User manual for `casl_p4est`

arthur guittet

April 5, 2016

Contents

1 Installation **1**
- 1.1 Access to Stampede and Comet 1
- 1.2 Installing `p4est` 2
- 1.3 Installing `casl_p4est` 3
- 1.4 Compiling against the `casl_p4est` library 3

2 Using the `p4est` library **3**
- 2.1 Forests of Octrees and parallelization 3
- 2.2 Organization of the quadrants 4
- 2.3 The ghost quadrants data structure 5
- 2.4 The nodes data structure 6
- 2.5 Communications with `Petsc` 7
- 2.6 Finite differences and local tree reconstruction 7

1 Installation

1.1 Access to Stampede and Comet

The first step is to create an account on the XSEDE website (www.xsede.org). After that, the account can be added to an active XSEDE grant by the corresponding PI and the resources available through that allocation can be activated. They need to be activated one by one, for instance if both Stampede and Comet are part of the allocation, each of them needs to be activated separately for the specific user.

Once the access is granted, the login nodes of the clusters can be accessed through ssh with the following commands,

```
ssh username@stampede.tacc.utexas.edu
ssh username@comet.sdsc.edu
```

The password is the XSEDE password. Note that when updating the password on XSEDE, several minutes can be required for the update to be propagated to the supercomputers. For ease of access, the following information about an ssh server can be added in `.ssh/config` (for unix systems)

```
Host stampede
Hostname stampede.tacc.utexas.edu
User username
```

A different name can be chosen for the `Host` part. The supercomputer can then be accessed with

```
ssh stampede
```

To transfer files from and to a supercomputer, use the "scp" or "rsync" commands for small file, and "globus" for big transfers between supercomputers (checkout www.globus.org). The documentation for Stampede can be found online on the Texas Advanced Computing Center website, at https://portal.tacc.utexas.edu/user-guides/stampede, and is very well written. The documentation for Comet is not as good ... but they work very similarly.

1.2 Installing p4est

The p4est library can be installed from a tarball or directly from the git repository. Some bugs have been fixed and numerous features added since the 1.1 version, and using the git repository is recommended. Start by cloning the repository with

```
git clone https://github.com/cburstedde/p4est.git path_to_local_git_folder
```

In the folder created (replace "path_to_local_git_folder" with you choice), run

```
git submodule init && git submodule update
```

to initialize the sc submodule on which p4est depends. A memory alignment bug was fixed recently, and the branch with the corresponding fix must be selected by

```
git checkout fix-memalign
```

in both the main git folder and the "sc" sub-folder. You then need to run

```
./bootstrap
```

from the git folder, followed by the configure command with your choice of option. For a release version, use

```
./configure --prefix=/path/to/install/folder --enable-mpi --without-blas
--enable-shared --enable-memalign=16 CFLAGS=-O2 CPPFLAGS=-O2 FCFLAGS=-O2
```

If building for debug, the run

```
./configure --prefix=/path/to/install/folder --enable-mpi --without-blas
--enable-shared --enable-debug --enable-memalign=16 CFLAGS="-O0 -g"
CPPFLAGS="-O0 -g" FCFLAGS="-O0 -g"
```

Note that you must have mpi installed and in the PATH. The default install folder (it should be something like "/usr/lib/p4est") can also be selected by omitting the "prefix" option. The p4est library can no be built and installed by running

```
./make && ./make install
```

The casl_p4est library also requires Petsc, which is available on the public repositories if running Linux Mint or Ubuntu. Otherwise, you can install it by following the instructions from the Petsc website (https://www.mcs.anl.gov/petsc/). Here are the flags I use if the package is not available on a public repository

```
./configure --download-fblaslapack --download-hypre=1
--prefix=/path/to/install/dir --with-debugging=0 --with-mpi-dir=/path/to/mpi/dir
--with-shared-libraries=1 COPTFLAGS="-O2" CXXOPTFLAGS="-O2" FOPTFLAGS="-O2"
```

On Stampede and Comet, you can load the `Petsc` module with the module manager, by running

```
module load petsc
```

The `p4est` library is also available as a module on Stampede, however it's the 1.1 version that does not have some of the new features and bug fixes that are needed by `casl_p4est`.

1.3 Installing `casl_p4est`

The first step is to create a bitbucket account (at bitbucket.org) and ask someone with admin rights to add you to the git repository. Once this is done, you can clone the repository by running

```
git clone https://username@bitbucket.org/cburstedde/casl_p4est.git local_folder
```

where "username" is your bitbucket username and "local_folder" is the path to the folder where the library is to be installed. You can use git through the command line or with a Graphical User Interface. I personally recommend SmartGit.

1.4 Compiling against the `casl_p4est` library

A sample `Makefile` and a sample `project.pro` file are located in the "doc/casl_p4est_manual" folder. The general things you need in you `Makefile` are

- include the headers for the `p4est` library and path to the compiled library for the linker

  ```
  INCPATH += -I/path/to/the/p4est/install/include
  LIBS += -Wl,-rpath,/path/to/the/p4est/install/lib
          -L/path/to/the/p4est/install/lib -lp4est -lsc
  ```

- include the headers for the `Petsc` library and path to the compiled library for the linker. Depending on the system, the PETSC_DIR might already be set as an environment variable, it is for instance the case on Stampede.

  ```
  INCPATH += -I/path/to/the/petsc/install/include
  LIBS += -Wl,-rpath,/path/to/the/petsc/install/lib
          -L/path/to/the/petsc/install/lib -lpetsc
  ```

- include the headers for the `Petsc` library and path to the compiled library for the linker. Depending on the system, the PETSC_DIR might already be set as an environment variable, it is for instance the case on Stampede.

  ```
  INCPATH += -I/path/to/the/casl/p4est/library
  ```

2 Using the `p4est` library

The following sections contain information about the `p4est` data structures and how to use them, including samples of code.

2.1 Forests of Octrees and parallelization

The `p4est`library works with forests of Octrees, that is a collection of Octrees rooted in a shared macromesh. The macromesh can be complicated, but for the `casl_p4est`we limit ourselves to Cartesian macromeshes. This can be changed in the future if needed. The first step is therefore to declare a macromesh, with

```
my_p4est_brick_t brick;
p4est_connectivity_t *connectivity = my_p4est_brick_new(nx, ny, nz,
                                 xm, xM, ym, yM, zm, zM, &brick, px, py, pz);
```

with `nx`, `ny` and `nz` are the dimensions of the macromesh. For instance, setting those to 2 will generate a 2x2x2 macromesh, i.e. the framework for a forest of 8 Octrees. The other parameters are the physical dimensions of the domain and the periodicity information. Every process knows about the macromesh, it is a shared information.

The forest can now be created and the Octrees refined, for example with a level-set function `LS` defined as a CF_3 (Continuous Function in $\mathbb{R}^3$, or CF_2 in 2d), with

```
p4est_t *p4est = my_p4est_new(mpicomm, connectivity, 0, NULL, NULL);
splitting_criteria_cf_t criteria(lmin, lmax, &LS, 1.2);
p4est->user_pointer = (void*)(&criteria);
for(int lvl=0; lvl<lmax; ++lvl)
{
        my_p4est_refine(p4est, P4EST_FALSE, refine_levelset_cf, NULL);
        my_p4est_partition(p4est, P4EST_FALSE, NULL);
}
```

with `mpicomm` the mpi communicator to use, `lmin` and `lmax` the min and max level of each Octree, and `1.2` the Lipschitz constant for the level-set function (choosen conservatively to be larger than 1 here). The refinement is done within a loop because initially the macromesh is distributed across the processes evenly. This means that a 1x1x1 macromesh will be attributed to a single process, even if running 1024 mpi tasks. Conversely, a 2x2x2 macromesh with 4 processes will result in 2 blocks per process.

The forest is partitioned across processes by linearizing the trees, i.e. building an array of the leaves ordered by a particular procedure based on a space-filling curve. `p4est` uses the Z-ordering, or Morton ordering. Though alternatives exist (Hilbert curve for instance), their performances are similar to the Z-curve. The array is then distributed evenly across the processes. Figure 1 illustrates the process.

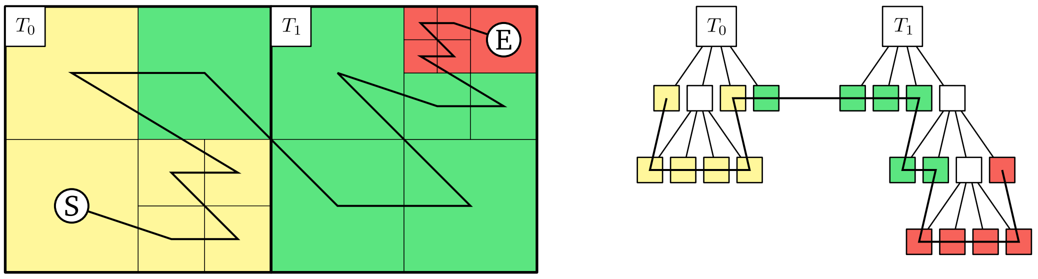

Figure 1: Illustration of the Z-curve ordering and of the partitioning process for a forest of 2 Octrees (macromesh 2x1) and for 3 mpi processes.

2.2 Organization of the quadrants

The forest is now ready for use if the algorithm is based on cell-centered data. Each process knows about its local chunk of the quadrants array, stored inside the local tree structures. The number of quadrants owned by a process is given by

```
p4est->local_num_quadrants
```

while the entire forest contains

```
p4est->global_num_quadrants
```

Each chunk of the local quadrants belongs to a tree, and thanks to the Z-curve property the values are contiguous. Each tree stores the index at which its quadrants start in the local ordering in

```
tree->quadrants_offset
```

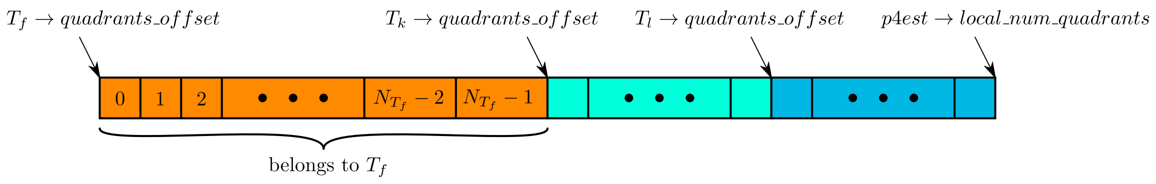
$T_f \rightarrow quadrants\_offset$
$T_k \rightarrow quadrants\_offset$
$T_l \rightarrow quadrants\_offset$
$p4est \rightarrow local\_num\_quadrants$
0
1
2
$N_{T_f}-2$
$N_{T_f}-1$
belongs to T_f

Figure 2: Representation of the quadrants data structure. T_f is the first local tree for the current mpi process, T_l is the last local tree, and N_{T_f} is the number of local quadrants located in tree T_f, i.e. $T_f \rightarrow$ `quadrants` $\rightarrow$ `elem_count`.

Figure 2 summarizes the `quadrants` structure, and the code structure for accessing the quadrants information is

```
for(p4est_topidx_t tree_idx=p4est->first_local_tree;
        tree_idx<=p4est->last_local_tree;
        ++tree_idx)
{
  p4est_tree_t *tree = (p4est_tree_t*)sc_array_index(p4est->trees, tree_idx);
  for(size_t q=0; q<tree->quadrants.elem_count; ++q)
  {
    p4est_locidx_t quad_idx = q+tree->quadrants_offset;
    p4est_quadrant_t *quad =
                    (p4est_quadrant_t*)sc_array_index(&tree->quadrants, q);
  }
}
```

Here `quad_idx` is the index of the quadrant in the local ordering, and `q` is the index of the quadrant in the local tree. The global index of a local quadrant can be obtained by adding the global offset of its owning process to its index

```
p4est_gloidx_t gloidx = quad_idx + p4est->global_first_quadrant[p4est->mpirank];
```

2.3 The ghost quadrants data structure

So far, each process only knows about its local information. The `p4est_ghost_t` gives access to a layer of ghost cells around a process. The structure can be initialized with

```
p4est_ghost_t *ghost = my_p4est_ghost_new(p4est, P4EST_CONNECT_FULL);
```

`P4EST_CONNECT_FULL` indicates that the ghost neighbors are gathered across faces, edges and corners. This constructs the structure for a ghost layer of depth one. The ghost layer can be expanded with

```
my_p4est_ghost_expand(p4est, ghost);
```

This results in a ghost layer of depth two. The expand function can be called the desired number of times to build a ghost layer of larger depth.

The ghost data structures contains the array of quadrants in the ghost layer, stored in

```
ghost->ghosts
```

and of size

```
ghost->ghosts.elem_count
```

One can loop over the ghost directly with

```
for (g = 0; g < ghost->ghosts.elem_count; ++g)
{
      q = (p4est_quadrant_t*)sc_array_index(&ghost->ghosts, g);
}
```

The ghost structure also contains the information about the owner of a ghost quadrant and the tree it belongs to. Figure 3 illustrate the structure of this information.

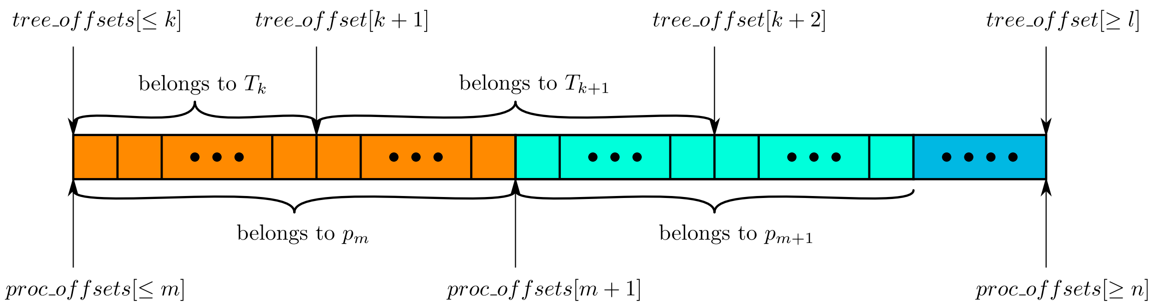

Figure 3: Representation of the ghost layer information. The `tree_offsets` arrays and `proc_offsets` are members of the `ghost_t` data structure. T_k corresponds to the tree of index k and p_m to mpi process of index m. For this particular example, the trees of index strictly smaller than k or larger or equal to l are not represented in the ghost layer, and similarly none of the quadrants in the ghost layer belong to a mpi processes with a rank strictly smaller than m or larger or equal to n.

2.4 The nodes data structure

If working on node (i.e. vertices) based data, the nodes structure must be initialized. It contains the basic information about the nodes, such as the index of the nodes and the remote information for ghost layer nodes. The nodes structure is initialized by calling

```
p4est_nodes_t *nodes = my_p4est_nodes_new(p4est, ghost);
```

The ghost information is optional and can be replaced by `NULL` if the ghost layer is not used. In the original `p4est` library, the nodes are organized differently than in `casl_p4est`. That is, the ghost nodes for lower ranks are located at the beginning of the nodes array, the ghost nodes for higher ranks are at the end, and the local nodes are in between, starting at `nodes`→`offset_owned_indeps`. Furthermore, the hanging nodes are indexed separately.

In `casl_p4est`, the nodes are reorganized to match the quadrants organization, thus simplifying the coupling with `Petsc`.

2.5 Communications with Petsc

Most of the communication can be handled with the Vec data type provided by Petsc. The Petsc library is thoroughly documented online. A Vec is an array that can have a ghost chunk. A ghosted array can be created with the data located at the quadrants' center or at the nodes. The nodes code is

```
Vec phi;
VecCreateGhostNodes(p4est, nodes, &phi);
```

The actual data stored in the Vec can be accessed by

```
double *phi;
VecGetArray(phi, &phi_p);
for (p4est_locidx_t n = 0; n<nodes->num_owned_indeps; ++n)
{
  (do something with phi_p)
}
VecRestoreArray(phi, &phi_p);
```

In the previous code, the pointer phi_p points to a contiguous array of the local part of the data stored in the Vec. Petsc does not guarantee that the local part and the ghosted part are stored in the same location, and therefore accessing the entire data - including the ghosted part - should be done after getting the local form of the Vec

```
Vec phi_loc;
double *phi_p;
VecGhostGetLocalForm(phi, &phi_loc);
VecGetArray(phi_loc, &phi_p);
for(size_t n=0; n<nodes->indep_nodes.elem_count; ++n)
{
  (do something with phi_p)
}
VecRestoreArray(phi_loc, &phi_p);
VecGhostRestoreLocalForm(phi, &phi_loc);
```

In many places in the library, the ghosted values are accessed without first accessing the local form of the Vec, this is a dangerous practice that has worked ... so far.

The ghost values can be synchronized with the remote processes through the Petsc API, thus leading to the general communication layout

```
{ do work for local values shared with other processes }
VecGhostUpdateBegin(phi, INSERT_VALUES, SCATTER_FORWARD);
{ do work for local values not shared with other processes }
VecGhostUpdateEnd(phi, INSERT_VALUES, SCATTER_FORWARD);
```

2.6 Finite differences and local tree reconstruction