



# BaNaNa Talk

Trilinos

---

Alexander Heinlein

May 19, 2021

TU Delft

*Project*




**BaNaNa**

## Disclaimer

The following slides will give a brief overview over the software package Trilinos. It is far from complete, but on the final slides, some *references to additional introductory material and tutorials will be given.*

# What is Trilinos?

From the report

 M. A. Heroux, R. A. Bartlett, V. E. Howle, R. J. Hoekstra, J. J. Hu, T. G. Kolda, R. B. Lehoucq, K. R. Long, R. P. Pawlowski, E. T. Phipps, and A. G. Salinger

## **An overview of the Trilinos project.**

ACM Transactions on Mathematical Software (TOMS) 31.3 (2005): 397-423.

*“The Trilinos Project is an effort to facilitate the design, development, integration, and ongoing support of mathematical software libraries within an object-oriented framework for the solution of large-scale, complex multiphysics engineering and scientific problems.”*

**Trilinos** is a collection of more than 50 software packages:

- Each Trilinos package is a *self-contained, independent piece of software* with its own set of requirements, its own development team<sup>1</sup> and group of users.
- However, there are often certain *dependencies between different Trilinos packages*. Some Trilinos packages also *depend on third party libraries*.
- Generally, a *certain degree of interoperability* of the different Trilinos packages is provided.

---

<sup>1</sup>Trilinos lead: Sandia National Laboratories

# Why using Trilinos?

## Wide range of functionality

<b>Data services</b>	Vectors, matrices, graphs and similar data containers, and related operations
<b>Linear and eigen-problem solvers</b>	For large, distributed systems of equations
<b>Nonlinear solvers and analysis tools</b>	Includes basic nonlinear approaches, continuation methods and similar
<b>Discretizations</b>	Tools for the discretization of integral and differential equations
<b>Framework</b>	Tools for building, testing, and integrating Trilinos capabilities

## Portable parallelism

Trilinos is targeted for all major parallel architectures, including

- distributed-memory using the Message Passing Interface (MPI),
- multicore using a variety of common approaches,
- accelerators using common and emerging approaches, and
- vectorization.

*“... as long as a given algorithm and problem size contain enough latent parallelism, **the same Trilinos source code** can be compiled and execution on **any reasonable combination of distributed, multicore, accelerator and vectorizing computing devices.**”* — Trilinos Website

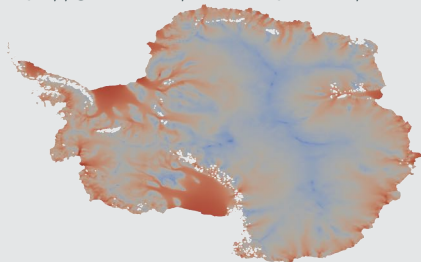
# Examples for Using Trilinos

Many scientific computing software packages are **based on Trilinos** or **provide a Trilinos interface**.

## Antarctica Landice Simulations



<https://github.com/SNLComputation/Albany>

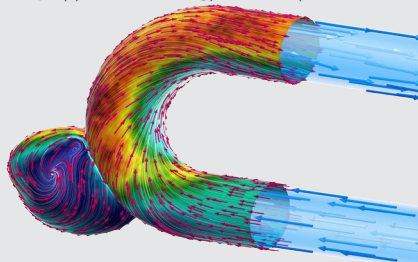


See **Heinlein, Perego, Rajamanickam (TR 2021)**.

## Intracranial Aneurysm Simulations



<https://bitbucket.org/lifev-dev/lifev-release>



See **Giese, Heinlein, Klawonn, Knepper, Sonnabend (2019)**.

# How to install Trilinos?

## Option 1

Trilinos is **available through most package managers** for Linux operating systems. However, when installing Trilinos via package manager, we **do not have full control over its configuration**.

## Option 2

In order to have **full control over the configuration** of Trilinos, it may be compiled and installed from the source files.

1. The source files of Trilinos have to be downloaded:

- Trilinos is developed using a **Git** repository, which is hosted on GitHub:

`https://github.com/trilinos/Trilinos`

*(Git is a free and open source distributed version control system; GitHub is a platform for hosting git repositories)*

- The code can be *downloaded as a zip archive* or by *cloning the git repository*:

```
git clone https://github.com/trilinos/Trilinos.git
```

2. Trilinos is then configured using **CMake**.

(*CMake is an open-source, cross-platform family of tools designed to build, test and package software.*)

- Exemplary CMake scripts are provided in the Trilinos repository in the subfolder `sampleScripts`. They may have to be modified to make sure that the right paths, compilers, flags, etc. are used.

3. In order to compile and install Trilinos, we need *three directories* in total. In particular,

```
cmake -D CMAKE_INSTALL_PREFIX=$INSTALL_DIR $SOURCE_DIR
```

should be executed within a `BUILD_DIR` directory.

(*Other CMake parameters can be added with `-D CMAKE_PARAMETER=value`*)

`SOURCE_DIR` is the directory with the source code of Trilinos.

`BUILD_DIR` is a directory which is used to compile Trilinos. It will contain both the object files containing all the **compiled source code** as well as the **compiled tests** and **examples**.

`INSTALL_DIR` is the directory which, finally, will contain the necessary *header files* and *libraries* to **use Trilinos as a library**.

Particularly important are also the CMake parameters

```
-D TPL_ENABLE_XXX:BOOL=ON/OFF
```

for enabling/disabling the third-party library XXX and

```
-D Trilinos_ENABLE_YYY:BOOL=OFF
```

for enabling/disabling the Trilinos package YYY.

4. After configuration with `cmake`, Trilinos can be compiled with

```
make -j4
```

and installed with

```
make install
```

**Note:** The parameter `-j4` indicates that **compilation should be performed in parallel** using 4 threads. The number of threads can be varied to reduce compilation time, depending on the available hardware.



## Dependencies

The dependencies result from the choice of packages. Examples:

MPI	—	Message Passing Interface <sup>2</sup>
BLAS	—	Basic Linear Algebra Subprograms <sup>3</sup>
LAPACK	—	Linear Algebra PACKage <sup>4</sup>
Boost	—	Peer-reviewed portable C++ libraries <sup>5</sup>
METIS & ParMETIS	—	Graph Partitioning <sup>6</sup>
HDF5	—	Hierarchical Data Format <sup>7</sup>
MUMPS	—	MULTifrontal Massively Parallel sparse direct Solver <sup>8</sup>
⋮		⋮

---

<sup>2</sup><https://www.mpi-forum.org/>

<sup>3</sup><http://www.netlib.org/blas/>

<sup>4</sup><http://www.netlib.org/lapack/>

<sup>5</sup><https://www.boost.org/>

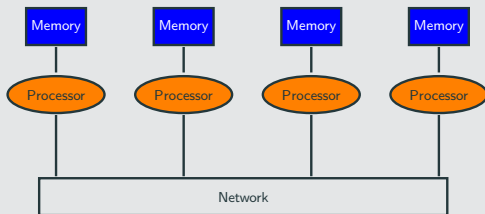
<sup>6</sup><http://glaros.dtc.umn.edu/gkhome/views/metis>

<sup>7</sup><https://www.hdfgroup.org/solutions/hdf5/>

<sup>8</sup><http://mumps.enseiht.fr/>

## Distributed-memory parallelization

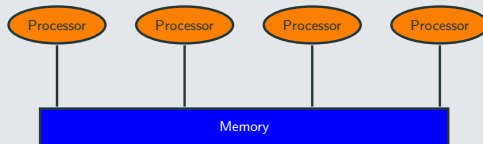
- **Process**-based parallelization
- **Each processor** has its **own internal memory**
- ⊕ No memory access conflicts
- ⊖ Requires (*possibly slow*) data exchange through a network



Using **MPI**

## Shared-memory parallelization

- **Thread**-based parallelization
- **All processors** access a **shared memory**
- ⊕ Changes in shared memory are visible to all
- ⊖ Memory access conflicts



Using

CPU	GPU
OpenMP	CUDA
Pthreads	

## Distributed-memory parallelization (MPI)

**MPI parallelization** is provided through the *parallel linear algebra framework*:

- At the moment, there are two different linear algebra frameworks/packages, the older **Epetra** package and the more recent **Tpetra** package.
- The linear algebra frameworks both provide parallel implementations of
  - vectors,
  - sparse matrices,
  - redistributors,
  - and more...
- Based on Epetra and Tpetra, Trilinos currently provides two stacks of packages, providing a similar range of functionality.
- Tpetra is built upon Kokkos; see right.

## Shared-memory parallelization (X)

A systematic framework for **shared-memory parallelization** is provided by the Kokkos programming model:

- **Kokkos** implements a programming model in C++ for writing performance portable applications targeting all major HPC platforms.
- **KokkosKernels** implements local computational kernels for linear algebra and graph operations, using the Kokkos programming model.
- Support for CUDA, HPX, OpenMP and Pthreads.
- Tpetra automatically provides access to the functionality of Kokkos.

# Overview – Trilinos Packages

	MPI (Epetra-based)	MPI+X (Tpetra-based)
Linear algebra	<b>Epetra &amp; EpetraExt</b>	<b>Tpetra</b>
Direct sparse solvers	<b>Amesos</b>	<b>Amesos2</b>
Iterative solvers	<b>AztecOO</b>	<b>Belos</b>
Preconditioners: <ul style="list-style-type: none"><li>• One-level (incomplete) factorization</li><li>• Multigrid</li><li>• Domain decomposition</li></ul>	<b>IFPACK</b> <b>ML</b>	<b>Ifpack2</b> <b>MueLu</b> <b>ShyLU</b>
Eigenproblem solvers		<b>Anasazi</b>
Nonlinear solvers	<b>NOX &amp; LOCA</b>	
Partitioning	<b>Isorropia &amp; Zoltan</b>	<b>Zoltan2</b>
Example problems	<b>Galeri</b>	
Performance portability		<b>Kokkos &amp; KokkosKernels</b>
Interoperability	<b>Stratimikos &amp; Thyra</b>	
Tools	<b>Teuchos</b>	
⋮	⋮	⋮

More details on <https://trilinos.github.io>.

# FROSch Domain Decomposition Solver Framework

How I am involved in Trilinos:



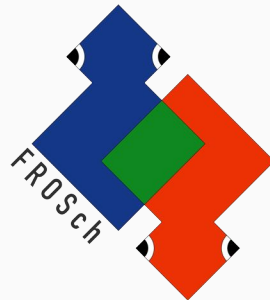
Lead: Sandia National  
Laboratories

- Teko: **Block preconditioners** for multi-physics problems
- Ifpack/Ifpack2: **One-level overlapping Schwarz preconditioners**  
→ **Algebraic** but **not scalable**
- ShyLU/BDDC: **BDDC** (Balancing Domain Decomposition by Constraints)  
**preconditioner**  
→ **Scalable** but **less algebraic**

FROSch (Fast and Robust Overlapping Schwarz)

- **Schwarz preconditioners** with **algebraic coarse spaces** based on extension operators, e.g., **GDSW** (Generalized–Dryja–Smith–Widlund) coarse spaces  
→ **Algebraic** and **scalable**
- Part of the package ShyLU:  
(Joint work with the Scalable Algorithms group of the **Sandia National Laboratories (SNL)**, Albuquerque, USA)
- Implementation based on Xpetra  
→ Can be used with Epetra and Tpetra (linear algebra packages)  
*Extension to current architectures, e.g., GPUs, using the Kokkos programming model*

Easy access to FROSch through unified Trilinos solver interface Thyra.



## Remainder of the presentation

Focus on an **introduction to the Tpetra linear algebra package with respect to distributed-memory (MPI) parallelization.**

## Out of the scope

An introduction to all Trilinos packages including **shared-memory (X) parallelization using Kokkos.**

**Before working with Trilinos**, please also take a look at the Teuchos package! It provides **many useful tools** and is used all over the Trilinos code.

- **Memory management** (e.g., Teuchos::RCP **smart pointers** or Teuchos::Array **arrays with additional functionality**)  
*(very helpful to replace many standard C++ data types and containers)*
- **Parameter lists**  
*(very helpful for handling parameters for functions, classes, or whole programs)*
- **Communication** (e.g., Teuchos::Comm)  
*(See [https://docs.trilinos.org/dev/packages/teuchos/doc/html/classTeuchos\\_1\\_1Comm.html](https://docs.trilinos.org/dev/packages/teuchos/doc/html/classTeuchos_1_1Comm.html))*
- **Numerics** (e.g., BLAS and LAPACK wrappers)
- **Output support, exception handling, unit testing support**, and much more ...

→ Teuchos Doxygen documentation:

<https://docs.trilinos.org/dev/packages/teuchos/doc/html/>

## Important classes:

<code>Tpetra::Map</code>	<b>Parallel distributions:</b> Contains information used to distribute vectors, matrices, and other objects
<code>Tpetra::Vector</code> & <code>Tpetra::MultiVector</code>	<b>Distributed dense vectors:</b> Provides vector services such as scaling, norms, and dot products.
<code>Tpetra::Operator</code>	<b>Base class for linear operators:</b> Abstract interface for operators (e.g., matrices and preconditioners).
<code>Tpetra::RowMatrix</code>	<b>Distributed sparse matrices:</b> An abstract interface for row-distributed sparse matrices; derived from <code>Tpetra::Operator</code> .
<code>Tpetra::CrsMatrix</code>	<b>Distributed sparse matrices:</b> Specific implementation of <code>Tpetra::RowMatrix</code> , utilizing compressed row storage (CRS) format
<code>Tpetra::Import</code> & <code>Tpetra::Export</code>	<b>Import/Export classes:</b> Allow efficient transfer of objects built using one mapping to a new object with a new mapping.

→ Tpetra Doxygen documentation: <https://docs.trilinos.org/dev/packages/tpetra/doc/html/>



- The parallel linear algebra objects from Tpetra are **distributed based on the rows**.
- **Example:** Consider the case of a vector  $V \in \mathbb{R}^5$  and a sparse matrix  $A \in \mathbb{R}^{5 \times 5}$

$$V = \begin{bmatrix} v \\ w \\ x \\ y \\ z \end{bmatrix} \quad A = \begin{bmatrix} a & b & & & \\ c & d & e & & \\ & f & g & h & \\ & & i & j & k \\ & & & l & m \end{bmatrix}$$

distributed among two parallel processes:

$$V = \begin{bmatrix} v \\ w \\ x \\ y \\ z \end{bmatrix} \quad A = \begin{bmatrix} a & b & & & \\ c & d & e & & \\ & f & g & h & \\ & & i & j & k \\ & & & l & m \end{bmatrix} \quad \begin{array}{l} \text{proc 0} \\ \text{proc 1} \end{array}$$

- This can be implemented by storing the *local portions of the vector and the matrix*:

$$\begin{array}{l}
 V_0 = \begin{bmatrix} v \\ x \\ z \end{bmatrix} \quad A_0 = \begin{bmatrix} a & b & & & \\ & f & g & h & \\ & & & l & m \end{bmatrix} \quad \text{proc 0} \\
 V_1 = \begin{bmatrix} w \\ y \end{bmatrix} \quad A_1 = \begin{bmatrix} c & d & e & & \\ & & i & j & k \end{bmatrix} \quad \text{proc 1}
 \end{array}$$

**Problem:** If only the partitioned data is available on the processes, the global vector  $V$  and matrix  $A$  cannot be restored. In particular, it is not clear where the local rows are located in the global matrix.

- Therefore, we additionally store the **global row indices corresponding to the local rows**, here denoted as  $M_0$  and  $M_1$  (local-to-global map):

$$\begin{array}{l}
 V_0 = \begin{bmatrix} v \\ x \\ z \end{bmatrix} \quad A_0 = \begin{bmatrix} a & b & & & \\ & f & g & h & \\ & & & l & m \end{bmatrix} \quad M_0 = \begin{bmatrix} 0 \\ 2 \\ 4 \end{bmatrix} \quad \text{proc 0} \\
 V_1 = \begin{bmatrix} w \\ y \end{bmatrix} \quad A_1 = \begin{bmatrix} c & d & e & & \\ & & i & j & k \end{bmatrix} \quad M_1 = \begin{bmatrix} 1 \\ 3 \end{bmatrix} \quad \text{proc 1}
 \end{array}$$

- Using the local-to-global map, the global objects are fully specified. **Process 0:**

$$\begin{array}{l}
 V_0 = \begin{bmatrix} v \\ x \\ z \end{bmatrix} \quad A_0 = \begin{bmatrix} a & b & & & & \\ & f & g & h & & \\ & & & & l & m \end{bmatrix} \quad M_0 = \begin{bmatrix} 0 \\ 2 \\ 4 \end{bmatrix} \quad \text{proc 0} \\
 \rightarrow V_0 = \begin{bmatrix} v \\ x \\ z \end{bmatrix} \quad A_0 = \begin{bmatrix} a & b & & & & \\ f & g & h & & & \\ & & & & l & m \end{bmatrix}
 \end{array}$$

- Process 1:**

$$\begin{array}{l}
 V_1 = \begin{bmatrix} w \\ y \end{bmatrix} \quad A_1 = \begin{bmatrix} c & d & e & & & \\ & & & i & j & k \end{bmatrix} \quad M_1 = \begin{bmatrix} 1 \\ 3 \end{bmatrix} \quad \text{proc 1} \\
 \rightarrow V_1 = \begin{bmatrix} w \\ y \end{bmatrix} \quad A_1 = \begin{bmatrix} c & d & e & & & \\ & & & i & j & k \end{bmatrix}
 \end{array}$$

- In summary, in addition to the **local portions of the global Tpetra objects**, **local-to-global mappings** are necessary to describe parallel distributed global objects:

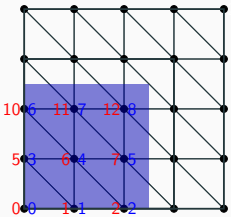
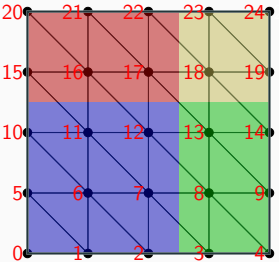
$$V = \begin{bmatrix} v \\ w \\ x \\ y \\ z \end{bmatrix} \quad A = \begin{bmatrix} a & b & & & & \\ c & d & e & & & \\ & f & g & h & & \\ & & i & j & k & \\ & & & l & m & \end{bmatrix}$$

proc 0  
proc 1

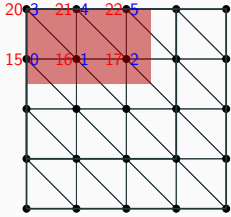
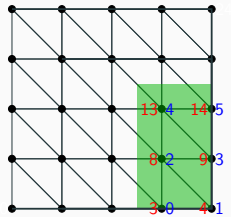
- The local-to-global mappings are stored in `Tpetra::Map` objects.

See [https://docs.trilinos.org/dev/packages/tpetra/doc/html/classTpetra\\_1\\_1Map.html](https://docs.trilinos.org/dev/packages/tpetra/doc/html/classTpetra_1_1Map.html) for more details.

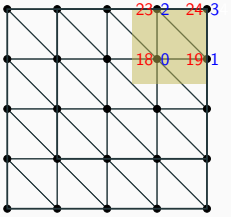
# Tpetra::Map – Exemplary Map/Distribution for a Mesh



global indices

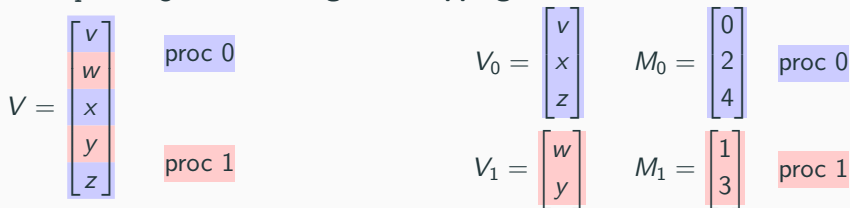


local indices



As previously shown, a **parallel distributed vector** (`Tpetra::Vector`) essentially corresponds to

- arrays containing the **local portions of the vectors** (entries) and
- a `Tpetra::Map` storing the **local-to-global mapping**.



Constructor:

```
Vector ( const Teuchos::RCP< const map_type > &map, /* optional */ )
```

`map`

`Tpetra::Map` object specifying the parallel distribution of the `Tpetra::Vector`. The map also defines the length (local and global) of the vector.

- The `Tpetra::MultiVector` allows for the construction of **multiple vectors with the same parallel distribution**:

$$V = \begin{bmatrix} v_{11} & \cdots & v_{1m} \\ v_{21} & \cdots & v_{2m} \\ \vdots & \ddots & \vdots \\ v_{(n-1)1} & \cdots & v_{(n-1)m} \\ v_{n1} & \cdots & v_{nm} \end{bmatrix} \in \mathbb{R}^{n \times m} \quad \text{with } n \gg m$$

- A typical use case would be a **linear equation system with multiple right hand sides**:

$$AX = B$$

with  $A \in \mathbb{R}^{n \times n}$ ,  $X \in \mathbb{R}^{n \times m}$ , and  $B \in \mathbb{R}^{n \times m}$ . Here,  $A$  would typically be a sparse matrix and  $X$  and  $B$  multivectors.

- It can also be used to implement **skinny dense matrices**.

→ When constructing a `Tpetra::MultiVector` object, the number of vectors has to be specified.

As previously shown, a **parallel distributed sparse matrix** (`Tpetra::CrsMatrix`) essentially corresponds to

- the **local portions** of the sparse matrix and
- a `Tpetra::Map` storing the **local-to-global mapping** corresponding to the rows.

$$A = \begin{bmatrix} a & b & & & & & & & \\ c & d & e & & & & & & \\ & f & g & h & & & & & \\ & & & i & j & k & & & \\ & & & & & l & m & & \end{bmatrix} \quad \begin{array}{l} \text{proc 0} \\ \text{proc 1} \end{array}$$
$$A_0 = \begin{bmatrix} a & b & & & & & & & \\ & f & g & h & & & & & \\ & & & & l & m & & & \end{bmatrix} \quad M_0 = \begin{bmatrix} 0 \\ 2 \\ 4 \end{bmatrix} \quad \text{proc 0}$$
$$A_1 = \begin{bmatrix} c & d & e & & & & & & \\ & & & i & j & k & & & \end{bmatrix} \quad M_1 = \begin{bmatrix} 1 \\ 3 \end{bmatrix} \quad \text{proc 1}$$

In the `Tpetra::CrsMatrix`, the local portions of the sparse matrix are stored in *compressed row storage (CRS) format*.

Minimal constructor:

```
CrsMatrix ( const Teuchos::RCP< const map_type > &rowMap,  
           const size_t maxNumEntriesPerRow, /* optional */ )
```

`rowMap`

Parallel distribution of the rows

`maxNumEntriesPerRow`

Maximum number of nonzero entries per row



## Tpetra::CrsMatrix – Column Map

- In addition to the row map, which corresponds to the local-to-global mapping of the row indices, e.g.,

$$A = \begin{bmatrix} a & b & & & & & \\ c & d & e & & & & \\ & f & g & h & & & \\ & & i & j & k & & \\ & & & l & m & o & \\ & & & & p & q & \end{bmatrix}$$

$$M_0 = \begin{bmatrix} 0 \\ 1 \end{bmatrix} \quad \text{proc 0}$$
$$M_1 = \begin{bmatrix} 2 \\ 3 \end{bmatrix} \quad \text{proc 1}$$
$$M_2 = \begin{bmatrix} 4 \\ 5 \end{bmatrix} \quad \text{proc 2}$$

there is also **local-to-global mapping for the column indices**, the *column map*.

- If the column map is not specified at the construction of the matrix, it can be generated automatically by the Tpetra::CrsMatrix object at a later point.

$$A = \begin{bmatrix} a & b & c & d & e \\ f & g & h & i & j & k \\ l & m & o & p & q \end{bmatrix}$$

$$M_0 = \begin{bmatrix} 0 \\ 1 \end{bmatrix} \quad \text{proc 0}$$

$$M_1 = \begin{bmatrix} 2 \\ 3 \end{bmatrix} \quad \text{proc 1}$$

$$M_1 = \begin{bmatrix} 4 \\ 5 \end{bmatrix} \quad \text{proc 2}$$

A compatible *column map* would corresponding to this *row map* would be:

$$A = \begin{bmatrix} a & b & c & d & e \\ f & g & h & i & j & k \\ l & m & o & p & q \end{bmatrix}$$

$$\tilde{M}_0 = \begin{bmatrix} 0 \\ 1 \\ 2 \end{bmatrix} \quad \text{proc 0}$$

$$\tilde{M}_1 = \begin{bmatrix} 2 \\ 3 \\ 4 \\ 5 \end{bmatrix} \quad \text{proc 1}$$

$$\tilde{M}_2 = \begin{bmatrix} 3 \\ 4 \\ 5 \end{bmatrix} \quad \text{proc 2}$$

- Column maps are **generally not unique**, as in our example:

$$A = \begin{bmatrix} a & b & & & & & \\ c & d & e & & & & \\ & f & g & h & & & \\ & & i & j & k & & \\ & & & l & m & o & \\ & & & & p & q & \end{bmatrix}$$

$$\tilde{M}_0 = \begin{bmatrix} 0 \\ 1 \\ 2 \end{bmatrix} \quad \text{proc 0}$$

$$\tilde{M}_1 = \begin{bmatrix} 2 \\ 3 \\ 4 \\ 5 \end{bmatrix} \quad \text{proc 1}$$

$$\tilde{M}_2 = \begin{bmatrix} 3 \\ 4 \\ 5 \end{bmatrix} \quad \text{proc 2}$$

**Not unique** means that multiple processes share global indices.

## Matrix-vector multiplication:

- As mentioned earlier, the class `Tpetra::CrsMatrix` is derived from `Tpetra::Operator`. Any `Tpetra::Operator` can be applied to a `Tpetra::Vector` or `Tpetra::MultiVector` resulting in another `Tpetra::Vector` or `Tpetra::MultiVector`, respectively.
- The parallel application of any `Tpetra::Operator` is characterized by two maps, the *domain map* and the *range map*.

**domain map** The map of any vector the operator is applied to.

**range map** The map of the resulting vector.

*(Both the domain map and the range map have to be unique!)*

- In particular, for a `Tpetra::CrsMatrix`, the following **very general situation**, where the *row map*, *domain map*, and *range map* are all different, is allowed:

$$\begin{bmatrix} a & b & & & \\ c & d & e & & \\ & f & g & h & \\ & & i & j & k \\ & & & l & m \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} = \begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \\ b_4 \end{bmatrix}$$

- Performing the **matrix-vector multiplication**

$$\begin{bmatrix} a & b & & & \\ c & d & e & & \\ & f & g & h & \\ & & i & j & k \\ & & & l & m \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} = \begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \\ b_4 \end{bmatrix}$$

will obviously **require communication**.

- The corresponding **communication is performed automatically**. However, the *domain map* and *range map* must have already been specified before application to a vector.
- The *domain map* and *range map* can be specified within the `fillComplete()` call.
- If they are not specified, they will automatically be chosen as the *row map* of the matrix:

$$\begin{bmatrix} a & b & & & \\ c & d & e & & \\ & f & g & h & \\ & & i & j & k \\ & & & l & m \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} = \begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \\ b_4 \end{bmatrix}$$

**Caution:** In contrast to the *domain map* and *range map*, the *row map* does not have to be unique.

- Trilinos **GitHub repository**: <https://github.com/trilinos/Trilinos>
- Trilinos **website**: <https://trilinos.github.io/index.html>
  - **Documentation**: <https://trilinos.github.io/documentation.html>
  - Each package has its own **Doxygen documentation**: For instance, Tpetra: <https://docs.trilinos.org/dev/packages/tpetra/doc/html/index.html>
  - **Getting started**: [https://trilinos.github.io/getting\\_started.html](https://trilinos.github.io/getting_started.html)
- Trilinos **hands-on tutorials**:  
[https://github.com/trilinos/Trilinos\\_tutorial/wiki/TrilinosHandsOnTutorial](https://github.com/trilinos/Trilinos_tutorial/wiki/TrilinosHandsOnTutorial)
- Kokkos resources on GitHub: <https://github.com/kokkos>

**Are there any questions at this point?**

**Remainder of the time:** Short demonstration.

## **Additional Slides**

## Reminder: Compressed row storage sparse matrix format

The **C**ompressed **R**ow **S**torage storage format does not require any knowledge about the structure of the matrix and is very general purpose. As the name implies, one compresses row by row.

### Definition 1 (CRS format)

The **compressed row storage (CRS) format** for a matrix  $A$  with  $n$  rows and  $m$  non-zero entries is defined by two 1D arrays  $val$  and  $col\_ind$  of length  $m$  and another array  $row\_ptr$  of length  $n + 1$ .

Only the  $m$  non-zero entries of  $A$  are written row-by-row in  $val$ , and the corresponding column indices are written in  $col\_ind$ .  $row\_ptr[i]$  points to the first entry of the  $i$ -th row in  $val$ , where the last entry of  $row\_ptr$  points to the first entry in the fictitious  $n + 1$ -th row.



## Example 2

For the matrix

$$A = \begin{pmatrix} 3 & 0 & 1 & 0 & -4 \\ 0 & 0 & 1.3 & 0 & 0 \\ 0 & 7 & 0 & 0 & 6.4 \\ 0 & 0 & -1 & 1 & 0 \\ 3.2 & 0 & 0 & 12 & 0 \end{pmatrix}$$

we obtain the following CRS matrix format (with 0 index base)

val	3	1	-4	1.3	7	6.4	-1	1	3.2	12
col_ind	0	2	4	2	1	4	2	3	0	3

row_ptr	0	3	4	6	8	10
---------	---	---	---	---	---	----

**Remark:** There also exist other sparse matrix format, e.g., the CCS (compressed columns storage) format. We will restrict ourselves to the CSR format because it is used in the parallel `Tpetra::CrsMatrix` matrix class.