The p4est software for parallel AMR
Howto document and step-by-step examples

Carsten Burstedde, Lucas C. Wilcox, and Tobin Isaac

January 6, 2015

Abstract

We introduce the p4est software library for parallel adaptive mesh refinement (AMR). It represents the mesh as one or more conforming hexahedra that can be individually refined as adaptive octrees, yielding a distributed non-conforming mesh. Both 2D and 3D are supported. p4est is intended to be compiled and linked against by numerical simulation codes that require efficient in-core, static or dynamic load balancing and (re-)adaptation of the computational mesh. This document presents a detailed description of the main interface and its basic usage and walks the reader through several example programs, demonstrating the design and usage of the algorithms. The source code and all related documents are available on the web page www.p4est.org.

Contents

1 Introduction

Many applications in applied mathematics and numerical simulation use a mesh of computational cells that covers the domain of interest. The numerical solution is then approximated by functions, each of which is associated with a small set of cells (or even one). Dynamic *adaptive* methods change the mesh during the simulation by local refinement or coarsening, and *parallel* methods distribute (*partition*) the mesh between multiple processors, where each processor ideally receives an equal share of the computational load. `p4est` isolates the task of parallel dynamic adaptive mesh refinement (AMR), coarsening, and load balancing by encapsulating algorithms that scale well to large numbers ($> 10^5$) of processors.

The `p4est` algorithms are predominantly used for the numerical solution of partial differential equations, but also support various other tasks where fast hierarchical space subdivision is required, for example to locate particles in space or to organize, sort, and search in large data sets. This is possible since `p4est` separates the AMR topology from any numerical information: The former is stored and modified internal to the `p4est` software library, while an application is free to define the way it uses this information and arranges numerical and other data.

A general AMR simulation pipeline is described in [7], which is not specific to `p4est` but can in principle be implemented using any AMR library. The `p4est` algorithms and main interface routines are defined in [9] and [12], which are also the suggested citations for users of the software. An example usage of `p4est` as scalable mesh backend for the general-purpose finite element software `deal.II` is described in [3]. A reference implementation of `p4est` in `C` can be freely downloaded [5] and used and extended under the GNU General Public License. The code contains commented header files and simple example programs, several of which are documented in Section 4 below.

2 Interface schematics

The basic structure used in `p4est` is a *connectivity* of quadtrees (2D) or octrees (3D) which covers the domain of interest in a conforming macro-mesh. This includes the case of using exactly one tree for representing the hypercube. These trees can then be arbitrarily refined and coarsened, where the storage of quadrants/octants is distributed in parallel. Thus, the mesh primitives used in `p4est` are quadrilaterals in 2D and hexahedra in 3D. The adaptive structure allows for quadrants/octants of different sizes to neighbor each other, which is commonly called *non-conforming.* This concept leads to *hanging* faces and edges.

In this document, we cover 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 in these figures 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 usually negligible 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,

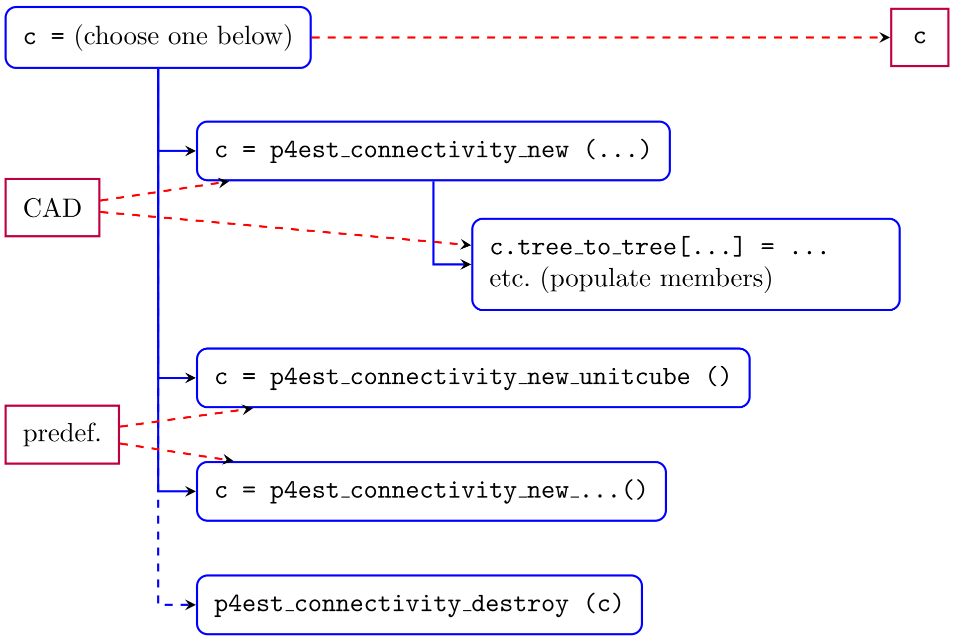

c = (choose one below)
c
c = p4est_connectivity_new (...)
CAD
c.tree_to_tree[...] = ...
etc. (populate members)
c = p4est_connectivity_new_unitcube ()
predef.
c = p4est_connectivity_new_...()
p4est_connectivity_destroy (c)

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 (i.e., the octree roots). It can be created by translating CAD or mesh data file formats or by using one of several predefined* `p4est` *functions (for example to select the unit cube). Once created, it is generally not changed anymore. The data format is documented in the extensive comment blocks in* `p4est_connectivity.h` *(2D) and* `p8est_connectivity.h` *(3D); the encoding is also defined in [9]. In the following,* `p4est` *always refers to both 2D and 3D.*

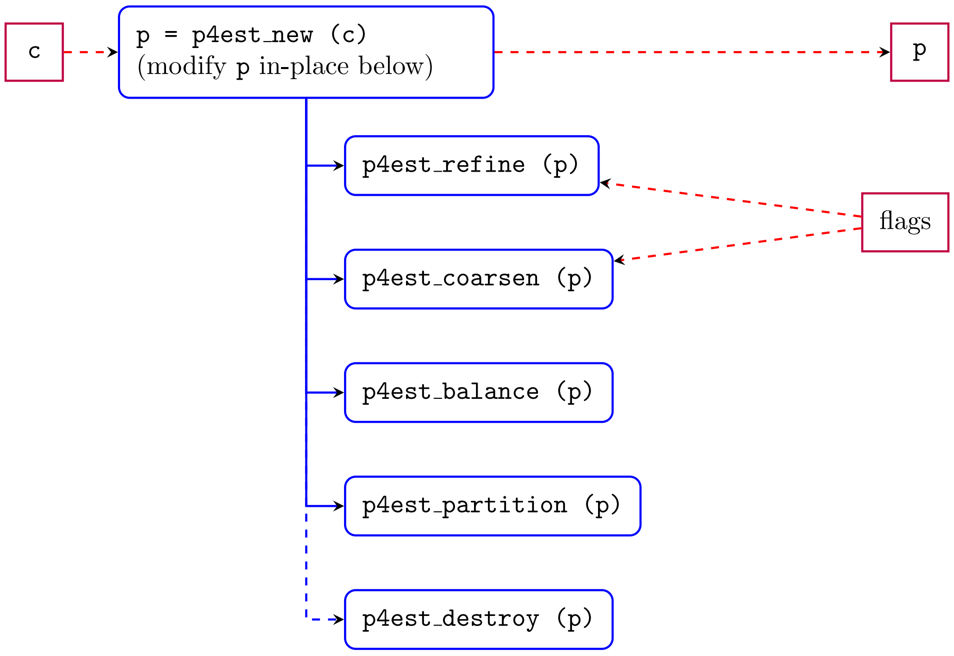


Figure 2: *Part B, changing the refinement structure and partition. With the connectivity* `c` *created in part A, we can build the distributed* `p4est` *structure and modify it in place. Refinement and coarsening are controlled by callback functions that usually query flags determined by the application.* `p4est_balance` *ensures 2:1 size balance between neighbors and is optional, depending on the application's requirements.* `p4est_partition` *redistributes the cells (octants) in parallel and can optionally take a per-octant weighting callback. The* `C struct p` *must only be changed by these API calls and is understood as read-only otherwise. It can be inspected directly by an application, for example to loop through data associated with local leaves.*

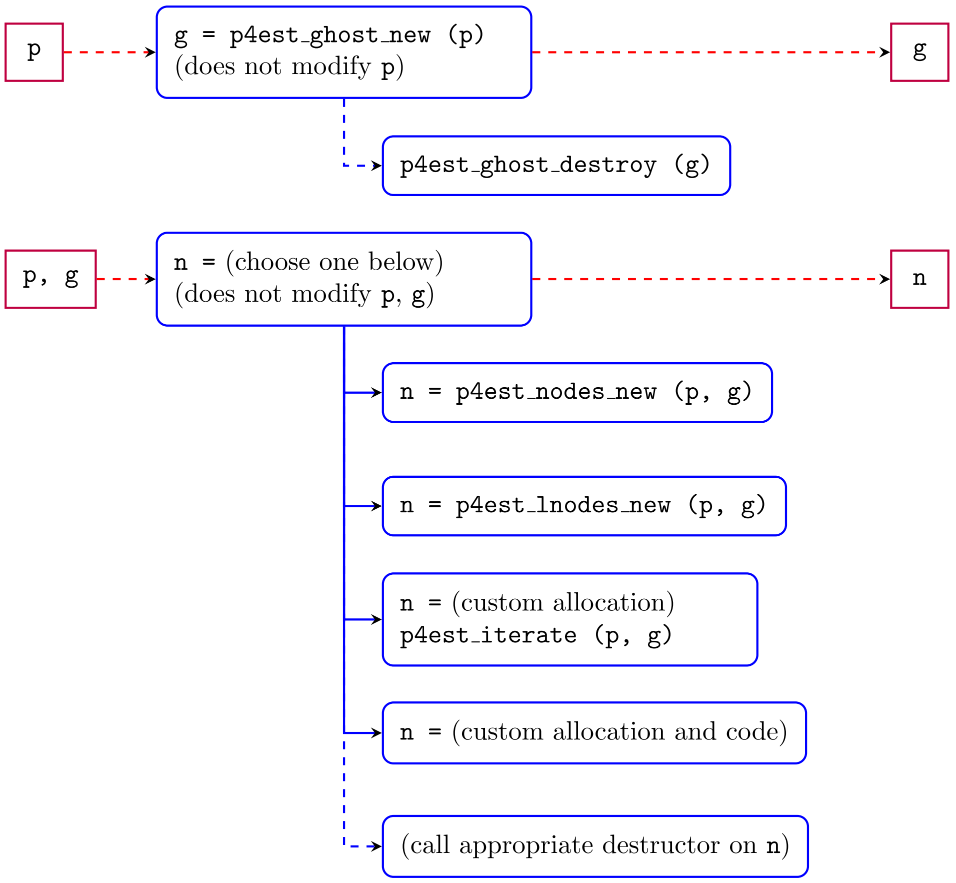
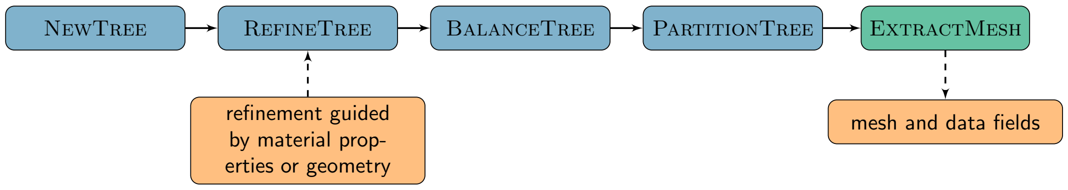
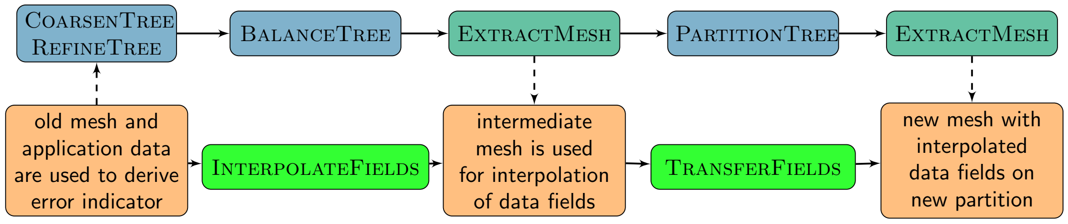


Figure 3: *Part C, creating an application-specific numbering of degrees of freedom. For a given* `p4est` *snapshot, we first create a ghost layer* `g` *of off-processor leaves, which will be outdated and should be destroyed once* `p` *is changed again in part B. The* `p4est_nodes` *and* `p4est_lnodes` *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 be used to identify node locations and their hanging status for any custom element type.* `p4est_iterate` *can also be used for applications that have no concept of nodes at all and just need to query the local mesh neighborhood.*



(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 [7].)*

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 proposed 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.

3 Building and installing `p4est`

The `p4est` software is organized as a library and comes with several example programs and tests that link to the library. The software is preferably installed standalone into a dedicated directory, where an application developer can then find the header and library files to compile and link against.

`p4est` does not have any library dependencies except for a recent zlib version [10]. `p4est` uses the `adler32_combine` function for a parallel checksum and relies on zlib compression in the VTK output routines. If `configure` does not detect this function, it will print a message saying that these features will not work. The user can fix this for example by installing zlib's development binary package, or by compiling a current zlib source and pointing the `LIBS` and `CFLAGS` environment variables to it. The submodule `sc` also checks for a development version of the Lua library. Lua is not used in `p4est`; the message can be ignored or fixed in the same way.

3.1 Installation from a downloaded `.tar.gz` file

`p4est` uses the GNU autoconf/automake/libtool build system. The `INSTALL` contains generic instructions that apply to `p4est` as well. The `p4est` tar archive posted on the web page contains the configure script and all required auxiliary files. As detailed in the `INSTALL` document, it honors certain environment variables and command line options. The following options to `configure` are often used.

`--enable-mpi` is necessary for parallel functionality. Without this option, `p4est` does not require MPI and replaces the calls to it by a dummy implementation that always assumes an `MPI_Comm_size` of 1 and an `MPI_Comm_rank` of 0. With this option, it uses the original MPI calls.

`--enable-debug` enables assertions and additional code for verification. This is generally helpful for development, even if it is somewhat slower and produces a lot of diagnostic log messages.

`CFLAGS="-Wall -g -O0"` is an example of setting flags for a development build compiled with the GNU compilers. `p4est` is written in plain `C` and does not require `FFLAGS` or `CXXFLAGS`.

Both in-source and out-of-source builds are supported. For a one-time installation, `./configure` can be called in the root directory of the unpacked tree. For development or for testing multiple configure options, it is advised to create an empty directory for each configuration and invoke configure from there with a relative path.

The subpackage `sc` is contained in the tarball and used by default. It is possible to use a version of `sc` that has been `make install`'d independently (usually not worth the effort for one-time builds):

1. Change into a new empty directory and call `sc/configure` with its relative path. Use `--prefix=`*path-to-install-sc* to specify the destination for the compiled files. Then call `make install`.

2. Create another empty directory and call the `p4est` configure with its relative path and the options `--prefix=`*path-to-install-p4est* and `--with-sc=`*path-to-install-sc*. Make sure to use the same compiler flags and configure options as in the previous step. Finally, `make install`.

The above is the recommended procedure to split the `p4est` installation into two packages, say for a linux binary distribution. The benefit is that other packages that might require `sc` do not force `p4est` to get installed. The administrator may or may not choose to put the header and library files for the two packages into different directories.

3.2 Installation from source

When installing from source, GNU autotools must be invoked to generate the necessary configuration files and the configure script. A bootstrap script is provided for this purpose: give the shell command `./bootstrap` before following the instructions in the `INSTALL` file. Then proceed as in Section 3.1.

3.3 Installation from a git repository

If you have obtained `p4est` from a git repository, the `sc` submodule must be cloned before installation. This can be accomplished with the shell command `git submodule init && git submodule update`. After the submodule has been obtained, you can install from source as described above in Section 3.2.

3.4 Using `p4est` through the deal.II interface

The generic adaptive finite element software library `deal.II` interfaces to `p4est` since version 7.0 to parallelize its mesh handling. We have added the script `doc/p4est-setup.sh` to compile and install `p4est` with predefined options; this creates both a debug and production version with minimal logging. To know what is going on within `p4est` at run time, the log level needs to be relaxed in the script.

4 Example programs

This section presents several instructive example programs that work with `p4est`. They are part of the source code under the `example/steps/` directory. The examples compile and run and are commented and understandable by itself. In this document we comment on (slightly modified) excerpts and provide additional background information about the logical structure of `p4est`. The examples are optimized for clarity, not for speed—please adapt as needed.

4.1 Hello world

The program `example/step/p4est_step1` creates a refinement pattern from image data and writes an output file in VTK format. It comes in two variants for 2D and 3D (both compiled from the same source file to prevent redundancy). The image has dimensions 32×32, which we state as follows:

```
#define P4EST_STEP1_PATTERN_LEVEL 5
#define P4EST_STEP1_PATTERN_LENGTH (1 << P4EST_STEP1_PATTERN_LEVEL)
static const int    plv = P4EST_STEP1_PATTERN_LEVEL;
static const int    ple = P4EST_STEP1_PATTERN_LENGTH;
```

We will have to scale this information to match the unit cube domain that we choose for our mesh connectivity (cf. the interface schematics in Section 2). The code below is from them `main` function.

```
  mpicomm = sc_MPI_COMM_WORLD;         /* #define'd depending on --enable-mpi. */
#ifndef P4_TO_P8
  conn = p4est_connectivity_new_unitsquare ();
#else
  conn = p8est_connectivity_new_unitcube ();
#endif
  p4est = p4est_new (mpicomm, conn, 0, NULL, NULL);
```

At this point in the main program we have a parallel mesh structure that is initially unrefined, that is, the whole domain consists of one root octant. To refine the mesh we call

```
  recursive = 1;
  p4est_refine (p4est, recursive, refine_fn, NULL);
```

which walks through the forest, in this case recursively, and queries the refinement flag via the callback function `refine_fn`. This function receives a pointer to a `p4est_quadrant` that has coordinate members `x`, `y` (and `z` in 3D). `p4est` works in integer coordinates of type `p4est_qcoord_t` and defines the length of a tree root as a power of two (this is from `p8est.h`):

```
#define P8EST_MAXLEVEL 18
#define P8EST_ROOT_LEN ((p4est_qcoord_t) 1 << P8EST_MAXLEVEL)
```

Consequently, a quadrant of refinement level $0 \leq \ell \leq 5$ covers $2^{5-\ell}$ pixels of the image, which we then iterate through to react to dark values that we use as refinement indicators.

```
static int
refine_fn (p4est_t * p4est, p4est_topidx_t which_tree,
           p4est_quadrant_t * quadrant)
{
  /* omitted: manage 3D extrusion and levels greater than 5 */
  tilelen = 1 << (plv - quadrant->level);       /* Pixel size of quadrant */
  offsi = quadrant->x / P4EST_QUADRANT_LEN (plv);       /* Pixel x offset */
  offsj = quadrant->y / P4EST_QUADRANT_LEN (plv);       /* Pixel y offset */
  for (j = 0; j < tilelen; ++j) {
    for (i = 0; i < tilelen; ++i) {
      p = hw32_header_data[ple * (ple - 1 - (offsj + j)) + (offsi + i)];
      if (p < 128) {
        return 1;
      }
    }
  }
  return 0;
}
```

The refinement pattern is now established, but since all quadrants are children of the original root, they are on the same processor. (This is a situation that we usually avoid by refining with `recursive = 0;` and repartitioning after every refinement/coarsening. See the finite element solver discussed in Section 4.4 for the recommended procedure.) To equidistribute the quadrants in parallel, call:

```
  partforcoarsen = 0;
  p4est_partition (p4est, partforcoarsen, NULL);
```

The partition-for-coarsening flag may be set to true to avoid that any family of equal-size quadrant siblings is split between processors. In this example, we also exercise the 2:1 size balance for demonstration purposes; if any neighbors across a face have a size difference bigger than two then the larger neighbor is refined (recursively if necessary).

```
  balance = 1;
  if (balance) {
    p4est_balance (p4est, P4EST_CONNECT_FACE, NULL);
    p4est_partition (p4est, partforcoarsen, NULL);
  }
```

We may strengthen the balance requirement to `P4EST_CONNECT_EDGE` or `P4EST_CONNECT_CORNER`. Now it remains to write output for visualizing the mesh as displayed in Figure 5.

```
  p4est_vtk_write_file (p4est, NULL, P4EST_STRING "_step1");
```

We have opted against reference counting or garbage collection; the user is welcome to implement such features in their own wrapper code. The destruction of the forest and connectivity is done as follows.

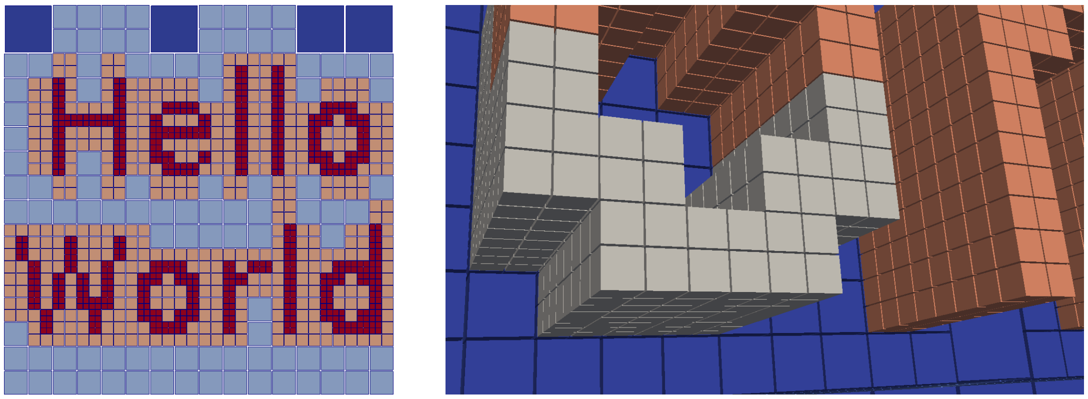


Figure 5: *Left: ParaView visualization of the VTK files created by* `example/steps/p4est_step1` *(2D), color coded by refinement level. Right: results from* `mpirun -np 5 example/steps/p8est_step1` *(3D). These programs start with a unit square/cube domain and refine it according to a grayscale image that contains the text "Hello World." The 3D plot has been created using a background selection of refinement levels 4–6 (blue) and a foreground selection of octants at level 6, color coded by MPI rank. Both programs enforce 2:1 size balance across the mesh faces to grade the refinement.*

```
  p4est_destroy (p4est);
  p4est_connectivity_destroy (conn);
```

Reading the actual source code, the reader will see the MPI initialization/finalization and additional macros to check assertions.

4.2 Creating a connectivity from a mesh file

The second example `example/step/p4est_step2` reads in a connectivity from a mesh file and then adapts, balances, and distributes it. We have chosen to implement an ABAQUS `.inp` file reader for quadrilateral and hexahedral meshes since both Cubit (`https://cubit.sandia.gov/`) and Gmsh (`http://geuz.org/gmsh/`) can output to this format. The `.inp` reader can easily be be adapted to other formats.

To generate a quadrilateral mesh using Gmsh one needs to first download and install it from `http://geuz.org/gmsh/`. In addition to the source, on the website there are prebuilt binaries for Windows, Mac OS X, and Linux. After obtaining Gmsh, the next thing to do is generating a domain geometry. Here is a example geometry file for a cube with a cylindrical hole.

```
lc = 1;
r1 = 5;
r2 = 3;

Point(1) = {-r1,-r1,-r1,lc};
Point(2) = { r1,-r1,-r1,lc};
Point(3) = { r1, r1,-r1,lc};
Point(4) = {-r1, r1,-r1,lc};

Point(5) = {  0,  0,-r1,lc};

Point(6) = {  0,-r2,-r1,lc};
Point(7) = {  0, r2,-r1,lc};
Point(8) = { r2,  0,-r1,lc};
Point(9) = {-r2,  0,-r1,lc};

Line(1) = {1,2};
Line(2) = {2,3};
Line(3) = {3,4};
Line(4) = {4,1};

Circle(5) = {8, 5, 7};
Circle(6) = {7, 5, 9};
Circle(7) = {9, 5, 6};
Circle(8) = {6, 5, 8};

Line Loop(9)  = {1,2,3,4};
Line Loop(10) = {5,6,7,8};

Plane Surface(11) = {9, 10};
Recombine Surface {11};

Extrude {0,0,2*r1} {
  Surface{11}; Layers{2*r1/lc}; Recombine;
}
```

A 3D mesh file `hole.inp` can be generated from a geometry file `hole.geo` with the following command.

```
gmsh -3 -o hole.inp hole.geo
```

The mesh reader function in `p4est` reads a basic ABAQUS file supporting element types with the prefix `C2D4`, `CPS4`, and `S4` in 2D and prefix `C3D8` in 3D reading them as bilinear quadrilateral and trilinear hexahedral trees, respectively.

A basic 2D `.inp` mesh is given as

```
*Heading
 box.inp
*Node
1,  -5, -5, 0
2,   5, -5, 0
3,   5,  5, 0
4,  -5,  5, 0
5,   0, -5, 0
6,   5,  0, 0
7,   0,  5, 0
8,  -5,  0, 0
9,   1, -1, 0
10,  0,  0, 0
11, -2,  1, 0
*Element, type=CPS4, ELSET=Surface1
1,  1, 10, 11, 8
2,  3, 10, 9,  6
3,  9, 10, 1,  5
4,  7,  4, 8, 11
5, 11, 10, 3,  7
6,  2,  6, 9,  5
```

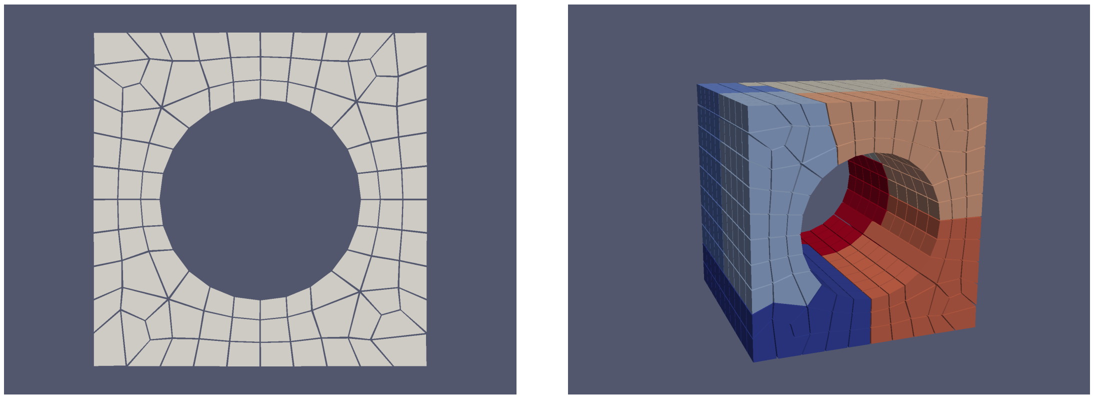


Figure 6: *Left: ParaView visualization of the VTK files created by* `example/steps/p4est_step2 example/steps/hole_2d_gmsh.inp` *(2D). This is the octree connectivity that can be adapted and balanced just like the built-in connectivities. Right: Results from* `mpirun -np 7 example/steps/p8est_step2 example/steps/hole_3d_gmsh.inp` *(3D), color coded by MPI rank. Note that the paths to the mesh files may differ depending on the installation.*

The `*Node` section contains the vertex number and x, y, and z components for each vertex. The `*Element` section gives the element number and the 4 vertices in 2D (8 vertices in 3D) of each element in counterclockwise order (for 3D the vertices are give in counterclockwise order for the bottom face followed by the top face).

The setup for this example is the same as for `example/step/p4est_step1` with the main differences being the refinement function and the construction of the connectivity of the forest. For this example we use a uniform refinement function and call it iteratively, partitioning after each new level of refinement.

The connectivity is created from an `.inp` mesh file with name `filename.inp` with the function call

```
conn = p4est_connectivity_read_inp (filename);
```

Depending on the mesh generator it may be useful to reorder the connectivity to improve the communication pattern before building the forest. If `p4est` is configured with METIS (`http://www.cs.umn.edu/~metis`) graph partitioner support (using the flag `--with-metis`) then we can use it reorder the connectivity for an optimized partitioning of the forest across processors. (We stress that METIS is not required in practice; asymptotic scalability is fine either way.) The following call will build the dual graph based on face connectivity to determine the partition.

```
#ifdef P4EST_WITH_METIS
p4est_connectivity_reorder (mpicomm, 0, conn, P4EST_CONNECT_FACE);
#endif /* P4EST_WITH_METIS */
```

Once the connectivity is read in from the file and possibly reordered it can be used just like the built-in connectivities. Visualizations of some generated meshes are displayed in Figure 6.

4.3 Iterating through an adaptive mesh: a parallel advection example

The emphasis of the first two examples was creating, adapting, and distributing a forest. The third example `example/step/p4est_step3` uses a simple advection problem (uniform velocity on a periodic domain) to demonstrate two common tasks for data associated with quadrants/octants: moving data between neighboring partitions, and comparing data in neighboring cells.

`p4est` has several options (see Figure 3) for relating mesh information to an application so that it can manage the data used to represent e.g. the solution of a PDE, but in this example we use `p4est`'s internal mechanisms for associating data with a forest and with each individual quadrant.

A forest has a `user_pointer` to keep track of the forest's application context. This can be a pointer to anything: in this example it is a pointer to a `C struct` with the problem description.

```
typedef struct step3_ctx
{
  double              center[P4EST_DIM];  /**< coordinates of the center of
                                               the initial condition Gaussian
                                               bump */
  double              bump_width;         /**< width of the initial condition
                                               Gaussian bump */
  // ...
}
step3_ctx_t;
```

To associate data with each quadrant, we define a `C struct` with the data fields that we want,

```
typedef struct step3_data
{
  double              u;             /**< the state variable */
  double              du[P4EST_DIM]; /**< the spatial derivatives */
  double              dudt;          /**< the time derivative */
}
step3_data_t;
```

and a routine for initializing data,

```
static void
step3_init_initial_condition (p4est_t * p4est, p4est_topidx_t which_tree,
                              p4est_quadrant_t * q)
{
  /* the data associated with a forest is accessible by user_pointer */
  step3_ctx_t        *ctx = (step3_ctx_t *) p4est->user_pointer;
  /* the data associated with a quadrant is accessible by p.user_data */
  step3_data_t       *data = (step3_data_t *) q->p.user_data;
  double              midpoint[3];

  step3_get_midpoint (p4est, which_tree, q, midpoint);
  /* initialize the data */
  data->u = step3_initial_condition (midpoint, data->du, ctx);
}
```

which we pass to `p4est_new()` or `p4est_new_ext()`, the latter being a version that exposes additional control parameters.

```
  /* Create a forest that has 16 (2^4) cells in each direction */
  p4est = p4est_new_ext (mpicomm, /* communicator */
                         conn,    /* connectivity */
                         0,       /* minimum quadrants per MPI process */
                         4,       /* minimum level of refinement */
                         1,       /* fill uniform */
                         sizeof (step3_data_t),         /* data size */
                         step3_init_initial_condition,  /* initializes data */
                         (void *) (&ctx));              /* context */
```

To pass the quadrants' data to neighboring processes, we use the ghost layer object `p4est_ghost_t` (see `p4est_ghost.h`).

```
  p4est_ghost_t      *ghost;

  /* create the ghost quadrants */
  ghost = p4est_ghost_new (p4est, P4EST_CONNECT_FULL);
  /* create space for storing the ghost data */
  ghost_data = P4EST_ALLOC (step3_data_t, ghost->ghosts.elem_count);
  /* synchronize the ghost data */
  p4est_ghost_exchange_data (p4est, ghost, ghost_data);
```

(Note that if you have quadrant-associated data that is not managed by `p4est`'s internal mechanism, you can still exchange ghost layer data with `p4est_ghost_exchange_custom()`.) The above ghost exchange is a synchronization point; it would be easy to extend it to overlap it with other computations.

In this advection example, we use an upwinding flux scheme, which means that we need to compare the data of neighboring quadrants. There are several cases to consider: a quadrant's face neighbor can be bigger, equal or smaller than itself (in this example `p4est_balance()` is used, so the size ratio can only be 2:1); the neighbor can be local or a ghost quadrant; the neighbor may be in a different tree of the forest. The function `p4est_iterate()` is designed to handle these issues by means of user supplied callback functions. This is how the fluxes between quadrants are calculated in this example.

```
    p4est_iterate (p4est,                 /* the forest */
                   ghost,                 /* the ghost layer */
                   (void *) ghost_data,   /* the synchronized ghost data */
                   step3_quad_divergence, /* callback to compute each quad's
                                             interior contribution to du/dt */
                   step3_upwind_flux,     /* callback to compute each quads'
                                             faces' contributions to du/dt */
#ifdef P4_TO_P8
                   NULL,                  /* there is no callback for the
                                             edges between quadrants */
#endif
                   NULL);                 /* there is no callback for the
                                             corners between quadrants */
```

Here are the first few lines of `step3_upwind_flux()`, where the data describing the two sides of a face are extracted.

```
static void
step3_upwind_flux (p4est_iter_face_info_t * info, void *user_data)
{
  // ... (other declarations)
  p4est_iter_face_side_t *side[2];
  sc_array_t         *sides = &(info->sides);

  /* because there are no boundaries, every face has two sides */
  P4EST_ASSERT (sides->elem_count == 2);

  side[0] = p4est_iter_fside_array_index_int (sides, 0);
  side[1] = p4est_iter_fside_array_index_int (sides, 1);
  // ...
```

a look at the definition of `p4est_iter_face_side_t` in `p4est_iterate.h` shows all of the context that is automatically calculated for the user.

```
typedef struct p4est_iter_face_side
```

```
{
  p4est_topidx_t      treeid;          /**< the tree on this side */
  int8_t              face;            /**< which quadrant side the face
                                            touches */
  int8_t              is_hanging;      /**< boolean: one full quad (0) or
                                            two smaller quads (1) */
  /* if is_hanging == 0, use is.full to access per-quadrant data;
   * if is_hanging == 1, use is.hanging to access per-quadrant data */
  union p4est_iter_face_side_data
  {
    struct
    {
      int8_t              is_ghost;    /**< boolean: local (0) or ghost (1) */
      p4est_quadrant_t   *quad;        /**< the actual quadrant */
      p4est_locidx_t      quadid;      /**< index in tree or ghost array */
    }
    full;
    struct
    {
      int8_t              is_ghost[2]; /**< boolean: local (0) or ghost (1) */
      p4est_quadrant_t   *quad[2];     /**< the actual quadrant */
      p4est_locidx_t      quadid[2];   /**< index in tree or ghost array */
    }
    hanging;
  }
  is;
}
```

The user can thus exchange ghost values in parallel and then call `p4est_iterate` and compute all necessary updates for the next time step without further communication.

While this program may appear somewhat daunting at first sight, we like to point out that `p4est` is not a solver library and thus does not implement any numerical tasks. This program basically supplies the numerical part and leverages over 30,000 lines of `p4est` parallel AMR logic.

4.4 A minimal finite element solver

As a final example we present an adaptive finite element solver for the Poisson equation on the unit square; see `example/steps/p4est_step4`. We focus on processing hanging nodes and the parallel scatter operation in the matrix vector product. We do not implement any preconditioner here, which would of course be recommended to achieve a mesh-independent iteration count. Given the node numbering in `p4est_lnodes`, it is in fact fairly straightforward to interface to a parallel sparse linear algebra library.

This example is truly minimal and works at the lowest possible level of encoding hanging nodes, relying on the `p4est` numbering conventions [9]. Application developers may choose to encapsulate this logic into their own numbering and access schemes to limit the amount of code that depends on `p4est` data structures.—

We begin by refining the mesh non-recursively, which is recommended for good load balance. The 2:1 balance function is called after the loop below.

```
  for (level = startlevel; level < endlevel; ++level) {
    p4est_refine (p4est, 0, refine_fn, NULL);
    /* Refinement has lead to up to 4x/8x more elements; redistribute them. */
    p4est_partition (p4est, 0, NULL);
  }
```

Next we classify the finite element nodes, which lie at the corners of an element for a piecewise bi-/trilinear discretization. For the unit square or cube, it is easy to determine the domain boundary; for more complex geometries we would examine the tree connectivity to check whether a tree touches the domain boundary or another tree.

```
static int
is_boundary_unitsquare (p4est_t * p4est, p4est_topidx_t tt,
                        p4est_quadrant_t * node)
{
  /* For this simple connectivity it is sufficient to check x, y (and z). */
  return (node->x == 0 || node->x == P4EST_ROOT_LEN ||
          node->y == 0 || node->y == P4EST_ROOT_LEN ||
#ifdef P4_TO_P8
          node->z == 0 || node->z == P4EST_ROOT_LEN ||
#endif
          0);
}
```

For any given element we can access information about the hanging status of its boundary entities; to this end we use a dedicated decoding function. It reads the `face_code`, which is available in the `p4est_lnodes` structure and consists of one integer per element, where the lowest $D =$ `P4EST_DIM` bits hold the child number of the element relative to its parent, the next D bits the status of the potentially hanging faces, and in 3D another D bits the status of the potentially hanging edges.

```
      /* We need to determine whether any node on this element q is hanging. */
      anyhang = lnodes_decode2 (lnodes->face_code[q], hanging_corner);
```

The element nodes are enumerated in `p4est_lnodes`. We do not store any node coordinates and leave it to the application to compute them when needed. Care must be taken to use the node from the parent if one element meets a larger neighbor.

```
      for (i = 0; i < P4EST_CHILDREN; ++i) {    /* Loop over 2**D corners. */
        lni = lnodes->element_nodes[P4EST_CHILDREN * k + i];   /* Element index k. */
        if (bc[lni] < 0) {
          if (anyhang && hanging_corner[i] >= 0) {
            /* This node is hanging; access the referenced node instead. */
            p4est_quadrant_corner_node (parent, i, &node);
          }
          else {
            p4est_quadrant_corner_node (quad, i, &node);
          }

          /* Determine boundary status of independent node. */
          bc[lni] = is_boundary_unitsquare (p4est, tt, &node);

          /* Transform per-tree reference coordinates into physical space. */
          p4est_qcoord_to_vertex (p4est->connectivity, tt,
                                  node.x, node.y, vxyz);

          /* Use physical space coordinates to evaluate functions */
          rhs[lni] = func_rhs (vxyz);
          uexact[lni] = func_uexact (vxyz);
        }
      }
```

The matrix vector product is implemented as a loop over the processor-local elements. If the element has hanging nodes, we interpolate them from the independent degrees of freedom, multiply with the element matrix, and then apply the transpose of the interpolation. The interpolation function exploits the `face_code` in the same way as `lnodes_decode2` above. We list the version for piecewise D-linear elements.

```
  const int           c = (int) (face_code & ones);     /* ones = 2**D - 1 */
  int                 work = (int) (face_code >> P4EST_DIM);

  /* Compute face hanging nodes first (this is all there is in 2D). */
  for (i = 0; i < P4EST_DIM; ++i) {
    if (work & 1) {
      ef = p4est_corner_faces[c][i];
      sum = 0.;
      for (j = 0; j < P4EST_HALF; ++j) {
        sum += inplace[p4est_face_corners[ef][j]];
      }
      inplace[c ^ ones ^ (1 << i)] = sum / P4EST_HALF;  /* divide by 2**(D-1) */
    }
    work >>= 1;
  }

#ifdef P4_TO_P8
  /* Compute edge hanging nodes afterwards */
  for (i = 0; i < P4EST_DIM; ++i) {
    if (work & 1) {
      ef = p8est_corner_edges[c][i];
      inplace[c ^ (1 << i)] = .5 * (inplace[p8est_edge_corners[ef][0]] +
                                    inplace[p8est_edge_corners[ef][1]]);
    }
    work >>= 1;
  }
#endif
```

Arbitrary-order tensor-product elements are implemented in much the same way; we would use matrices for polynomial interpolation on both hanging faces and edges. Below is the hanging-node matrix vector product and transpose interpolation using predefined hanging-node lookup tables.

```
        for (i = 0; i < P4EST_CHILDREN; ++i) {  /* Loop over corners. */
          sum = 0.;
          for (j = 0; j < P4EST_CHILDREN; ++j) {
            sum += (*matrix)[i][j] * inloc[j];  /* Element matvec. */
          }
          if (hanging_corner[i] == -1) {
            /* This node is not hanging, there is no need to transform. */
            c = 0;
            ncontrib = 1;
            contrib_corner = &i;
            sum *= factor;      /* Geometry scaling on the unit cube. */
          }
          else {
```

```
      /* This node is hanging.  Use hanging node relations from the
       * reference quadrant by transforming corner numbers with ^ c. */
      c = hanging_corner[i];      /* Child number of quadrant. */
      ncontrib = corner_num_hanging[i ^ c];
      contrib_corner = corner_to_hanging[i ^ c];
      sum *= factor / (double) ncontrib;
    }
    /* The result is added into the output vector. */
    for (j = 0; j < ncontrib; ++j) {
      h = contrib_corner[j] ^ c;  /* Inverse transform of node number. */
      if (!isboundary[h]) {
        out[all_lni[h]] += sum;   /* Sum into node vector. */
      }
    }
  }
```

It remains to scatter and sum the contributions for all processors that share access to a node. To this end we use the function `share_sum`, which is short since it leverages the `p4est_lnodes_share_all` routine that we have implemented specifically for this purpose. It works for any polynomial order.

5 Further reading

5.1 Technical papers on `p4est`

This document is linked on the official `p4est` web page [5], which also contains links to the source code and examples, generated documentation, and the detailed technical papers summarized below.

The technical paper [9] describes the logic for connecting neighboring octrees, determining the coordinate transformations between them, and several high-level algorithms such as `p4est_refine`, `p4est_coarsen`, `p4est_balance`, `p4est_partition`, and the original `p4est_ghost` version. It contains scalability results on over 220k cores of the *Jaguar* supercomputer. The technical paper [12] (submitted) contains an updated `p4est_ghost` version and algorithms for iterating through the mesh topology and numbering finite element degrees of freedom in parallel. It contains scalability results on over 458k cores of the *JUQUEEN* supercomputer. Togother, these two papers constitute the algorithms reference for `p4est`. In addition, [11] contains an in-depth discussion and demonstration of our optimized parallel 2:1 balance algorithm.

5.2 Projects using `p4est`

Our initial motivation developing `p4est` was to enable scalable adaptive simulations for earth's mantle convection, specifically in supporting development of the Rhea code [1, 2, 8, 15]. `p4est` has been a separate module from the beginning to allow for easy interfacing with other applications. The authors are using it for simulating e.g. seismic wave propagation [17], ice sheet dynamics [13], atmospheric flow [6], and to investigate techniques such as parallel geometric/algebraic multigrid [16]. All of these applications scale to between 10^3 and 10^5 CPUs; the biggest mesh we have created so far consists of roughly 513×10^9 elements (demonstrated on both *Jaguar* and *JUQUEEN*).

We have interfaced the `deal.II` software for generic adaptive finite element simulations to `p4est`, thus removing a scalability bottleneck that goes back to the design of `deal.II`'s original replicated mesh storage [3, 4]. A prominent example of using `p4est` through `deal.II` is the mantle convection code `ASPECT` [14].

We are contacted on a regular basis by researchers working with `p4est`. Please email your comments/questions/ideas for contribution to the mailing list `p4est@librelist.com`.

6 Acknowledgements

We would like to thank Omar Ghattas for his continued support. We gratefully acknowledge Johann Rudi, Johannes Holke, Ethan Alan Hereth, Alex Fikl, and Hannes Frank for their contributions to the software.

The development of `p4est` was partially supported by the US National Science Foundation (NSF Grants No. OCI-0749334, CCF-0427985, CNS-0540372, CNS-0619838, DMS-0724746, OPP-0941678) and the US Department of Energy (DOE Grants No. 06ER25782, 08ER25860, SC0002710). The authors thank the Texas Advanced Computing Center (TACC) for providing them with access to the Ranger supercomputer under NSF TeraGrid award MCA04N026, and the National Center for Computational Science (NCCS) for early-user access to the Jaguar Cray XT5 supercomputer. Any opinions, findings and conclusions or recomendations expressed in this paper, the source code, or the documentation are those of the authors and do not necessarily reflect the views of the US National Science Foundation (NSF).

7 References

[1] L. Alisic, M. Gurnis, G. Stadler, C. Burstedde, and O. Ghattas, *Multi-scale dynamics and rheology of mantle flow with plates*, Journal of Geophysical Research, 117 (2012), p. B10402.

[2] L. Alisic, M. Gurnis, G. Stadler, C. Burstedde, L. C. Wilcox, and O. Ghattas, *Slab stress and strain rate as constraints on global mantle flow*, Geophysical Research Letters, 37 (2010), p. L22308.

[3] 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.

[4] W. Bangerth, R. Hartmann, and G. Kanschat, `deal.II` *Differential Equations Analysis Library, Technical Reference*, Interdisziplinäres Zentrum für Wissenschaftliches Rechnen (IWR). `http://www.dealii.org`.

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

[6] C. Burstedde, D. Calhoun, K. Mandli, and A. R. Terrel, *ForestClaw: Hybrid forest-of-octrees AMR for hyperbolic conservation laws*, in Proceedings of ParCo '13, August 2013. `http://arxiv.org/abs/1308.1472`.

[7] 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.

[8] C. Burstedde, G. Stadler, L. Alisic, L. C. Wilcox, E. Tan, M. Gurnis, and O. Ghattas, *Large-scale adaptive mantle convection simulation*, Geophysical Journal International, 192 (2013), pp. 889–906.

[9] 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.

[10] J.-L. Gailly and M. Adler, *A massively spiffy yet delicately unobtrusive compression library*. `http://zlib.net/`.

[11] T. Isaac, C. Burstedde, and O. Ghattas, *Low-cost parallel algorithms for 2:1 octree balance*, in Proceedings of the 26th IEEE International Parallel & Distributed Processing Symposium, IEEE, 2012.

[12] T. Isaac, C. Burstedde, L. C. Wilcox, and O. Ghattas, *Recursive algorithms for distributed forests of octrees*. In preparation, 2014.

[13] T. Isaac, G. Stadler, and O. Ghattas, *Solution of nonlinear Stokes equations discretized by high-order finite elements on nonconforming and anisotropic meshes, with application to ice sheet dynamics*, SIAM Journal on Scientific Computing (submitted), (2014).

[14] M. Kronbichler, T. Heister, and W. Bangerth, *High accuracy mantle convection simulation through modern numerical methods*, Geophysical Journal International, 191 (2012), pp. 12–29.

[15] G. Stadler, M. Gurnis, C. Burstedde, L. C. Wilcox, L. Alisic, and O. Ghattas, *The dynamics of plate tectonics and mantle flow: From local to global scales*, Science, 329 (2010), pp. 1033–1038.

[16] H. Sundar, G. Biros, C. Burstedde, J. Rudi, O. Ghattas, and G. Stadler, *Parallel geometric-algebraic multigrid on unstructured forests of octrees*, in SC12: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, ACM/IEEE, 2012.

[17] L. C. Wilcox, G. Stadler, C. Burstedde, and O. Ghattas, *A high-order discontinuous Galerkin method for wave propagation through coupled elastic-acoustic media*, Journal of Computational Physics, 229 (2010), pp. 9373–9396.