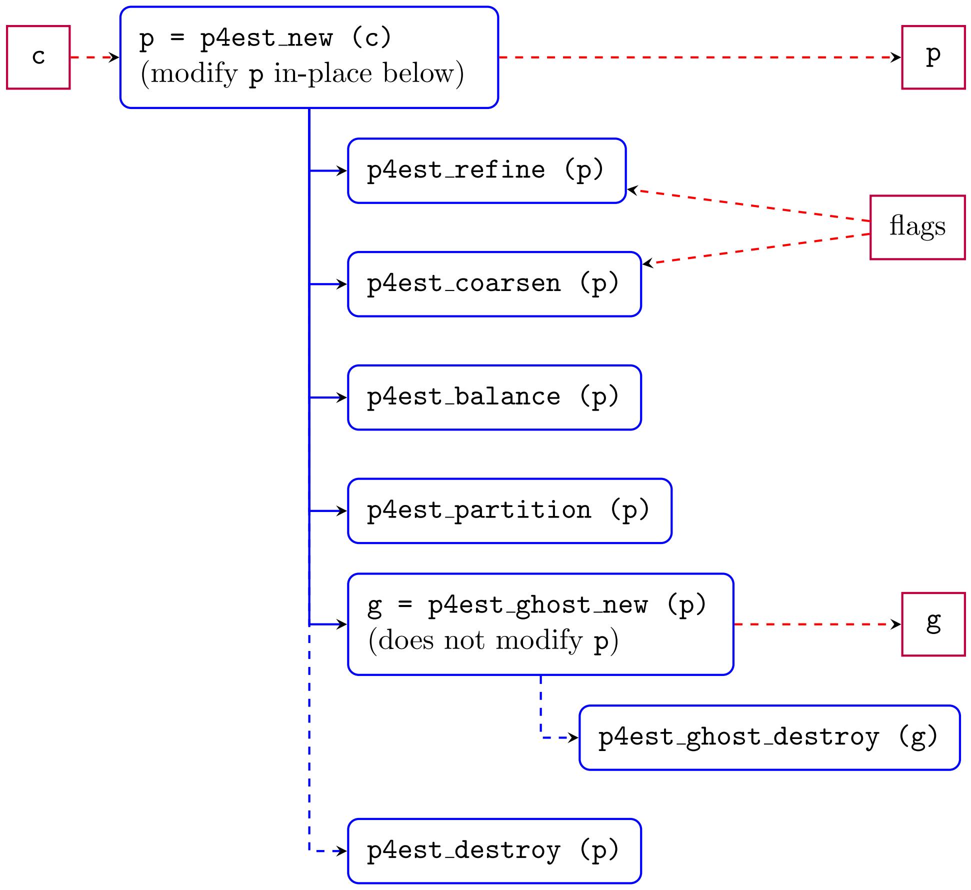


Figure 2: Part B, changing the refinement structure and partition. With the connectivity `c` created in part A, we can create the distributed `p4est` structure. Several functions for its modification exist. For a given `p4est` snapshot, we can create a ghost layer `g` of off-processor leaves, which will be outdated and should be destroyed once `p` is changed again. Refinement and coarsening are controlled by callback functions that usually query flags determined by the application. The `C structs` `p` and `g` can be inspected directly by an application, for example to loop through data associated with local and ghost leaves.

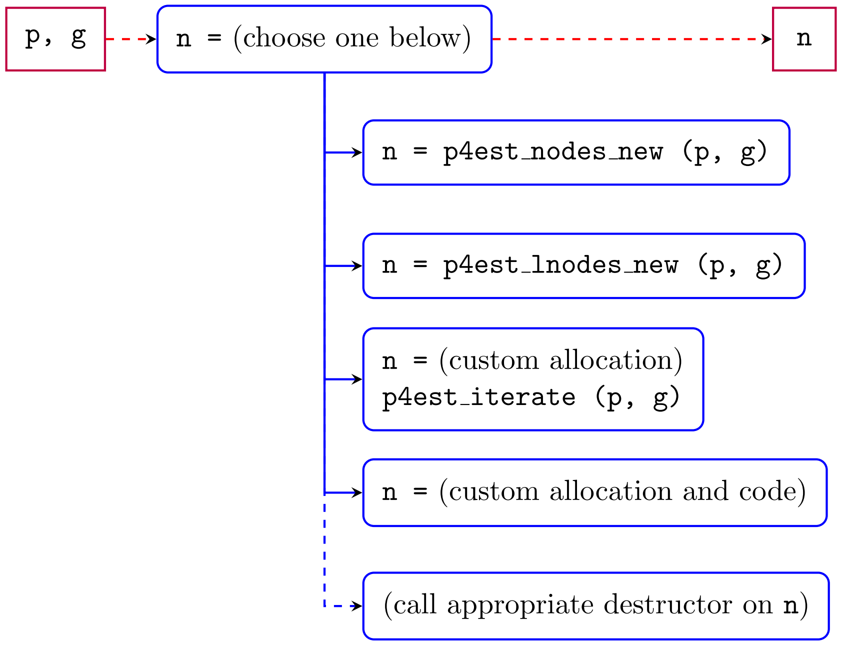
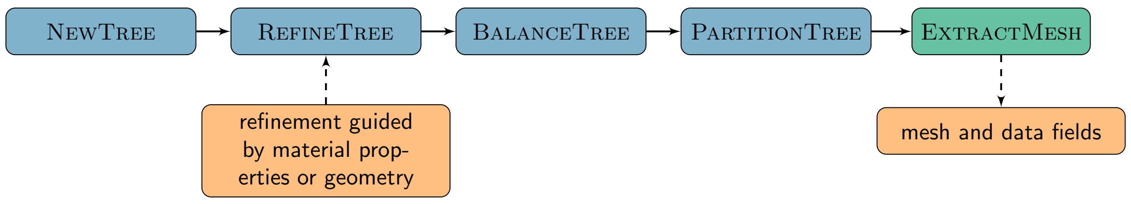
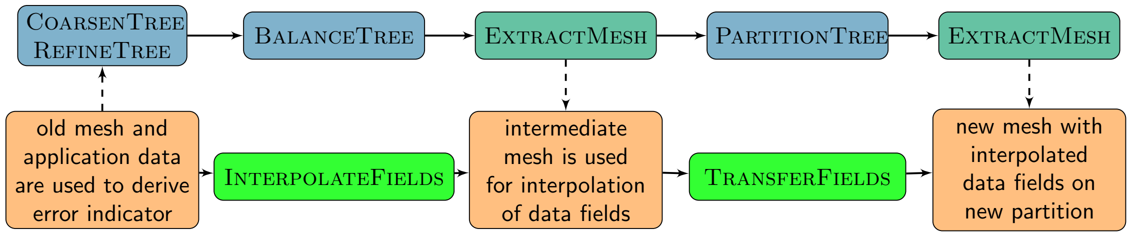

p, g
n = (choose one below)
n
n = p4est_nodes_new (p, g)
n = p4est_lnodes_new (p, g)
n = (custom allocation)
p4est_iterate (p, g)
n = (custom allocation and code)
(call appropriate destructor on n)

Figure 3: Part C, creating an application-specific numbering of degrees of freedom. The `nodes` and `lnodes` constructors create a processor-local view of globally unique node numbers and the dependency lists for hanging nodes for continuous tensor-product piecewise linear and piecewise polynomial finite elements, respectively. The iterator provides a generic way to traverse the refinement structure and to have callbacks executed for every face, edge, and corner neighborhood, which can then be used to identify node locations and their hanging status for any custom element type. The `C` prototypes and documentation can be found in the respective `p4est .h` files.



(a) Generating the initial refinement structure and mesh.


CoarsenTree RefineTree
BalanceTree
ExtractMesh
PartitionTree
ExtractMesh
old mesh and application data are used to derive error indicator
InterpolateFields
intermediate mesh is used for interpolation of data fields
TransferFields
new mesh with interpolated data fields on new partition

(b) A-posteriori change of refinement structure and mesh.

Figure 4: Separation between `p4est` on one hand and application-specific mesh and numerical information on the other. The dark blue nodes in the picture correspond to in-place operations on the forest `p` that fall under part B (Figure 2). The dark green node labeled ExtractMesh refers to creating a node numbering structure `n` (part C; see Figure 3) and the allocation of numerical space in application memory guided by `n`. The bright green fields in the bottom-most row refer to moving the application-specific numerical information from the old mesh to the new one. We suggest doing this in two steps: Interpolation works processor-local since the partition has not been changed at this point. Then, after partitioning, transfer moves data to the new owner processors of the respective degrees of freedom without changing the refinement pattern any further. Both operations require the existence of both the old and new `n` structures, and allow for freeing the older one afterwards. (Plots originally published in [3].)

essentially free in terms of execution time, so we suggest to call it whenever load balance is desirable for subsequent operations.

The definition and organisation of numerical data is entirely left up to the application. The application calls modification operations for the forest (part B), guided by the numerical state related to the local leaves. A numerical application will require its own numbering scheme for degrees of freedom, which is most of the time defined via node locations on a reference element and dependencies between hanging nodes and the independent nodes they are interpolated from. `p4est` provides several paths for aiding the application in defining their data layout, henceforth called the mesh (part C), and not to be confused with the coarse octree mesh (part A). Figure 4 contains illustrations on the sequence of operations that are required to create a new refinement pattern from scratch or by modifying an existing one, and to move the application-specific numerical data from an old to a new mesh.

References

[1] W. Bangerth, C. Burstedde, T. Heister, and M. Kronbichler, *Algorithms and data structures for massively parallel generic adaptive finite element codes*, ACM Transactions on Mathematical Software, 38 (2011), pp. 14:1–14:28.

[2] C. Burstedde, *p4est: Parallel AMR on forests of octrees*, 2010. `http://www.p4est.org/`.

[3] C. Burstedde, O. Ghattas, G. Stadler, T. Tu, and L. C. Wilcox, *Towards adaptive mesh PDE simulations on petascale computers*, in Proceedings of Teragrid '08, 2008.

[4] 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 (2011), pp. 1103–1133.