SANDIA REPORT

SAND2019-0120
Unlimited Release
Printed January 8, 2019

Nalu's Linear System Assembly using Tpetra

Stefan Domino and Alan Williams

Prepared by
Sandia National Laboratories
Albuquerque, New Mexico 87185 and Livermore, California 94550

Sandia National Laboratories is a multimission laboratory managed and operated by National Technology and Engineering Solutions of Sandia, LLC., a wholly owned subsidiary of Honeywell International, Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525.

Sandia National Laboratories

Issued by Sandia National Laboratories, operated for the United States Department of Energy by National Technology and Engineering Solutions of Sandia, LLC.

Available to DOE and DOE contractors from
U.S. Department of Energy
Office of Scientific and Technical Information
P.O. Box 62
Oak Ridge, TN 37831

Telephone: (865) 576-8401
Facsimile: (865) 576-5728
E-Mail: reports@adonis.osti.gov
Online ordering: http://www.osti.gov/bridge

Available to the public from
U.S. Department of Commerce
National Technical Information Service
5285 Port Royal Rd
Springfield, VA 22161

Telephone: (800) 553-6847
Facsimile: (703) 605-6900
E-Mail: orders@ntis.fedworld.gov
Online ordering: http://www.ntis.gov/help/ordermethods.asp?loc=7-4-0#online

SAND2019-0120
Unlimited Release
Printed January 8, 2019

Nalu's Linear System Assembly using Tpetra

Stefan Domino and Alan Williams

Sandia National Laboratories
P.O. Box 5800
Albuquerque, NM 87185

Abstract

The Nalu Exascale Wind application assembles linear systems using data structures provided by the Tpetra package in Trilinos. This note describes the initialization and assembly process. The purpose of this note is to help Nalu developers and maintainers to understand the code surrounding linear system assembly, in order to facilitate debugging, optimizations, and maintenance. [1]

[1]Sandia National Laboratories is a multimission laboratory managed and operated by National Technology and Engineering Solutions of Sandia, LLC., a wholly owned subsidiary of Honeywell International, Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525. This report followed the Sandia National Laboratories formal review and approval process (SAND2019-0120), and is suitable for unlimited release.

This page intentionally left blank.

Contents

This page intentionally left blank.

Chapter 1

Overview

Nalu constructs a linear system by mapping degrees of freedom at mesh nodes to equations, i.e., rows in the matrix and rhs vector. In a parallel run, the linear system that is given to the solver contains equations that correspond to locally owned mesh nodes, which are the mesh nodes that are owned by the local MPI rank. Each MPI rank can also have mesh nodes which are shared but not owned, as well as mesh nodes that are ghosted. Shared-but-not-owned nodes are owned by another MPI rank but are connected to an element that is locally owned. Ghosted mesh nodes tend to arise in cases that involve periodic boundaries, and cases that involve mesh contact which is handled via a Discontinuous Galerkin scheme.

Consider the simple two-element mesh decomposed onto 2 MPI ranks and shown in figures 1.1 and 1.2.

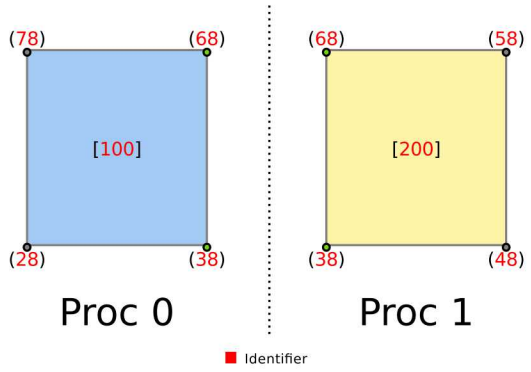

Figure 1.1: Parallel-decomposed mesh. Two elements, one on each MPI processor. Note that some mesh nodes appear on both processors.

Nalu uses Tpetra::Map objects to identify the degrees of freedom and processor layout. There are two maps, `ownedRowsMap_` and `sharedNotOwnedRowsMap_`. For the simple two-element mesh in this example, figure 1.3 shows how the owned and shared maps would be defined.

We then create `Tpetra::CrsGraph` objects for owned and shared (these objects are called `ownedGraph_` and `sharedNotOwnedGraph_`), each using the appropriate row-map. These graphs both use the same column map, and the creation and initialization of the column map will be discussed in a later section. We also create Tpetra::CrsMatrix objects for owned and shared, called `ownedMatrix_` and `sharedNotOwnedMatrix_`.

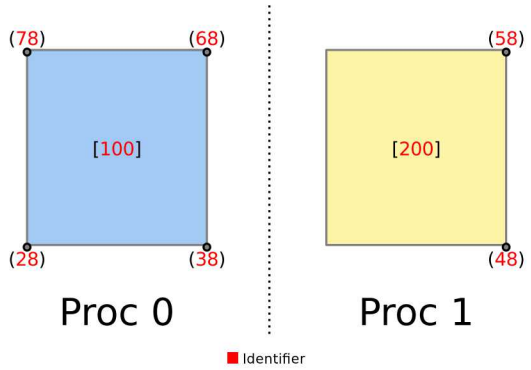

Figure 1.2: Locally-owned nodes. Nodes 38 and 68 are owned by process 0 and are shared but not owned by process 1.

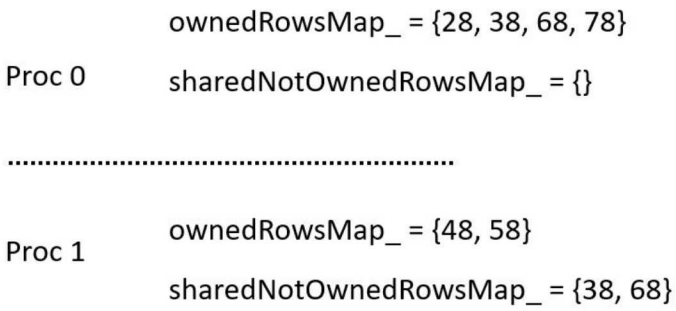

Figure 1.3: Owned and Shared-not-Owned Maps.

Each processor contributes an element-matrix of coefficients for its local elements as shown in figure 1.4.

Note that processor 1 has contributions for some rows (38 and 68) that it doesn't own. Each processor contributes the rows of its element-contributions to the appropriate matrix object depending on whether the row is owned or not. Once all element-contributions have been assembled, the contents of the shared-but-not-owned matrix are sent to the appropriate processors and added to the owned matrix.

Listing 1.1: Assembly using export

```
sharedNotOwnedMatrix_->fillComplete();

ownedMatrix_->doExport(*sharedNotOwnedMatrix_,
                       *exporter_, Tpetra::ADD);

ownedMatrix_->fillComplete();
```

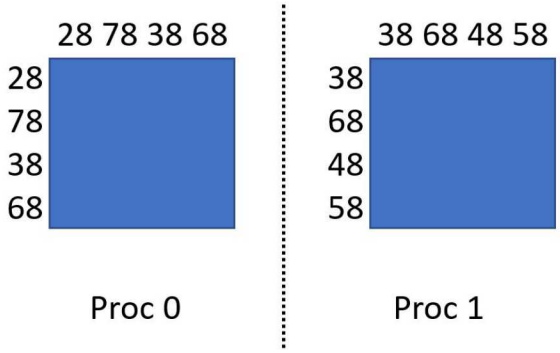

Figure 1.4: Element-matrix contributions per processor.

This is done using Tpetra methods as shown in listing 1.1. The result is an assembled global matrix as shown in figure 1.5.

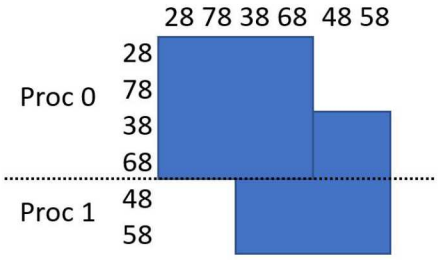

Figure 1.5: Assembled `ownedMatrix_`.

Note that corresponding operations are performed in the assembly of the right-hand-side vector, in terms of owned and shared, export, etc.

Once the `fillComplete` operation has completed, the linear system is fully assembled and is ready to be passed to solvers and/or preconditioners. Later sections will go into more detail on the construction of the column map and graph objects, in order to correctly and efficiently incorporate off-processor column entries, etc.

We broadly split the construction of the linear system into two phases called initialization and assembly. The initialization phase is where we construct the maps, perform communication to send column indices to appropriate processors, and construct graph objects. The assembly phase is where we assemble coefficient values into the matrix using a "sum-into" operation, and also modify the linear system to enforce boundary conditions. In simulations where the structure of the linear system doesn't change from one timestep to the next, we can gain efficiency by performing the initialization phase once and then reusing the data structures for many assembly and solve phases.

Mesh nodes can also be ghosted, which means they are copied from the owning processor to another processor which has no connectivity relationship with those nodes. Ghosted nodes (on the receiving processor) are neither owned nor shared. Ghosting can happen when periodic boundaries of the mesh are mapped to each other across MPI processors. Ghosting can also happen when a mesh surface is in contact with another mesh surface where the elements on either side of the contact don't share connected nodes. See figure 1.6 for an example of a mesh with contact boundaries going through the middle. In this case elements along the sliding boundary can be ghosted to the processor on the other side of the boundary. When elements are ghosted, the connected nodes of those elements are also ghosted.

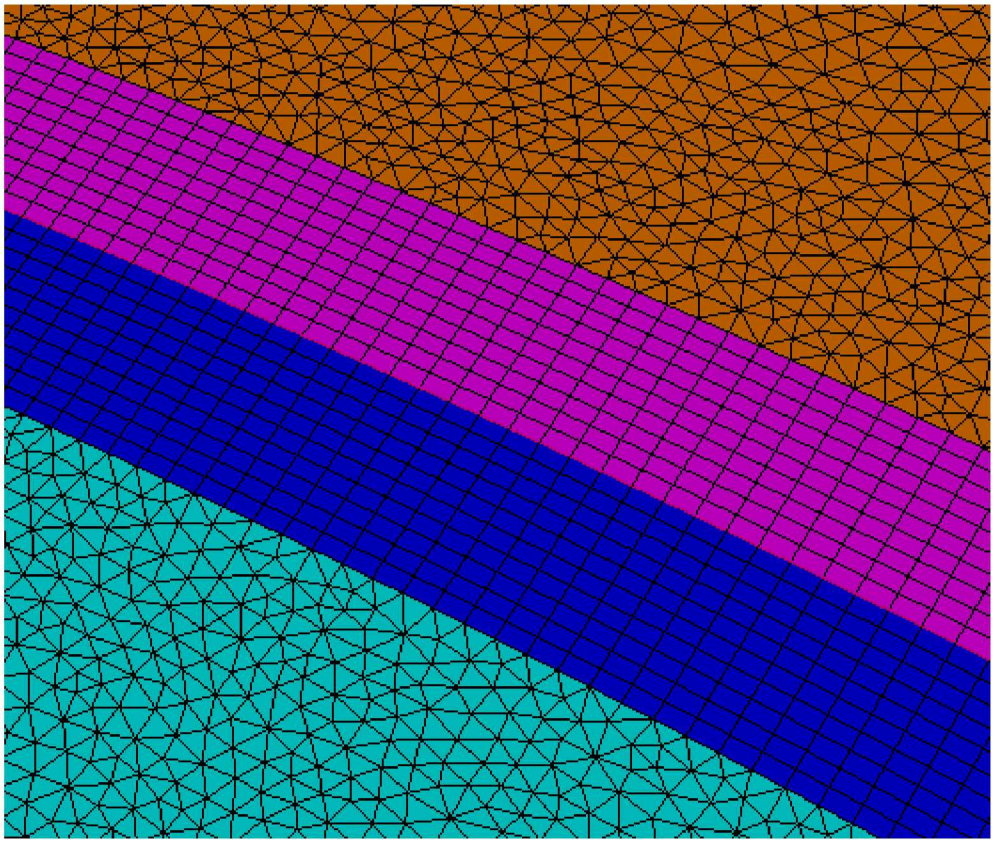

Figure 1.6: Mesh with sliding contact boundary.

Ghosted nodes don't directly produce equations (rows) in the assembled linear system, but can result in additional column-entries in other rows.

Chapter 2

Initialization

Each linear system assembly includes an initialization phase, where the degrees of freedom and decomposition are defined using Tpetra::Map, and connectivities are mapped to a compressed row storage (CRS) graph structure. The following list describes, with some pseudo-code, the broad sequence used to create and initialize the Tpetra objects. The code that implements these steps resides in the Nalu class `TpetraLinearSystem`.

1. Create owned and shared maps, and exporter to communicate between them.

   ```
   ownedRowsMap_ = new Map(ownedGlobalIds, ...);
   sharedNotOwnedRowsMap_ = new Map(sharedNotOwnedGlobalIds, ...);
   exporter_ = new Exporter(sharedNotOwnedRowsMap_,
                            ownedRowsMap_);
   ```

2. Accumulate lists of connectivities defined by element-node connections, etc. Currently these connectivity lists are stored in a simple vector-of-vectors structure to represent the 'ragged table' of node-to-node connections.

3. Using mesh information, determine row-lengths due to local connections and due to contributions from neighbor processors (which share rows that we own, and will thus contribute column-indices to some of our rows). The STK mesh structure can tell us the list of neighboring MPI processors, i.e., the MPI processors which have portions of the mesh which connect to the local MPI processor's portion of the mesh.

   We then traverse our connectivities and for each shared-but-not-owned entity we send column entries to the graph on the MPI processor that owns the entity. This enables us to compute exact row-lengths for each row in the graph.

   Note that this step, and the remaining initialization steps, are implemented in the Nalu method `TpetraLinearSystem::finalizeLinearSystem()`.

4. Create and fill Kokkos::Views representing the local portion of the compressed-sparse-row graphs. These are the `rowPointers` and `columnIndices` arrays that will be passed into the `Tpetra::CrsGraph::setAllIndices` method later. As an implementation detail, during construction we hold these Kokkos::Views in a Nalu structure called `LocalGraphArrays` and this object has methods for inserting entries, etc.

5. Construct column map. Fill a list of ids representing the union of all column-entries that will be on the local processor. These ids are ordered as follows.

   (a) Owned: indices that the local processor owns.

   (b) Shared: indices that the local processor shares but doesn't own.

   (c) Remote: sometimes called 'reachable', these are indices that are neither owned nor shared, but are connected to shared on a neighboring processor. These indices are grouped by which processor they come from.

```
totalColsMap_ = new Map(colGlobalIds, ...);
```

6. Construct Graphs.

```
sharedNotOwnedGraph_ = new Graph(sharedNotOwnedRowsMap_,
                                 totalColsMap_, ...);
ownedGraph_ = new Graph(ownedRowsMap_,
                        totalColsMap_, ...);

ownedGraph_->setAllIndices(/* our computed local
                              owned-graph indices */);
sharedNotOwnedGraph_->setAllIndices(/* our computed local
                                       shared-graph indices */);

importer = new Import(ownedRowsMap_, non-owned-col-ids, source-pids);

ownedGraph_->expertStaticFillComplete(ownedRowsMap_,
                           ownedRowsMap_, importer, ...);
sharedNotOwnedGraph_->expertStaticFillComplete(ownedRowsMap_,
                                  ownedRowsMap_, importer, ...);
```

7. Construct Matrices

```
ownedMatrix_ = new Matrix(ownedGraph_);
sharedNotOwnedMatrix_ = new Matrix(sharedNotOwnedGraph_);
```

Chapter 3

Assembly

The linear system assembly phase is where matrix and rhs-vector contributions are added, using sum-into APIs.

Nalu computes matrix and rhs-vector contributions in classes which are derived from the `Kernel` base-class. These kernels include computations such as continuity-advection, momentum-advection-diffusion, and many others. These kernels are called within an algorithm which manages mesh traversal, handles the gathering of field-data and master-element computations, etc. The following pseudo-code shows the execution or flow of these algorithms.

```
select buckets for part(s) that algorithm is applied to

for each selected bucket {
   ScratchViews scratch(gathered fields);

   for each elem in bucket {
        fill_scratch_views(scratch, ...);

        for each active kernel {
           kernel->execute(lhs, rhs, scratch);
        }
        sum_into_linear_system(lhs, rhs);
    }
}
```

Note that in the above pseudo-code, `fill_scratch_views` includes both gathering needed field-data as well as making master-element calls such as grad-op, etc.

Note also that significant detail has been omitted, such as the handling of SIMD sub-looping, interleaving the SIMD data, etc. To see the actual algorithm code, see the Nalu classes `AssembleElemSolverAlgorithm`, and `AssembleFaceElemSolverAlgorithm`.

The `sum_into_linear_system` step is where the computed coefficients are contributed to the linear-system. This is done by obtaining the local matrix from the `Tpetra::CrsMatrix` object using the method `getLocalMatrix()` and then summing directly into that. Similarly, rhs coefficients

are summed into the `Kokkos::View` provided by the `getLocalView` method of `Tpetra::Vector`.

Recall that we are operating on two matrix objects, `ownedMatrix_` and `sharedNotOwnedMatrix_`. During this sum-into process, coefficients are added to the appropriate matrix (and rhs-vector) depending on whether they are associated with an owned or shared mesh node. Once all local assembly is complete, the shared matrix is then communicated and added to the local matrix using `Tpetra` operations, called from within the Nalu method `TpetraLinearSystem::loadComplete()`:

```
sharedNotOwnedMatrix_->fillComplete();

ownedMatrix_->doExport(*sharedNotOwnedMatrix_, *exporter, Tpetra::ADD);

ownedMatrix_->fillComplete();

ownedRhs_->doExport(*sharedNotOwnedRhs_, *exporter_, Tpetra::ADD);
```

3.1 Boundary condition enforcement

Dirichlet boundary conditions are enforced by modifying the appropriate rows of the assemled linear system before calling the linear-solver.

Insert pseudo-code for dirichlet BC enforcement here.

DISTRIBUTION:

1 MS 0899 Technical Library, 9536 (electronic copy)

This page intentionally left blank.


Sandia National Laboratories