The Omega_h Users Manual

Dan Ibanez
Sandia National Laboratories
daibane@sandia.gov

April 11, 2022

1 Purpose

The purpose of Omega_h is to adapt triangle and tetrahedral meshes to conform to a given metric field, effectively using combinations of MPI and shared-memory parallelism, and providing good results as defined by element quality and edge length measures.

2 Obtaining

Omega_h is developed and distributed via GitHub, a popular software hosting platform based on the Git distributed version control system.

`https://github.com/sandialabs/omega_h`

The most common way to obtain Omega_h is to use Git to clone the repository and automatically check out the `master` branch:

```
git clone git@github.com:sandialabs/omega_h.git
```

3 Compiling

3.1 Operating System

Omega_h currently only supports POSIX-like operating systems. It has only been tested on Linux, Mac OS X, and certain POSIX-like supercomputer kernels.

3.2 CMake

Omega_h's compilation process is controlled by the CMake build system. In order to build Omega_h, one should install a recent version of CMake, with version 3.18.0 being the minimum acceptable version. CMake is available in most Linux package managers or from their website:

`https://cmake.org/download/`

It is also recommended that users adopt the CMake build system for their own project, in order to take advantage of the metadata that Omega_h outputs when it compiles and installs, which is readable by other CMake projects.

3.3 Compiler

Omega_h is written in the C++14 standard of C++, so a compiler with complete support for that standard is needed. Omega_h's CMake files will accept the GCC, Clang, and Intel compilers, and support can be added fairly easily for other compilers upon request.

3.4 Dependencies

All dependencies of Omega_h are optional, meaning that one can compile it by itself and obtain a fairly functional code for mesh adaptation, although it will not have parallel features yet. Optional dependencies of Omega_h are:

1. Zlib: This widely used and installed C library implements efficient data compression algorithms. Omega_h uses it compress its own `.osh` file format and VTK's `.vtu` files.

2. MPI: The Message Passing Interface is a standard defining (at least) a C library that enables multi-process parallelism. This is required if you want to use multi-process parallelism in Omega_h. The two good open-source implementations that Omega_h is known to work with are MPICH and OpenMPI, and we recommend MPICH for its support of the latest MPI standard and its cleaner memory management. In order to use this, we typically just use the MPI compiler wrapper `mpicxx` as the `CMAKE_CXX_COMPILER`.

3. Kokkos: This C++14 library implements shared-memory parallelism constructs and allows Omega_h to (mostly) not worry about the details of OpenMP and CUDA. It is required if you want to use shared-memory parallelism in Omega_h. Enabling Kokkos provides two more options, `Omega_h_ENABLE_CUDA` and `Omega_h_ENABLE_OpenMP`, which are the main on-node parallelism backends that Kokkos uses for GPUs and multi-core CPUs, respectively.

4. libMeshb: This C library implements the '.mesh' and '.meshb' file formats used by INRIA, NASA, and others. It is required to read and write '.meshb' files from Omega_h. Note that currently Omega_h follows a particular convention in what those files are expected to contain, namely elements, vertices, and sides on the boundary.

5. EGADS: This C API wraps over OpenCASCADE in a human-manageable way. It is required if you want Omega_h to snap new vertices to geome-

try. Note that classification in Omega_h should match the numbering of geometric entities in EGADS.

Each of these dependencies is first controlled by a CMake boolean, for example the dependence on Kokkos is controlled by the boolean `Omega_h_ENABLE_Kokkos`. Dependencies which require finding libraries and/or headers have an associated prefix path where those libraries and headers are installed, for example `Kokkos_PREFIX` indicates the location where Kokkos was installed. So to use Kokkos one may give the following two arguments to the `cmake` command-line program:

```
-DOmega_h_ENABLE_Kokkos:BOOL=ON \
-DKokkos_PREFIX:PATH=$HOME/install/kokkos \
```

3.5 Other Options

4 Usage

4.1 File Format

Omega_h implements I/O functionality for a custom mesh format. The format stores parallel meshes using one file per partition, and stores them all in a single directory which is typically named with the `.osh` extension to indicate that it is conceptually an Omega_h "file". The directory may look something like this:

```
cube.osh/
cube.osh/nparts
cube.osh/version
cube.osh/0.osh
cube.osh/1.osh
cube.osh/2.osh
cube.osh/3.osh
```

The file `cube.osh/2.osh` encodes, in a binary format, the mesh partition stored in MPI rank 2. As will be described in Section 5.2, Omega_h stores most information into arrays. In order to write this information to file, byte swapping is first applied to the entries as a way to ensure that information remains consistent between little endian and big endian computers, and then the array is compressed with Zlib (assuming the user has compiled Omega_h with Zlib). These two steps ensure the file is both compact and lossless, i.e. the mesh data when loaded from file will be bitwise identical to the data that was written to file.

This format is a convenient way for users to manage meshes and attached information, and to build an ecosystem of modular command-line tools which act on said meshes.

It is recommended that users only parse this format using the APIs provided by the Omega_h library.

We do try to ensure the format is backwards-compatible, i.e. a file tree written by one version of Omega_h should remain readable by all subsequent versions of Omega_h.

4.2 Utility Programs

For some basic operations, mostly involving file format conversions, Omega_h will compile command-line programs that perform these operations.

1. `msh2osh`: Convert from Gmsh's native text format to the Omega_h binary format.

2. `osh2vtk`: Convert from the Omega_h binary format (tree) to a tree as follows:

   ```
   given-name/
   given-name/pieces.pvtu
   given-name/pieces/
   given-name/pieces/piece_0.vtu
   given-name/pieces/piece_1.vtu
   ```

   Where `given-name` is the second argument provided to the program. Each `.vtu` file represents one parallel partition, and opening the `.pvtu` in Paraview (or another viewer) will render the entire parallel mesh.

3. `meshb2osh`, `osh2meshb`: these two programs become available when the `libMeshb` dependency is enabled, and convert back and forth between the Gamma mesh format (sometimes used by INRIA and NASA for CFD applications) and (serial) Omega_h file trees.

4. `oshdiff`: Modeled after the `exodiff` program from the Exodus II project, this program compares two Omega_h mesh files with configurable floating-point tolerances.

5. `vtkdiff`: Modeled after the `exodiff` program from the Exodus II project, this program compares two VTK file trees with configurable floating-point tolerances. The trees must have been created by Omega_h in the first place, due to limitations in Omega_h's VTK format reader.

6. `osh_box`: Generates a structured simplex mesh of a parallelepiped domain, by first constructing a quadrilateral or hexahedral grid and then subdividing said grid into triangles or tetrahedra. The parameters are the extents of the box and the number of elements that span the length of the box along each axis.

7. `osh_part`: Partitions an Omega_h mesh. The input is a `.osh` file tree with a certain number of parts, and the output is a `.osh` file tree with a different number of parts. An increase in the number of parts must

multiply the number of parts by a power of two, due to limitations of Omega_h's Recursive Inertial Bisection implementation. The number of parts may also be decreased, in which case certain consecutive parts will be merged.

8. `osh_scale`: Adapts a mesh such that it retains approximately the same variation of resolution (metric gradient) and upon output contains approximately the desired number of elements. This can be convenient for convergence studies, to generate a series of meshes which have prescribed element counts and whose resolution variation over the spatial domain mirrors the variation in resolution of the initial starting mesh.

4.3 Header and Library

In most cases, users should call the C++14 API of Omega_h directly from their own C++ code. In terms of the build system, users need their code to include the `Omega_h.hpp` header file and link their own code to the `libomega_h.so` or `libomeg_h.a` library. In the case of static linking (`libomega_h.a`), it is also necessary to link to the final executable to all libraries that Omega_h depends on. In the case of dynamic linking, `libomega_h.so` is guaranteed to already be linked to its dependencies via the `RPATH` mechanism, so only `libomega_h.so` needs to be linked to the user application.

4.4 Via CMake

The most convenient way to handle the header file inclusion and library linking to Omega_h is to use the CMake build system for the user's project as well. Omega_h uses CMake to export file

`${CMAKE_INSTALL_PREFIX}/lib/cmake/Omega_hConfig.cmake`

which provides all the necessary include and link information to CMake when CMake is configuring the user's project. The way for the user's project to read that file is via CMake's `find_package` command:

```
find_package(Omega_h 8.2.0 PATHS ${OMEGA_H_PREFIX})
```

Only the first argument ("Omega_h") is really necessary. The second argument specifies a minimum required version number. CMake will only accept an Omega_h installation that has the same or higher version number, unless the major version number is higher, because that indicates a change that breaks backwards compatibility. The last two arguments can be used to indicate that Omega_h is installed somewhere other than the standard system directories (e.g. `/usr/local`). If `find_package` succeeds, subsequent CMake code has access to a target called `Omega_h::omega_h`, which represents all the information needed to link to Omega_h, and is used via `target_link_libraries` with a user's library or executable target, like so:

```
add_executable(my_simulation_code driver.cpp algebra.cpp)
target_link_libraries(my_simulation_code Omega_h::omega_h)
```

4.5 The Omega_h Namespace

As a C++ library, Omega_h has a responsibility not to pollute the user's global namespace, whether it is the C++ namespace or the preprocessor namespace. As such, all its public C++ symbols will be contained in a C++ namespace called `Omega_h`, and all preprocessor symbols that it publicly defines have names beginning with `OMEGA_H`. A relevant C++ language feature is the ability to alias namespaces, i.e. users could do the following:

```
namespace osh = Omega_h;
```

and in subsequent code Omega_h symbols would be available in the namespace `osh`.

5 Public Symbols

5.1 Scalar Types

Omega_h builds all its data structures out of arrays of basic scalar types. Over time it seems that a set of four basic types is sufficient to form efficient data structures for all its needs: one floating-point type (`double`), and three integers types of varying width (`int8_t`, `int32_t`, and `int64_t`). Omega_h names these types `Real`, `I8`, `LO` (a.k.a. `I32`), and `GO` (a.k.a. `I64`). `LO` and `GO` stand for local and global ordinals, terms borrowed from the Trilinos project. There is also `Int`, which is meant for local small integer values, not as the entry type for a large array.

5.2 Read and Write: The Array Classes

The vast majority of Omega_h data is stored in contiguous, dynamically allocated arrays. These arrays are implemented as four templated classes, defined by two binary properties.

First, Omega_h gains certain benefits of functional programming related to "const-correctness" by having separate types for arrays which are modifiable and those which are not. All data starts as a modifiable array ("Write"), and as soon as possible it is converted to a read-only object ("Read"). Ideally, there is only one modifiable array in existence at any particular time: the array being created by the currently executing code. One of the more interesting benefits of this is the ability to use shallow copies of data with confidence via the Copy-On-Write convention, for example two mesh objects may share the same coordinates array with confidence that if one mesh decides to change its coordinates, the other mesh will not be affected.

Second, Omega_h is designed for efficient use of GPUs, which have memory that is separate from the memory attached to the CPU. At the time of this writing, explicit control of where memory resides and when to transfer it from the GPU to the CPU is the most reliable way to get good performance on GPU-accelerated systems. In addition, due to the high cost of memory transfer, the

best performance is achieved by keeping data as much as possible on the GPU and not moving it to the CPU unless absolutely necessary. Hence, each array type includes "Host" classes, which represent a copy of the array on the CPU. These variants are the minority, most arrays in Omega_h are in GPU memory when it is compiled with CUDA support.

In total, the four classes are `Read`, `Write`, `HostRead`, and `HostWrite`. Each of these four classes, in turn, is templated on a scalar type, and explicitly instantiated for each of the four scalar types described in Section 5.1, so for example `Read<Real>`. Finally, some convenience types are defined, for example `Reals` is an alias for `Read<Real>`. Likewise, `LOs` is an alias for `Read<LO>`.

Consistent with our philosophy of keeping data as much on the GPU as possible, and taking advantage of the benefits of immutable data structures, the majority of data in Omega_h is stored as a `Read`, after being created as a `Write`. In the rare occasions when data is read or written to file, or transmitted across a CPU-connected network interface (e.g. MPI), then the `HostRead` and `HostWrite` classes are used.

5.3 Maps

Maps are a conceptual data structure in Omega_h, although not a concrete C++ type, because they can simply be represented as a single array of integers (`Read<LO>`). Here we define a map as mapping from one index space to another. An index space is a range of consecutive non-negative integers, typically starting at zero and ending at some $(n-1)$, where n is the cardinality of the index space. An index space is an implicit representation of a set of n things.

5.4 Graphs

Another prevalent data structure in Omega_h is called a `Graph`, and is a representation of a directed graph (as defined by graph theory) in "compressed row" format, which is optimized for traversing the nodes of a graph, and at each node a of the graph, querying which nodes b are adjacent to a (equivalently, which edges (a, b) exist in the graph).

The structure consists of two integer arrays (the integer type is `LO`, the type used for indexing all arrays). The first array (`a2ab`) is a map from source nodes a to edges (a, b) which are sorted by source node. Because the "`ab`" index space is edges sorted by source node, it follows that all edges (a, b) with the same a have consecutive indices, and so all that needs to be stored for each a is the first and last index in the `ab` space. In addition, because the last index for one node always precedes the first index for the next node, only one of them needs to be stored. As such, the array `a2ab` has size $(n_a + 1)$, where n_a is the total number of source nodes. Given a node in the `a` index space, the first and last indices in the `ab` index space are at `a2ab[a]` and `a2ab[a + 1]`. The second array maps the sorted edges to their destination node, and so is named `ab2b`. Here is the typical way in which such a structure is accessed (in non-parallel host code):

```
for (LO a = 0; a < na; ++a) {
  for (auto ab = a2ab[a]; ab < a2ab[a + 1]; ++ab) {
    auto b = ab2b[ab];
    // do something with the (a,b) pair
  }
}
```

The name "compressed row" comes from a popular format for storing sparse matrices, and in some sense the `Graph` data structure is simply that sparse matrix format with the non-zero values omitted, because the sparsity pattern of a matrix alone is sufficient to encode a directed graph.

In the special cases where the directed graph has constant degree (i.e. all the nodes a have the same number of adjacent nodes b), then we omit the `a2ab` array, because its indices are known to be multiples of the constant degree. Downward adjacencies are a common example of this, e.g. we know that all triangles have exactly three vertices. In this case, the iteration from triangles to vertices might look like:

```
for (LO tri = 0; tri < ntris; ++tri) {
  for (auto tv = (tri * 3); tv < ((tri + 1) * 3); ++tv) {
    auto vert = tri_verts2verts[tv];
    // do something with the (tri,vert) pair
  }
}
```

5.5 The Mesh: A (Mostly) Immutable Cache

The Omega_h `Mesh` structure represents a triangle or tetrahedral mesh of a 2D or 3D domain. The mesh is described in terms of adjacency arrays, which are implemented as `Graph` data structures. These adjacency arrays describe the relationships between simplices (vertices, edges, triangles, and tetrahedra) in the mesh. In Omega_h, a mesh that contains an "element" (e.g. a tetrahedron) must also contain all the simplices on its boundary (e.g. the four triangles, six edges, and four vertices of the tetrahedron). Elements usually share these boundary simplices, for example to "adjacent" tetrahedra share a triangle, three edges, and three vertices. Although a mesh can be uniquely described by the adjacency from elements to vertices, storing a full topological representation is valuable for doing complex, robust mesh adaptation.

In an `Omega_h::Mesh` structure, the adjacency relationships cannot be changed once specified or derived. The mesh usually begins by specifying the element-to-vertex adjacency, and Omega_h will proceed to derive all other relevant adjacencies as needed. The only way to change the adjacencies is to create a new `Mesh` object.

For efficiency, all adjacency relationships are cached after being derived, since many of them are expensive to derive and are often requested repeatedly throughout mesh adaptation and simulation.