Short Trilinos Documentation

Bernd Riedel

February 27, 2018

Contents

1 Introduction to Trilinos

Shortly explained Trilinos is an object-oriented framework for the solution of large-scale, complex multi-physics engineering and scientific problems. It contains a lot of independent packages which provide specific functionalities such as

1. Basic linear algebra computation
2. Linear/Nonlinear solving and preconditioning
3. Solving of eigenvalue problems
4. Optimization
5. Partitioning and load balancing
6. Mesh generation and adaption

This documention will mainly focus on features 1 and 5 to show and explain the usage, advantages/disadvantages and some critical points. One important capability which is not mentioned yet, because it is more or less availabe in each package, is the parallilization of code. Trilinos enables in the most packages the use of a MPI library (e.g. OpenMPI or MPICH).
In addition to this documentation it is recommended to have a look at the example files located in the "NuTo-Trilinos" branch on GitHub, which will also be referenced a few times within this paper.

2 Main data structure packages

The core of many computations consists of some basic linear algebra objects like matrices and vectors. The corresponding types and methods are defined in the Petra package collection. Roughly spoken this collection are divided into two packages, called Epetra and Tpetra.

2.1 Epetra

The package Epetra ("E" for essential) contains concrete classes for (multi-)vectors, matrices and graphs, in sparse or dense format. Furthermore it provides serial and parallel capabilities. As backend it uses BLAS and LAPACK where possible to achive good performance results.

The important classes for the most tasks are **Epetra_Vector**, **Epetra_MultiVector**, **Epetra_CrsMatrix** (sparse matrix) and **Epetra_VbrMatrix** (sparse block matrix). To construct these objects a communicator, **Epetra_Comm** (abstract class), is needed at first. Depending on the desired functionality, serial or parallel, you have to use the correct implementation of this communicator, **Epetra_SerialComm** or **Epetra_MpiComm**. Afterwards one has to create an **Epetra_Map** object with the help of the used communicator. This map describes the mapping of process specific indices to global indices, for example if a matrix should be distributed over more processes. In simple cases the map constructors build linear mappings, i.e. the first processor contains the global IDs $0, .., m$ and the second processor the IDs $m+1, .., n$. But in general an arbitrary mapping is possible, also with overlapping indices. The created map object can be used to create an **Epetra_Graph** which represents the sparse structure of a matrix or a matrix (or a vector) can be constructed directly (see example below).

Code example 1:

```
void createSparseMatrix()
{
  Epetra_MpiComm Comm(MPI_COMM_WORLD);  //MPI communicator
  int myProcID = Comm.MyPID();          //process ID
  int numProc = Comm.NumProc();         //complete number of processes
  int numGlobalEntries = 10;            //number of global IDs
  Epetra_Map map(numGlobalEntries, 0, Comm);
  int entriesPerRow = 3;                //estimated or known number of nonzeros per row
  Epetra_CrsMatrix mat(Epetra_DataAccess::Copy, map, entriesPerRow, true);  //global size 10 x 10
}
```

Remark: The last parameter in the matrix constructor describes the memory profile. If it is set to true, the value of *entriesPerRow* will be interpreted as maximal and the needed memory will be allocated at this point. This leads to a performance improvement in multiplication and solving routines. But you are not able to add more entries if necessary. If the parameter is set to false, the memory is dynamically determined. You have to use this parameter carefully, because some strange errors can occur, if you want to add more values than prescribed the profile is static.

If the matrix is not created with the help of a **Epetra_CrsGraph** (like in the example above) the entries have to be specified to compute the underlying sparse structure. That can be done with different functions for local and global insertion like **InsertMyValues()**, **InsertGlobalValues()**, **ReplaceMyValues()**, **ReplaceGlobalValues()**, **SumIntoMyValues()** or **SumIntoGlobalValues()**.

At the end the method **FillComplete()** should be called to signal the final sparse structure. That is necessary for correct interaction with other objects. Although no parameters are needed it is recommended to specify the domain and range map to describe the index spaces.

A further important feature is the export between local objects. Unfortunately the correct addition of overlapping matrices or vectors is not easily done by map construction (the term *overlap* means that local indices of different processes map to the same global index). For this purpose two **Epetra_Map**'s are needed. One describing a disjoint mapping of indices (called here and in the example *owningMap*), like in a simple linear map as mentioned above. The other (called *overlappingMap*) discribes the overlap of indices. With these two maps an **Epetra_Export** object can be constructed which itself represents the actual overlap. All local algebra objects will be created with the overlappingMap. In the end or if the global objects are needed, respectively, one can call an export method with the desired interaction behavior (see example below):

Code example 2:

```
void createOverlappingMatrix()
{
  //owningMap and overlappingMap created previously
  int entriesPerRowLocal = 5;
  int entriesPerRowGlobal = 10
  Epetra_CrsMatrix matLocal(Epetra_DataAccess::Copy, overlappingMap, entriesPerRowLocal, true);
  //set values of matLocal
  matLocal.FillComplete();  //fix sparse structure
  Epetra_CrsMatrix matGlobal(Epetra_DataAccess::Copy, owningMap, entriesPerRowGlobal, true);
  Epetra_Export exporter(overlappingMap, owningMap);
  matGlobal.Export(matLocal, exporter, Epetra_CombineMode::Add);  //add overlapping entries
}
```

Overlapping indices may be a result of computations on a partitioned domain. In this case the entries in a corresponding stiffness matrix have to be added together, if the local degrees of freedom on a subdomain describe the same global degrees of freedom. Therefore make use of **Epetra_CombineMode::Add**. How such owning and overlapping maps are constructed is shown for example in the "StructureMeshTest.cpp" file.

Beside the mentioned important features there are some basic methods of course, for value manipulation, multiplication/addition, scaling and inversion.

2.2 Tpetra and Xpetra

The Tpetra package ("T" stands for template) offers the same capabilities in general except the default usage in methods (smart pointers are needed). Due to that an additional package is necessary, **Teuchos**. This package provides some helpful tools and wrappers. The main features for the use of Tpetra are **Teuchos::Comm** and **Teuchos::RCP**. The first one represents the communicator (similar to **Epetra_Comm**) and the second one supports the construction of smart pointers for Tpetra objects. The same method as in code example 1 reads here:

Code example 3:

```
using Teuchos::RCP;
using Teuchos::rcp;
void createSparseMatrix_Tpetra()
{
  RCP<const Teuchos::Comm<int>> Comm = Tpetra::DefaultPlatform::getDefaultPlatform().getComm();
  int myProcID = Comm->getRank();          //process ID
  int numProc = Comm->getSize();          //complete number of processes
  int numGlobalEntries = 10;             //number of global IDs
  RCP<Tpetra::Map<int, int>> map = rcp(new Tpetra::Map<int, int>(numGlobalEntries, 0, Comm);
  int entriesPerRow = 3;                 //estimated or known number of nonzeros per row
  RCP<Tpetra::CrsMatrix<double, int, int>> mat =
    rcp(new Tpetra::CrsMatrix(map, entriesPerRow, Tpetra::ProfileType::StaticProfile);
}
```

Remark: At first it is noticeable that a distinction between a serial or parallel communicator is not absolutely required, because both (**Teuchos::SerialComm** and **Teuchos::MpiComm**) inherit from **Teuchos::Comm**. Furthermore it is also possible to prescribe the memory storage behavior with the **ProfileType** (**StaticProfile** or **DynamicProfile**). The template arguments describe the ordinal type of the corresponding indices (here all **int**'s, i.e. process IDs in **Comm** object, local and global IDs in the map and the matrix) and the type of the values in the matrix (here **double**).

The second code example can analogously be recreated by substituting the Epetra objects with Teuchos smart pointers of Tpetra objects, because all objects and methods are name very similar.

At the end of this section the Xpetra package will be mentioned. Probably in future versions of Trilinos this package will be used instead of Epetra and Tpetra, because it wraps both (not completely yet), but nowadays Tpetra is still the better choice. The syntax of Xpetra objects and methods is the same as for Tpetra, therefore a later adjustment should be straightforward.

3 Main solver packages

Although the Petra package provide some simple solve methods the use of explicitly solvers is recommended. In this section two important Trilinos packages will be presented. On the one hand **Amesos/Amesos2** (direct solvers) and on the other hand **Belos** (iterative solvers). Both offer serial and parallel computations.

In this section it is always assumed that the matrices and vectors which represent the problems are known.

3.1 Amesos/Amesos2

Amesos is a lightweight interface to different solver libraries which have to be installed external, e.g. LAPACK, SuperLU or MUMPS. Furhtermore it includes internal solvers called KLU (serial) and Paraklete (parallel). To run a solver you have to define an **Epetra_LinearProblem** at first. Hence it only works for **Epetra** objects. How that can be done is shown in the next example:

Code example 4:

```
void solve_Amesos(Epetra_CrsMatrix rA, Epetra_MultiVector rLhs, Epetra_MultiVector rRhs)
{
  Epetra_LinearProblem problem(&rA, &rLhs, &rRhs);
  Amesos Factory;
  Amesos_BaseSolver* solver;
  std::string solverType = "Klu";  //or "Paraklete", "Mumps", etc.
  solver = Factory.Create(solverType, problem);
  solver->Solve();
}
```

The future of **Amesos** is the **Amesos2** package. It provides the similar internal solvers as well as wrappers to external solvers. Since **Amesos2** is able to handle templates for the underlying objects it is possible to solve problems formulated with Epeta, Tpetra or Xpetra backend (see example 5 for the use of Tpetra). Furthermore if you are not sure if the desired (external) solver is installed, you can check it before with the **query()** function.

Code example 5:

```
using Teuchos::RCP;
using Teuchos::rcp;
void solve_Amesos2(RCP<const Tpetra::CrsMatrix<double, int, int>> rA,
     RCP<const Tpetra::Vector<double, int, int>> rRhs,
     RCP<Tpetra::Vector<double, int, int>> rLhs)
{
  typedef Tpetra::CrsMatrix<double, int, int> matType;
  typedef Tpetra::MultiVector<double, int, int> vecType;
  std::string solverType = "KLU2";  //or "Basker", "Mumps", "SuperLU", etc.
  if(Amesos2::query(solverType)){
    RCP<Amesos2::Solver<matType, vecType>> solver =
      Amesos2::create<matType, vecType>(solverType, rA, rLhs, rRhs);
    solver->symbolicFactorization();//can be done
    solver->numericFactorization(); //in one line by
    solver->solve();                //solver->symbolicFactorization().numericFactorization().solve()
  }
}
```

Remark: It is also possible to define some solver specific options with the help of another **Teuchos** functionality, called **ParameterList**. How it works in general see example 6. The names of the third library solvers can be found in the corresponding documentations.

3.2 Belos

As mentioned in the introduction to this section **Belos** is a package which offers the iterative solution of problems. There are different solvers available, for example GMRES, Block GMRES, CG and BiCGStab. Similar to **Amesos** it is necessary to define a specific linear problem object to compute a solution. Besides it is also able to use templates such that it does not matter which concrete objects you use for the matrices and vectors. It is again possible to set some solver specific options. A short example with **Tpetra** backend is shown below, a more complex can be found in "StructureMeshTest.cpp" in the repository.

Code example 6:

```
using Teuchos::RCP;
using Teuchos::rcp;
void solve_Belos(RCP<const Tpetra::CrsMatrix<double, int, int>> rA,
     RCP<const Tpetra::Vector<double, int, int>> rRhs,
     RCP<Tpetra::Vector<double, int, int>> rLhs)
{
  RCP<Teuchos::ParameterList> params = Teuchos::parameterList();
  params->set("Maximum Iterations", 500);
  typedef Tpetra::CrsMatrix<double, int, int> matType;
  typedef Tpetra::MultiVector<double, int, int> vecType;
  typedef Belos::LinearProblem<double, vecType, matType> problemType;
  RCP<problemType> problem = rcp(new problemType(rA, rLhs, rRhs);
  std::string solverType = "GMRES";  //or "CG", "MINRES", "Block CG", etc.
  typedef Belos::SolverFactory<double, vecType, matType> factory;

  if(factory.isSupported(solverType)){
    RCP<typedef Belos::SolverManager<double, vecType, matType> solver = factory.create(solverType);
    solver->setProblem(problem);
    Belos::ReturnType result = solver->solve();
  }
}
```

Remark: The parameters are always pairs of the parameter's name and the corresponding value. Further parameters (e.g. "Num Blocks", "Convergence Tolerance") are found in the example files or in the **Belos** documentation. Besides the complete list of supported solvers (with corresponding aliases) are found there in **Belos::SolverFactory**.

4 Partitioning and load balancing with Zoltan package

This section shows how to partition and migrate data between processes, e.g. for problem which should be solved on a few subdomains. Of course it is possible to do this more or less by hand by create all maps carefully and compute overlapping data on your own. But the following package, **Zoltan**, provides some methods which perform the partition and also the data migration with a little help by the user. For understanding it is recommended to have a look at the "ZoltanTest.cpp" (e.g. method *AssemblerZoltanTest_1D*) file. In the future it is possible that the new version **Zoltan2** will replace the current package but now are just a few partition and ordering methods impemented.

The main method for the mentioned purposes is **Zoltan_LB_Partition** which produces (of course) the partition of the prescribed complete structure. Although it is possible to run a separate migration function, there is a parameter to perform an automatic migration (how it is set see below). Though the **Zoltan** package provides general functionalities, the explicit handling with your data is unknown. Firstly it has to be clear what exactly should be migrated, i.e. which objects of the complete structure and what are the connections between these objects. In the example files a mesh will be partitioned. Therefore elements, which are connected by nodes, will migrate. Besides some so called query functions have to be defined by the user to describe how the objects can be handled. For each main step, partition and migration, four query functions are required.
For partition:

- **Zoltan_Set_Num_Obj_Fn** → prescribe (process) local number of data objects
- **Zoltan_Set_Obj_List_Fn** → prescribe local and global IDs of data objects
- **Zoltan_Set_HG_Size_CS_Fn** → prescribe number of data connections and connection IDs
- **Zoltan_Set_HG_CS_Fn** → prescribe local IDs of objects and connections

For migration:

- **Zoltan_Set_Obj_Size_Fn** → prescribe size of data objects
- **Zoltan_Set_Pack_Obj_Fn** → prescribe how/where to save data objects
- **Zoltan_Set_Unpack_Obj_Fn** → prescribe how to rebuild saved data objects
- **Zoltan_Set_Mid_Migrate_PP_Fn** → prescribe adaption of global structure

The last method **Zoltan_Set_Mid_Migrate_PP_Fn** is not absolutely need, but again recommended. There two further methods **Zoltan_Set_Pre_Migrate_PP_Fn** and **Zoltan_Set_Post_Migrate_PP_Fn** where some pre- and postprocessing steps can be declared. The first three functions for migration handle only one object at once. These can be replaced by **Zoltan_Set_Obj_Size_Multi_Fn**, **Zoltan_Set_Pack_Obj_Multi_Fn** and **Zoltan_Set_Unpack_Obj_Multi_Fn** to provide multiple data handling.

If the necessary query functions are created the complete procedure reads as in the example below.

Zoltan demonstration:

```
void Zoltan_partition_migration(int argc, char* argv)
{
  MeshType mesh();  //here the structure to partition
  float version;
  int returnVal = Zoltan_Initialize(argc, argv, &version);  //if returnVal == ZOLTAN_OK go on
  struct Zoltan_Struct* zolt = Zoltan_Create(MPI_COMM_WORLD);
  Zoltan_Set_Param(zolt, "LB_METHOD", "HYPERGRAPH");
  Zoltan_Set_Param(zolt, "LB_APPROACH", "PARTITION");  //or "REPARTITION"
  Zoltan_Set_Param(zolt, "AUTO_MIGRATE", "TRUE");  //auto migration after partition

  Zoltan_Set_Num_Obj_Fn(zolt, ..., &mesh);  //necessary methods have to be defined by user
  Zoltan_Set_Obj_List_Fn(zolt, ..., &mesh);
  Zoltan_Set_HG_Size_CS_Fn(zolt, ..., &mesh);
  Zoltan_Set_HG_CS_Fn(zolt, ..., &mesh);

  Zoltan_Set_Obj_Size_Fn(zolt, ..., &mesh);
  Zoltan_Set_Pack_Obj_Fn(zolt, ..., &mesh);
  Zoltan_Set_Unpack_Obj_Fn(zolt, ..., &mesh);
  Zoltan_Set_Mid_Migrate_PP_Fn(zolt, ..., &mesh);

  rc = Zoltan_LB_Partition(zolt, /* input (all remaining fields are output) */  //all output
              &changes,        /* 1 if partitioning was changed, 0 otherwise */ //objects have
              &numGidEntries,  /* Number of integers used for a global ID */    //to be declared
              &numLidEntries,  /* Number of integers used for a local ID */     //at the beginning
              &numImport,      /* Number of vertices to be sent to me */
              &importGlobalGids,  /* Global IDs of vertices to be sent to me */
              &importLocalGids,   /* Local IDs of vertices to be sent to me */
              &importProcs,    /* Process rank for source of each incoming vertex */
              &importToPart,   /* New partition for each incoming vertex */
              &numExport,      /* Number of vertices I must send to other processes*/
              &exportGlobalGids,  /* Global IDs of the vertices I must send */
              &exportLocalGids,   /* Local IDs of the vertices I must send */
              &exportProcs,    /* Process to which I send each of the vertices */
              &exportToPart); /* Partition to which each vertex will belong */

  Zoltan_Destroy(&zolt);
}
```

Remark: There are also some other values for the paramater *LB_METHOD*, but the most practical one is the hypergraph method because this enables the most general cases. There are no further assumptions made on the data and it is on your own what are the so called hyperedges and hypervertices (in the example files it is vice-versa, hyperedges are nodes and hypervertices are elements). Beside the shown parameters there exist a few others for possible graph weights, underlying ordering method, debug levels and so on. A full list can be found in the official **Zoltan** User's Guide. In some experiments it turns out that dynamic size objects (e.g. std::vector, Eigen::VectorXd, etc.) are not easy to use in the pack and unpack routines. Because of not clearly allocated memory it can happen that **Zoltan** is not able to find the correct and complete saved object. Therefore use at first fixed size objects or write complete save routines by hand. After the partition and the migration is done correctly, the structure object holds the right objects and information on each process.

Final remark: In the example files especially in *ZoltanTest.cpp* is a complete procedure implemented where at first a simple mesh object is created an distributed over the processes. Afterwards the partition/migration routine is called to get submeshes. The IDs of the degrees of freedom can then be locally enumerated but also know their global number. According to this a *Local-To-Global-Mapping* can be build used to generate owning and overlapping maps (see section 2) for the system matrices and right hand sides. If this mapping is created in the right way you can use a direct or iterative solver to get a parallel solution on the global problem.

5 Package documentations

Here is a short list where to find the documentation to the mentioned packages.

- Trilinos: `https://trilinos.org/about/documentation/`
- Epetra: `https://trilinos.org/docs/dev/packages/epetra/doc/html/index.html`
- Tpetra: `https://trilinos.org/docs/dev/packages/tpetra/doc/html/index.html`
- Xpetra: `https://trilinos.org/docs/dev/packages/xpetra/doc/html/index.html`
- Amesos2: `https://trilinos.org/docs/dev/packages/amesos2/doc/html/index.html`
- Belos: `https://trilinos.org/docs/dev/packages/belos/doc/html/index.html`
- Zoltan: `http://www.cs.sandia.gov/Zoltan/ug_html/ug.html`

Unfortunately some documentations are brief such that some try and error is the only way to get something right. A few little tutorials (`https://github.com/trilinos/Trilinos_tutorial/wiki/TrilinosHandsOnTutorial`) and other documents can help, e.g. Epetra Performance Optimization Guide (`https://trilinos.org/wordpress/wp-content/uploads/2014/10/EpetraPerformanceGuide.pdf`) or TriBITS Developers Guide (`https://tribits.org/doc/TribitsDevelopersGuide.html`).