SANDIA REPORT

SAND2016-5338
Unlimited Release
Printed June 2016

Ifpack2 User's Guide 1.0 (Trilinos version 12.6)

Andrey Prokopenko, Christopher M. Siefert, Jonathan J. Hu,
Mark Hoemmen, Alicia Klinvex

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

Sandia National Laboratories is a multi-program laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000.

Sandia National Laboratories

Issued by Sandia National Laboratories, operated for the United States Department of Energy by Sandia Corporation.

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

SAND2016-5338
Unlimited Release
Printed June 2016

Ifpack2 User's Guide 1.0 (Trilinos version 12.6)

Andrey Prokopenko
Scalable Algorithms
Sandia National Laboratories
Mailstop 1318
P.O. Box 5800
Albuquerque, NM 87185-1318
aprokop@sandia.gov

Christopher Siefert
Computational Math & Algorithms
Sandia National Laboratories
Mailstop 1318
P.O. Box 5800
Albuquerque, NM 87185-1318

Jonathan J. Hu
Scalable Algorithms
Sandia National Laboratories
Mailstop 9159
P.O. Box 0969
Livermore, CA 94551-0969
jhu@sandia.gov

Mark Hoemmen
Scalable Algorithms
Sandia National Laboratories
Mailstop 1320
P.O. Box 5800
Albuquerque, NM 87185-1318
mhoemme@sandia.gov

Alicia Klinvex
Scalable Algorithms
Sandia National Laboratories
Mailstop 1320
P.O. Box 5800
Albuquerque, NM 87185-1318
amklinv@sandia.gov

Abstract

This is the definitive user manual for the IFPACK2 package in the Trilinos project. IFPACK2 provides implementations of iterative algorithms (e.g., Jacobi, SOR, additive Schwarz) and processor-based incomplete factorizations. IFPACK2 is part of the Trilinos TPETRA solver stack, is templated on index, scalar, and node types, and leverages node-level parallelism indirectly through its use of TPETRA kernels. IFPACK2 can be used to solve to matrix systems with greater than 2 billion rows (using 64-bit indices). *Any options not documented in this manual should be considered strictly experimental*.

Acknowledgment

Many people have helped develop IFPACK2, and we would like to acknowledge their contributions here: Ross Bartlett, Tom Benson, Erik Boman, Joshua Booth, Julian Cortial, Kevin Deweese, Jeremie Gaidamour, Paul Lin, Travis Fisher, Sarah Osborn, Eric Phipps, and Paul Tsuji. Finally, Alan Williams did the original port from IFPACK and was the original lead developer of IFPACK2.

Contents

Appendix

List of Figures

List of Tables

Chapter 1

Getting Started

This section is meant to get you using IFPACK2 as quickly as possible. §1.1 gives a brief overview of IFPACK2. §1.2 lists IFPACK2's dependencies on other TRILINOS libraries and provides a sample cmake configuration line. Finally, some examples of code are given in §1.4.

1.1 Overview of IFPACK2

IFPACK2 is a C++ linear solver library in the TRILINOS project [6]. It originally began as a migration of IFPACK package capabilities to a new linear algebra stack. While it retains some commonalities with the original package, it has since diverged significantly from it and should be treated as completely independent package.

IFPACK2 only works with TPETRA [8] matrix, vector, and map types. Like TPETRA, it allows for different ordinal (index) and scalar types. IFPACK2 was designed to be efficient on a wide range of computer architectures, from workstations to supercomputers [10]. It relies on the "MPI+X" principle, where "X" can be threading or CUDA. The "X" portion, node-level parallelism, is controlled by a node template type. Users should refer to TPETRA's documentation for information about node and device types.

IFPACK2 provides a number of different solvers, including

- Jacobi, Gauss-Seidel, polynomial, distributed relaxation;
- domain decomposition solvers;
- incomplete factorizations.

This list of solvers is not exhaustive. Instead, references for further information are provided throughout the text. There are many excellent references for iterative methods, including [14].

Complete information on available capabilities and options can be found in §2.

1.2 Configuration and Build

IFPACK2 requires a C++11 compatible compiler for compilation. The minimum required version of compilers are GCC (4.7.2 and later), Intel (13 and later), and clang (3.5 and later).

1.2.1 Dependencies

Table 1.1 enumerates the dependencies of IFPACK2. Certain dependencies are optional, whereas others are required. Furthermore, IFPACK2's tests depend on certain libraries that are not required if you only want to link against the IFPACK2 library and do not want to compile its tests. Additionally, some functionality in IFPACK2 may depend on other Trilinos packages (for instance, AMESOS2) that may require additional dependencies. We refer to the documentation of those packages for a full list of dependencies.

Dependency	Library		Testing	
	Required	Optional	Required	Optional
TEUCHOS	×		×	
TPETRA	×		×	
TPETRAKERNELS	×			
AMESOS2		×		×
GALERI				×
XPETRA		×		×
ZOLTAN2		×		×
THYRATPETRAADAPTERS		×		
SHYLUHTS		×		×
MPI		×		×

Table 1.1. IFPACK2's required and optional dependencies, subdivided by whether a dependency is that of the IFPACK2 library itself (*Library*), or of some IFPACK2 test (*Testing*).

AMESOS2 and SUPERLU are necessary if you want to use either a sparse direct solve or ILUTP as a subdomain solve in processor-based domain decomposition. ZOLTAN2 and XPETRA are necessary if you want to reorder a matrix (e.g., reverse Cuthill McKee).

1.2.2 Configuration

The preferred way to configure and build IFPACK2 is to do that outside of the source directory. Here we provide a sample configure script that will enable IFPACK2 and all of its optional

dependencies:

```
export TRILINOS_HOME=/path/to/your/Trilinos/source/directory
cmake -D BUILD_SHARED_LIBS:BOOL=ON \
      -D CMAKE_BUILD_TYPE:STRING="RELEASE" \
      -D CMAKE_CXX_FLAGS:STRING="-g" \
      -D Trilinos_ENABLE_EXPLICIT_INSTANTIATION:BOOL=ON \
      -D Trilinos_ENABLE_TESTS:BOOL=OFF \
      -D Trilinos_ENABLE_EXAMPLES:BOOL=OFF \
      -D Trilinos_ENABLE_Ifpack2:BOOL=ON \
      -D Ifpack2_ENABLE_TESTS:STRING=ON \
      -D Ifpack2_ENABLE_EXAMPLES:STRING=ON \
      -D TPL_ENABLE_BLAS:BOOL=ON \
      -D TPL_ENABLE_MPI:BOOL=ON \
      ${TRILINOS_HOME}
```

More configure examples can be found in `Trilinos/sampleScripts`. For more information on configuring, see the TRILINOS Cmake Quickstart guide [1].

1.3 Interface to IFPACK2 methods

All IFPACK2 operators inherit from the base class `Ifpack2::Preconditioner`. This in turn inherits from `Tpetra::Operator`. Thus, you may use an IFPACK2 operator anywhere that a `Tpetra::Operator` works. For example, you may use IFPACK2 operators directly as preconditioners in TRILINOS' BELOS package of iterative solvers.

You may either create an IFPACK2 operator directly, by using the class and options that you want, or by using `Ifpack2::Factory`. Some of IFPACK2 preconditioners only accept a `Tpetra::CrsMatrix` instance as input, while others also may accept a `Tpetra::RowMatrix` (the base class of `Tpetra::CrsMatrix`). They will decide at run time whether the input `Tpetra::RowMatrix` is an instance of the right subclass.

`Ifpack2::Preconditioner` includes the following methods:

- `initialize()`

  Performs all operations based on the graph of the matrix (without considering the numerical values).

- `compute()`

  Computes everything required to apply the preconditioner, using the matrix's values.

- `apply()`

  Applies or "solves with" the preconditioner.

Every time that `initialize()` is called, the object destroys all the previously allocated information, and reinitializes the preconditioner. Every time `compute()` is called, the object recomputes the actual values of the preconditioner.

An IFPACK2 preconditioner may also inherit from `Ifpack2::CanChangeMatrix` class in order to express that users can change its matrix (the matrix that it preconditions) after construction using a `setMatrix` method. Changing the matrix puts the preconditioner back in an "pre-initialized" state. You must first call `initialize()`, then `compute()`, before you may call `apply()` on this preconditioner. Depending on the implementation, it may be legal to set the matrix to null. In that case, you may not call `initialize()` or `compute()` until you have subsequently set a nonnull matrix.

Warning. If you are familiar with the IFPACK package [16], please be aware that the behaviour of the IFPACK2 preconditioner is different from IFPACK. In IFPACK, the `ApplyInverse()` method applies or "solves with" the preconditioner M^{-1}, and the `Apply()` method "applies" the preconditioner M. In IFPACK2, the `apply()` method applies or "solves with" the preconditioner M^{-1}. IFPACK2 has no method comparable to IFPACK's `Apply()`.

1.4 Example: IFPACK2 preconditioner within BELOS

The most commonly used scenario involving IFPACK2 is using one of its preconditioners preconditioners inside an iterative linear solver. In TRILINOS, the BELOS package provides important Krylov subspace methods (such as preconditioned CG and GMRES).

At this point, we assume that the reader is comfortable with TEUCHOS referenced-counted pointers (RCPs) for memory management (an introduction to RCPs can be found in [3]) and the `Teuchos::ParameterList` class [17].

First, we create an IFPACK2 preconditioner using a provided `Teuchos::ParameterList`

```
typedef Tpetra::CrsMatrix<Scalar, LocalOrdinal, GlobalOrdinal, Node>
  Tpetra_Operator;

Teuchos::RCP<Tpetra_Operator> A;
// create A here ...
Teuchos::ParameterList paramList;
paramList.set( "chebyshev: degree", 1 );
paramList.set( "chebyshev: min eigenvalue", 0.5 );
paramList.set( "chebyshev: max eigenvalue", 2.0 );
// ...
Ifpack2::Factory factory;
RCP<Ifpack2::Ifpack2Preconditioner<> > ifpack2Preconditioner;
ifpack2Preconditioner = factory.create( "CHEBYSHEV", A )
ifpack2Preconditioner->setParameters( paramList );
ifpack2Preconditioner->initialize();
```

```
ifpack2Preconditioner->compute();
```

Besides the linear operator A, we also need an initial guess vector for the solution X and a right hand side vector B for solving a linear system.

```
typedef Tpetra::Map<LocalOrdinal, GlobalOrdinal, Node> Tpetra_Map;
typedef Tpetra::MultiVector<Scalar, LocalOrdinal, GlobalOrdinal, Node>
  Tpetra_MultiVector;

Teuchos::RCP<const Tpetra_Map> map = A->getDomainMap();

// create initial vector
Teuchos::RCP<Tpetra_MultiVector> X =
  Teuchos::rcp( new Tpetra_MultiVector(map, numrhs) );

// create right-hand side
X->randomize();
Teuchos::RCP<Tpetra_MultiVector> B =
  Teuchos::rcp( new Tpetra_MultiVector(map, numrhs) );
A->apply( *X, *B );
X->putScalar( 0.0 );
```

To generate a dummy example, the above code first declares two vectors. Then, a right hand side vector is calculated as the matrix-vector product of a random vector with the operator A. Finally, an initial guess is initialized with zeros.

Then, one can define a `Belos::LinearProblem` object where the `ifpack2Preconditioner` is used for left preconditioning.

```
typedef Belos::LinearProblem<Scalar, Tpetra_MultiVector, Tpetra_Operator>
  Belos_LinearProblem;

Teuchos::RCP<Belos_LinearProblem> problem =
  Teuchos::rcp( new Belos_LinearProblem( A, X, B ) );
problem->setLeftPrec( ifpack2Preconditioner );
bool set = problem.setProblem();
```

Next, we set up a BELOS solver using some basic parameters.

```
Teuchos::RCP<Teuchos::ParameterList> belosList =
  Teuchos::rcp(new Teuchos::ParameterList);
belosList->set( "Block Size", 1 );
belosList->set( "Maximum Iterations", 100 );
belosList->set( "Convergence Tolerance", 1e-10 );
belosList->set( "Output Frequency", 1 );
belosList->set( "Verbosity", Belos::TimingDetails + Belos::FinalSummary );

Belos::SolverFactory<Scalar, Tpetra_MultiVector, Tpetra_Operator> solverFactory;
```

```
Teuchos::RCP<Belos::SolverManager<Scalar, Tpetra_MultiVector, Tpetra_Operator> >
  solver = solverFactory.create( "Block CG", belosList );
solver->setProblem( problem );
```

Finally, we solve the system.

```
Belos::ReturnType ret = solver.solve();
```

It is often more convenient to specify the parameters as part of an XML-formatted options file. Look in the subdirectory `Trilinos/packages/ifpack2/test/belos` for examples of this.

This section is only meant to give a brief introduction on how to use IFPACK2 as a preconditioner within the TRILINOS packages for iterative solvers. There are other, more complicated, ways to use to work with IFPACK2. For more information on these topics, the reader may refer to the examples and tests in the IFPACK2 source directory (`Trilinos/packages/ifpack2`).

Chapter 2

IFPACK2 options

In this section, we report the complete list of input parameters. Input parameters are passed to IFPACK2 in a single `Teuchos::ParameterList`.

In some cases, the parameter types may depend on runtime template parameters. In such cases, we will follow the conventions in Table 2.1.

`MatrixType::local_ordinal_type`	`local_ordinal`
`MatrixType::global_ordinal_type`	`global_ordinal`
`MatrixType::scalar_type`	`scalar`
`MatrixType::node_type`	`node`
`Tpetra::Vector<scalar,local_ordinal,global_ordinal,node>`	`vector`
`Tpetra::MultiVector<scalar,local_ordinal,global_ordinal,node>`	`multi_vector`
`vector::mag_type`	`magnitude`

Table 2.1. Conventions for option types that depend on templates.

Note: if `scalar` is `double`, then `magnitude` is also `double`.

2.1 Point relaxation

Preconditioner type: "RELAXATION".

IFPACK2 implements the following classical relaxation methods: Jacobi (with optional damping), Gauss-Seidel, Successive Over-Relaxation (SOR), symmetric version of Gauss-Seidel and SOR. IFPACK2 calls both Gauss-Seidel and SOR "Gauss-Seidel". The algorithmic details can be found in [14].

Besides the classical relaxation methods, IFPACK2 also implements l_1 variants of Jacobi and Gauss-Seidel methods proposed in [2], which lead to a better performance in parallel applications.

Note: if a user provides a `Tpetra::BlockCrsMatrix`, the point relaxation methods become block

relaxation methods, such as block Jacobi or block Gauss-Seidel.

The following parameters are used in the point relaxation methods:

`"relaxation: type"`	[`string`] Relaxation method to use. Accepted values: "Jacobi", "Gauss-Seidel", "Symmetric Gauss-Seidel". **Default:** "Jacobi".
`"relaxation: sweeps"`	[`int`] Number of sweeps of the relaxation. **Default:** 1.
`"relaxation: damping factor"`	[`scalar`] The value of the damping factor ω for the relaxation. **Default:** 1.0.
`"relaxation: backward mode"`	[`bool`] Governs whether Gauss-Seidel is done in forward-mode (`false`) or backward-mode (`true`). Only valid for "Gauss-Seidel" type. **Default:** `false`.
`"relaxation: use l1"`	[`bool`] Use the l_1 variant of Jacobi or Gauss-Seidel. **Default:** `false`.
`"relaxation: l1 eta"`	[`magnitude`] η parameter for l_1 variant of Gauss-Seidel. Only used if `"relaxation: use l1"` is `true`. **Default:** 1.5.
`"relaxation: zero starting solution"`	[`bool`] Governs whether or not IFPACK2 uses existing values in the left hand side vector. If true, IFPACK2 fill it with zeros before applying relaxation sweeps which may make the first sweep more efficient. **Default:** `true`.
`"relaxation: fix tiny diagonal entries"`	[`bool`] If true, the compute() method will do extra work (computation only, no MPI communication) to fix diagonal entries. Specifically, the diagonal values with a magnitude smaller than the magnitude of the threshold `relaxation: min diagonal value` are increased to threshold for the diagonal inversion. The matrix is not modified, instead the updated diagonal values are stored. If the threshold is zero, only the diagonal entries that are exactly zero are replaced with a small nonzero value (machine precision). **Default:** `false`.
`"relaxation: min diagonal value"`	[`scalar`] The threshold value used in `"relaxation: fix tiny diagonal entries"`. Only used if `"relaxation: fix tiny diagonal entries"` is `true`. **Default:** 0.0.

"relaxation: check diagonal entries"	[bool] If true, the compute() method will do extra work (both computation and communication) to count diagonal entries that are zero, have negative real part, or are small in magnitude. This information can be later shown in the description. **Default:** false.
"relaxation: mtgs cluster size"	[int] Only has an effect if "relaxation: type" is "MT Gauss-Seidel" or "MT Symmetric Gauss-Seidel". If equal to 1 (default), point coloring parallel Gauss-Seidel is used. This has a faster compute() but may cause the preconditioned solver to converge more slowly. If set to $k > 1$, then multicolor block Gauss-Seidel is used with blocks of size k (see [15]). In the apply() there is significantly less error due to parallel updates of the LHS vector. **Default:** 1.
"relaxation: mtgs coloring algorithm"	[string] Only has an effect if "relaxation: type" is "MT Gauss-Seidel" or "MT Symmetric Gauss-Seidel". Selects which graph coloring algorithm from Kokkos Kernels will be used to find sets of independent rows. "Default" selects the algorithm based on the Node type. "Serial" is sequential and greedy. "VB", "VBBIT" and "VBCS" are parallel vertex-based (out of these, "VBBIT" is recommended). "EB" is parallel edge-based (faster than "VBBIT" on GPUs for irregular graphs). "VBD" and "VBDBIT" are parallel and deterministic. **Default:** "Default".
"relaxation: local smoothing indices"	[Teuchos::ArrayRCP<local_ordinal>] **Default:** empty.

A given method will only relax on the local indices listed in the ArrayRCP, in the order that they are listed. This can be used to reorder the relaxation, or to only relax on a subset of ids.

2.2 Block relaxation

Preconditioner type: "BLOCK_RELAXATION".

IFPACK2 supports block relaxation methods. Each block corresponds to a set of degrees of freedom within a local subdomain. The blocks can be non-overlapping or overlapping. Block relaxation can be considered as domain decomposition within an MPI process, and should not be confused with additive Schwarz preconditioners (see 2.5) which implement domain decomposition

across MPI processes.

There are several ways the blocks are constructed:

- Linear partitioning of unknowns

  The unknowns are divided equally among a specified number of partitions L defined by `"partitioner: local parts"`. In other words, assuming number of unknowns n is divisible by L, unknown i will belong to block number $\lfloor iL/n \rfloor$.

- Line partitioning of unknowns

  The unknowns are grouped based on a geometric criteria which tries to identify degrees of freedom that form an approximate geometric line. Current approach uses a local line detection inspired by the work of Mavriplis [11] for convection-diffusion. IFPACK2 uses coordinate information provided by `"partitioner: coordinates"` to pick ”close” points if they are sufficiently far away from the ”far” points. It also makes sure the line can never double back on itself.

  These ”line” partitions were found to be very beneficent to problems on highly anisotropic geometries such as ice-sheet simulations.

- User partitioning of unknowns

  The unknowns are grouped according to a user provided partition. A user may provide a non-overlapping partition `"partitioner: map"` or an overlapping one via either `"partitioner: parts"` or `"partitioner: global ID parts"`.

  A particular example of a smoother using this type of approach might be a Vanka smoother [18], where a user may in `"partition: parts"` pressure degrees of freedom, and request a overlap of one thus constructing Vanka blocks.

  When blocks overlap, multiple solutions at each dof are combined for all blocks that contain this dof. When overlapping blocks reside on different MPI ranks, use block relaxation with Schwarz by setting `"subdomain solver name"` to `"BLOCK_RELAXATION"` (see 2.5). `"schwarz: overlap level"` must be sufficient so that each block is fully contained within a subdomain. `"schwarz: combine mode"` determines how overlapped values across MPI ranks are combined. If `"Jacobi"` is used for `"relaxation: type"` when either `"schwarz: overlap level"` is not zero or when using `"partitioner: parts"` or `"partitioner: global ID parts"`, the main overlap combinations are

  - average solutions at each dof from all block solves containing the dof even when blocks reside on different MPI ranks. Set `"schwarz: combine mode"` to `"ADD"`, and `"partitioner: nonsymmetric overlap combine"` to `"true"` to do this.
  - average solutions from all block solves containing the dof residing on MPI rank owning the dof within non-overlapped version of sub-domain (RAS or Restrictive Additive Schwarz). Set `"schwarz: combine mode"` to `"ZERO"` and `"partitioner: nonsymmetric overlap combine"` to `"true"` for this.

– symmetric version of first option when employing CG Krylov solver (i.e., first average block residuals and then average block solutions). Set "schwarz: combine mode" to "ADD" and "partitioner: nonsymmetric overlap combine" to "false".

– symmetric version of second option when employing CG Krylov solver (i.e., average block residuals and block solutions for blocks on owning MPI rank). Set "schwarz: combine mode" to "ZERO" and "partitioner: nonsymmetric overlap combine" to "false".

The original partitioning may be further modified with "partitioner: overlap" parameter which will use the local matrix graph to construct overlapping partitions.

The blocks are applied in the order they were constructed. This means that in the case of overlap the entries in the solution vector relaxed by one block may later be overwritten by relaxing another block.

The following parameters are used in the block relaxation methods:

"relaxation: type"	See 2.1.
"relaxation: container"	string

"TriDi" Containers are used to store and solve block matrices. These container types are always available: "Dense", "TriDi" (equivalent to "Tridiagonal"), "Banded" and "SparseILUT". "Dense", "TriDi" and "Banded" block matrices are solved exactly LAPACK routines, and "SparseILUT" blocks are solved approximately using an incomplete LU factorization with thresholding.

If Amesos2 is enabled, "SparseAmesos" (equivalent to "SparseAmesos2") is available. The default Amesos2 sparse solver is KLU2, but this can be configured by setting "Amesos2 solver name" (see the Amesos2 documentation for all available solvers).

If experimental kokkos-kernels features are enabled (true by default), the "BlockTriDi" container (equivalent to "Block Tridiagonal") is available. This container's solver is the damped Jacobi method, using block tridiagonal matrices as the diagonal D. For a block size of 1, this is equivalent to standard damped Jacobi. This container is designed for high performance on KNL and GPU.

"relaxation: sweeps"	See 2.1.
"relaxation: damping factor"	See 2.1.
"relaxation: zero starting solution"	See 2.1.
"relaxation: backward mode"	See 2.1. Currently has no effect.

"block relaxation: decouple dofs"	[bool] Whether to separate blocks according to the different degrees of freedom (PDEs) at each node. This assumes that dofs/node is constant throughout the matrix. Each block will have the same sparsity pattern as the mesh graph's corresponding diagonal block. For example, when using a line partitioner this enables the use of the tridiagonal container even if the matrix's bandwidth is greater than 3. Decoupling matches the behavior of line smoothing in ML. **Default:** false.
"partitioner: type"	[string] The partitioner to use for defining the blocks. This can be either "linear", "line" or "user". **Default:** "linear".
"partitioner: overlap"	[int] The amount of overlap between partitions (0 corresponds to no overlap). Only valid for "Jacobi" relaxation. **Default:** 0.
"partitioner: local parts"	[int] Number of local partitions (1 corresponds to one local partition, which means "do not partition locally"). Only valid for "linear" partitioner type. **Default:** 1.
"partitioner: map"	[Teuchos::ArrayRCP<local_ordinal>] An array containing the partition number for each element. The *i*th entry in the ArrayRCP is the part (block) number that row *i* belongs to. Use this option if parts (blocks) do not overlap. Only valid for "user" partitioner type. **Default:** empty.
"partitioner: parts"	[Teuchos::Array<Teuchos::ArrayRCP <local_ordinal>>] Use this option if parts (blocks) overlap. The *i*th entry in the Array is an ArrayRCP containing all rows (local IDs) in part (block) *i*. Only valid for "user" partitioner type. **Default:** empty.
"partitioner: global ID parts"	[Teuchos::Array<Teuchos::ArrayRCP <global_ordinal>>] Use this option if parts (blocks) overlap. The *i*th entry in the Array is an ArrayRCP containing all rows (global IDs) in part (block) *i*. Only valid for "user" partitioner type. **Default:** empty.
"partitioner: nonsymmetric overlap combine""	[bool] Valid when using Jacobi relaxation with overlapping blocks. Corresponds to averaging only overlapping solutions as opposed to symmetric averaging which also averages residuals before performing block solves. **Default:** false.

"partitioner: line detection threshold"	[magnitude] Threshold used in line detection. If the distance between two connected points i and j is within the threshold times maximum distance of all points connected to i, then point j is considered close enough to line smooth. Only valid for "line" partition type. **Default:** 0.0.
"partitioner: PDE equations"	[int] Number of equations per node. Only used for "line" partitioning, and decoupled BlockRelaxation. **Default:** 1.
"partitioner: coordinates"	[Teuchos::RCP<multi_vector>] Coordinates of local nodes. Only valid for "line" partitioner type. **Default:** null.
"partitioner: maintain sparsity"	[bool] For OverlappingPartitioner, whether to sort the entries in each partition. **Default:** false.

2.3 Chebyshev

Preconditioner type: "CHEBYSHEV".

IFPACK2 implements a variant of Chebyshev iterative method following IFPACK's implementation. IFPACK has a special-case modification of the eigenvalue bounds for the case where the maximum eigenvalue estimate is close to one. Experiments show that the IFPACK imitation is much less sensitive to the eigenvalue bounds than the textbook version.

IFPACK2 uses the diagonal of the matrix to precondition the linear system on the left. Diagonal elements less than machine precision are replaced with machine precision.

IFPACK2 requires can take any matrix A but can only guarantee convergence for real valued symmetric positive definite matrices.

The following parameters are used in the Chebyshev method:

"chebyshev: degree"	[int] Degree of the Chebyshev polynomial, or the number of iterations. This overrides parameters "relaxation: sweeps" and "smoother: sweeps". **Default:** 1.
"relaxation: sweeps"	Same as "chebyshev: degree", for compatibility with IFPACK.
"smoother: sweeps"	Same as "chebyshev: degree", for compatibility with ML.

`"chebyshev: max eigenvalue"`	[`scalar|double`] An upper bound of the matrix eigenvalues. If not provided, the value will be computed by power method (see parameters `"eigen-analysis: type"` and `"chebyshev: eigenvalue max iterations"`). **Default:** computed.
`"chebyshev: min eigenvalue"`	[`scalar|double`] A lower bound of the matrix eigenvalues. If not provided, IFPACK2 will provide an estimate based on the maximum eigenvalue and the ratio. **Default:** computed.
`"chebyshev: ratio eigenvalue"`	[`scalar|double`] The ratio of the maximum and minimum estimates of the matrix eigenvalues. **Default:** 30.0.
`"smoother: Chebyshev alpha"`	Same as `"chebyshev: ratio eigenvalue"`, for compatibility with ML.
`"chebyshev: compute max residual norm"`	[`bool`] The `apply` call will optionally return the norm of the residual. **Default:** `false`.
`"eigen-analysis: type"`	[`string`] The algorithm for estimating the max eigenvalue. Currently only supports power method (”power-method” or ”power method”). The cost of the procedure is roughly equal to several matrix-vector multiplications. **Default:** ”power-method”.
`"chebyshev: eigenvalue max iterations"`	[`int`] Number of iterations to be used in calculating the estimate for the maximum eigenvalue, if it is not provided by the user. **Default:** 10.
`"eigen-analysis: iterations"`	Same as `"chebyshev: eigenvalue max iterations"`, for compatibility with ML.
`"chebyshev: min diagonal value"`	[`scalar`] Values on the diagonal smaller than this value are increased to this value for the diagonal inversion. **Default:** 0.0.
`"chebyshev: boost factor"`	[`double`] Factor used to increase the estimate of matrix maximum eigenvalue to ensure the high-energy modes are not magnified by a smoother. **Default:** 1.1.
`"chebyshev: assume matrix does not change"`	[`bool`] Whether `compute()` should assume that the matrix has not changed since the last call to `compute()`. If true, `compute()` will not recompute inverse diagonal or eigenvalue estimates. **Default:** `false`.

"chebyshev: operator inv diagonal"	[Teuchos::RCP<const vector>| Teuchos::RCP<vector>|const vector*| vector] If nonnull, a deep copy of this vector will be used as the inverse diagonal of the matrix, instead of computing it. Expert use only. **Default:** Teuchos::null.
"chebyshev: min diagonal value"	[scalar] If any entry of the matrix diagonal is less that this in magnitude, it will be replaced with this value in the inverse diagonal used for left scaling. **Default:** machine precision.
"chebyshev: zero starting solution"	See "relaxation: zero starting solution".

2.4 Incomplete factorizations

2.4.1 ILU(k)

Preconditioner type: "RILUK".

IFPACK2 implements a standard and modified (MILU) variants of the ILU(k) factorization [14]. In addition, it also provides an optional *a priori* modification of the diagonal entries of a matrix to improve the stability of the factorization.

The following parameters are used in the ILU(k) method:

"fact: iluk level-of-fill"	[int|global_ordinal|magnitude|double] Level-of-fill of the factorization. **Default:** 0.
"fact: relax value"	[magnitude|double] MILU diagonal compensation value. Entries dropped during factorization times this factor are added to diagonal entries. **Default:** 0.0.
"fact: absolute threshold"	[magnitude|double] Prior to the factorization, each diagonal entry is updated by adding this value (with the sign of the actual diagonal entry). Can be combined with "fact: relative threshold". The matrix remains unchanged. **Default:** 0.0.

`"fact: relative threshold"`	[`magnitude|double`] Prior to the factorization, each diagonal element is scaled by this factor (not including contribution specified by `"fact: absolute threshold"`). Can be combined with `"fact: absolute threshold"`. The matrix remains unchanged. **Default:** 1.0.

2.4.2 ILUT

Preconditioner type: "ILUT".

IFPACK2 implements a slightly modified variant of the standard ILU factorization with specified fill and drop tolerance ILUT(p, τ) [13]. The modifications follow the AZTECOO implementation. The main difference between the IFPACK2 implementation and the algorithm in [13] is the definition of `fact: ilut level-of-fill`.

The following parameters are used in the ILUT method:

`"fact: ilut level-of-fill"`	[`int|magnitude|double`] Maximum fill fraction that controls the number of entries to keep in the strict upper triangle of the current row, and in the strict lower triangle of the current row. This value must be $>= 1.0$. Each row of L (U) will have at most $\lceil \frac{(\texttt{level-of-fill}-1)nnz(A)}{2n} \rceil$ nonzero entries, in addition to those from the sparsity pattern of A, where $nnz(A)$ is the number of nonzero entries in the matrix, and n is the number of rows. ILUT always keeps the diagonal entry in the current row, regardless of the drop tolerance or fill level. **Note:** *This is different from the p in the classic algorithm in [13].* **Default:** 1.
`"fact: drop tolerance"`	[`magnitude|double`] A threshold for dropping entries (τ in [13]). **Default:** 0.0.
`"fact: absolute threshold"`	See 2.4.1.
`"fact: relative threshold"`	See 2.4.1.
`"fact: relax value"`	Currently has no effect. For backwards compatibility only.

2.4.3 ILUTP

Preconditioner type: “AMESOS2”.

IFPACK2 implements a standard ILUTP factorization [14]. This is done through is through the AMESOS2 interface to SuperLU [9]. We reproduce the AMESOS2 options here for convenience. *You should consider the AMESOS2 documentation to be the final authority.*

The following parameters are used in the ILUTP method:

`"ILU_DropTol"`	[`double`] ILUT drop tolerance. **Default:** 1e-4.
`"ILU_FillFactor"`	[`double`] ILUT fill factor. **Default:** 10.0.
`"ILU_Norm"`	[`string`] Norm to be used in factorization. Accepted values: “ONE_NORM”, “TWO_NORM”, or “INF_NORM”. **Default:** “INF_NORM”.
`"ILU_MILU"`	[`string`] Type of modified ILU to use. Accepted values: “SILU”, “SMILU_1”, “SMILU_2”, or “SMILU_3”. **Default:** “SILU”.

2.4.4 ShyLU FastILU

IFPACK2 provides an interface to the FastILU family of factorizations provided by ShyLU. They are available if Trilinos was configured with the

```
-D Trilinos_ENABLE_ShyLU_Node=ON
```

option. There are three values of “Preconditioner type:” that use the FastILU subpackage:

“Preconditioner type:”	Factorization
FAST_ILU	Incomplete LU
FAST_IC	Incomplete Cholesky
FAST_ILDL	Incomplete LDL*

FAST_ILU, FAST_IC, and FAST_ILDL all use iterative factorization algorithms in compute(). `"sweeps"` controls this iteration count. A higher number of sweeps improves the quality of the factorization. All three preconditioners also use an triangular block Jacobi solver in apply(). The Jacobi iteration count is controlled by `"triangular solve iterations"`. The valid set of parameters is the same for FAST_ILU, FAST_IC, and FAST_ILDL:

`"sweeps"`	[`int`] Number of iterations of ILU/IC/ILDL factorization algorithm. **Default:** 5.

`"triangular solve iterations"`	[`int`] Number of iterations of the block Jacobi triangular solver. **Default:** 1.
`"level"`	[`int`] Level of fill. **Default:** 0.
`"damping factor"`	[`double`] Damping factor ω for the Jacobi triangular solver. $0 < \omega \leq 1$. A lower ω slows convergence but improves stability. **Default:** 0.5.
`"shift"`	[`double`] Manteuffel shifting parameter α. **Default:** 0.
`"guess"`	[`bool`] Whether to run another instance of FastILU/IC/ILDL (but with a lower level of fill) to compute the initial guess (only has an effect if level of fill > 0). **Default:** true.
`"block size"`	[`int`] Block size for the block Jacobi solver. **Default:** 1.

2.5 Additive Schwarz

Preconditioner type: "SCHWARZ".

IFPACK2 implements additive Schwarz domain decomposition with optional overlap. Each subdomain corresponds to exactly one MPI process in the given matrix's MPI communication. For domain decomposition within an MPI process see 2.2.

One-level overlapping domain decomposition preconditioners use local solvers of Dirichlet type. This means that the inverse of the local matrix (possibly with overlap) is applied to the residual to be preconditioned. The preconditioner can be written as:

$$P_{AS}^{-1} = \sum_{i=1}^{M} P_i A_i^{-1} R_i,$$

where M is the number of subdomains (in this case, the number of (MPI) processes in the computation), R_i is an operator that restricts the global vector to the vector lying on subdomain i, P_i is the prolongator operator, and $A_i = R_i A P_i$.

Constructing a Schwarz preconditioner requires defining two components.

Definition of the restriction and prolongation operators. Users may control how the data is combined with existing data by setting `"combine mode"` parameter. Table 2.2 contains a list of modes to combine overlapped entries. The default mode is "ZERO" which is equivalent to using a restricted additive Schwarz [5] method.

Combine mode name	Description
"ADD"	Sum values into existing values
"ZERO"	Replace old values with zero
"INSERT"	Insert new values that don't currently exist
"REPLACE"	Replace existing values with new values
"ABSMAX"	Replace old values with maximum of magnitudes of old and new values

Table 2.2. Combine mode descriptions.

Definition of a solver for subdomain linear system. Some preconditioners may benefit from local modifications to the subdomain matrix. It can be filtered to eliminate singletons and/or re-ordered. Reordering will often improve performance during incomplete factorization setup, and improve the convergence. The matrix reordering algorithms specified in `"schwarz: reordering list"` are provided by ZOLTAN2. At the present time, the only available reordering algorithm is RCM (reverse Cuthill-McKee). Other orderings will be supported by the Zoltan2 package in the future.

To solve linear systems involving A_i on each subdomain, a user can specify the inner solver by setting `"inner preconditioner name"` parameter (or any of its aliases) which allows to use any IFPACK2 preconditioner. These include but are not necessarily limited to the preconditioners in Table 2.3.

Inner solver type	Description
"DIAGONAL"	Diagonal scaling
"RELAXATION"	Point relaxation (see 2.1)
"BLOCK_RELAXATION"	Block relaxation (see 2.2)
"CHEBYSHEV"	Chebyshev iteration (see 2.3)
"RILUK"	ILU(k) (see 2.4.1)
"ILUT"	ILUT (see 2.4.2)
"FAST_ILU"	FastILU (see 2.4.4)
"FAST_IC"	FastIC (see 2.4.4)
"FAST_ILDL"	FastILDL(see 2.4.4)
"AMESOS2"	AMESOS2's interface to sparse direct solvers
"DENSE" or "LAPACK"	LAPACK's LU factorization for a dense representation of a subdomain matrix
"CUSTOM"	User provided inner solver

Table 2.3. Additive Schwarz solver preconditioner types.

The following parameters are used in the Schwarz method:

"schwarz: inner preconditioner name"	[string] The name of the subdomain solver. **Default:** none.
"inner preconditioner name"	Same as "schwarz: inner preconditioner name".
"schwarz: subdomain solver name"	Same as "schwarz: inner preconditioner name".
"subdomain solver name"	Same as "schwarz: inner preconditioner name".
"schwarz: inner preconditioner parameters"	[Teuchos::ParameterList] Parameters for the subdomain solver. If not provided, the subdomain solver will use its specific default parameters. **Default:** empty.
"inner preconditioner parameters"	Same as "schwarz: inner preconditioner parameters".
"schwarz: subdomain solver parameters"	Same as "schwarz: inner preconditioner parameters".
"subdomain solver parameters"	Same as "schwarz: inner preconditioner parameters".
"schwarz: combine mode"	[string] The rule for combining incoming data with existing data in overlap regions. Accepted values: see Table 2.2. **Default:** "ZERO".
"schwarz: overlap level"	[int] The level of overlap (0 corresponds to no overlap). **Default:** 0.
"schwarz: num iterations"	[int] Number of iterations to perform. **Default:** 1.
"schwarz: use reordering"	[bool] If true, local matrix is reordered before computing subdomain solver. TRILINOS must have been built with ZOLTAN2 and XPETRA enabled. **Default:** false.
"schwarz: reordering list"	[Teuchos::ParameterList] Specify options for a ZOLTAN2 reordering algorithm to use. See "order_method". *You should consider the ZOLTAN2 documentation to be the final authority.* **Default:** empty.
"order_method"	[string] Reordering algorithm. Accepted values: "rcm", "minimum_degree", "natural", "random", or "sorted_degree". Only used in "schwarz: reordering list" sublist. **Default:** "rcm".
"schwarz: zero starting solution"	See "relaxation: zero starting solution".

"schwarz: filter singletons"	[bool] If true, exclude rows with just a single entry on the calling process. **Default:** false.
"schwarz: subdomain id"	Currently has no effect.
"schwarz: compute condest"	Currently has no effect. For backwards compatibility only.
"schwarz: update damping"	[double] The amount by which to damp the updates from the Schwarz solve (1.0 is no damping). **Default:** 1.0.

2.6 Hiptmair

IFPACK2 implements Hiptmair algorithm of [7]. The method operates on two spaces: a primary space and an auxiliary space. This situation arises, for instance, when preconditioning Maxwell's equations discretized by edge elements. It is used in MUELU [12] "RefMaxwell" solver [4].

Hiptmair's algorithm does not use Ifpack2::Factory interface for construction. Instead, a user must explicitly call the constructor

```
Teuchos::RCP<Tpetra::CrsMatrix<> > A, Aaux, P;
// create A, Aaux, P here ...
Teuchos::ParameterList paramList;
paramList.set("hiptmair: smoother type 1", "CHEBYSHEV");
// ...
RCP<Ifpack2::Ifpack2Preconditioner<> > ifpack2Preconditioner =
  Teuchos::rcp(new Ifpack2::Hiptmair(A, Aaux, P);
ifpack2Preconditioner->setParameters(paramList);
```

Here, A is a matrix in the primary space, *Aaux* is a matrix in auxiliary space, and P is a prolongator/restrictor between the two spaces.

The following parameters are used in the Hiptmair method:

"hiptmair: smoother type 1"	[string] Smoother type for smoothing the primary space. **Default:** "CHEBYSHEV".
"hiptmair: smoother list 1"	[Teuchos::ParameterList] Smoother parameters for smoothing the primary space. **Default:** empty.
"hiptmair: smoother type 2"	[string] Smoother type for smoothing the auxiliary space. **Default:** "CHEBYSHEV".
"hiptmair: smoother list 2"	[Teuchos::ParameterList] Smoother parameters for smoothing the auxiliary space. **Default:** empty.

`"hiptmair: pre or post"`	[`string`] IFPACK2 always relaxes on the auxiliary space. “pre” (“post”) means that it relaxes on the primary space before (after) the relaxation on the auxiliary space. “both” means that we do both “pre” and “post”. **Default:** “both”.
`"hiptmair: zero starting solution"`	See `"relaxation: zero starting solution"`.

References

[1] Trilinos CMake Quickstart. http://trilinos.org/build_instructions.html, 2014.

[2] Allison H Baker, Robert D Falgout, Tzanio V Kolev, and Ulrike Meier Yang. Multigrid smoothers for ultraparallel computing. *SIAM Journal on Scientific Computing*, 33(5):2864–2887, 2011.

[3] Roscoe A. Bartlett. Teuchos::RCP beginner's guide. Technical Report SAND2004-3268, Sandia National Labs, 2010.

[4] Pavel B Bochev, Jonathan J Hu, Christopher M Siefert, and Raymond S Tuminaro. An algebraic multigrid approach based on a compatible gauge reformulation of Maxwell's equations. *SIAM Journal on Scientific Computing*, 31(1):557–583, 2008.

[5] Xiao-Chuan Cai and Marcus Sarkis. A restricted additive Schwarz preconditioner for general sparse linear systems. *SIAM Journal on Scientific Computing*, 21(2):792–797, 1999.

[6] Michael A. Heroux and James M. Willenbring. A new overview of the Trilinos project. *Scientific Programming*, 20(2):83–88, 2012.

[7] Ralf Hiptmair. Multigrid method for H(div) in three dimensions. *Electron. Trans. Numer. Anal*, 6:133–152, 1997.

[8] Mark Hoemmen, Heidi Thornquist, Michael Heroux, and Michael Parks. Tpetra: next-generation distributed linear algebra. http://trilinos.org/packages/tpetra, 2014.

[9] Xiaoye S. Li, James W. Demmel, John R. Gilbert, Laura Grigori, Meiyue Shao, and Ichitaro Yamazaki. SuperLU Users' Guide. 2011.

[10] Paul Lin, Matthew Bettencourt, Stefan Domino, Travis Fisher, Mark Hoemmen, Jonathan Hu, Eric Phipps, Andrey Prokopenko, Sivasankaran Rajamanickam, Christopher Siefert, and Stephen Kennon. Towards extreme-scale simulations for low Mach fluids with second-generation Trilinos. *Parallel Processing Letters*, 24(04):1442005, 2014.

[11] Dimitri J Mavriplis. Directional agglomeration multigrid techniques for high-Reynolds-number viscous flows. *AIAA journal*, 37(10):1222–1230, 1999.

[12] Andrey Prokopenko, Jonathan J. Hu, Tobias A. Wiesner, Christopher M. Siefert, and Raymond S. Tuminaro. MueLu User's Guide 1.0. Technical Report SAND2014-18874, Sandia National Labs, 2014.

[13] Yousef Saad. ILUT: A dual threshold incomplete LU factorization. *Numerical Linear Algebra with Applications*, 1(4):387–402, 1994.

[14] Yousef Saad. *Iterative methods for sparse linear systems*. SIAM, 2nd edition, 2003.

[15] Yousef Saad and Masha Sosonkina. Enhanced parallel multicolor preconditioning techniques for linear systems. In *PPSC*, 1999.

[16] M. Sala and M. Heroux. Robust algebraic preconditioners with IFPACK 3.0. Technical Report SAND-0662, Sandia National Laboratories, 2005.

[17] Heidi Thornquist, Roscoe A. Bartlett, Mark Hoemmen, Christopher Baker, and Michael Heroux. Teuchos: Trilinos tools library. http://trilinos.org/packages/teuchos, 2014.

[18] S Pratap Vanka. Block-implicit multigrid solution of Navier-Stokes equations in primitive variables. *Journal of Computational Physics*, 65(1):138–158, 1986.

DISTRIBUTION:

1	MS 1320	Michael Heroux, 1426
1	MS 1318	Robert Hoekstra, 1423
1	MS 1318	Erik Strack, 1426
1	MS 1320	Mark Hoemmen, 1426
1	MS 1320	Alicia Klinvex, 1426
1	MS 1320	Paul Lin, 1426
1	MS 1318	Andrey Prokopenko, 1426
1	MS 1322	Christopher Siefert, 1443
1	MS 0899	Technical Library, 9536 (electronic copy)

DISTRIBUTION:

1	MS 9159	Jonathan Hu, 1426
1	MS 9159	Raymond Tuminaro, 1442
1	MS 0899	Technical Library, 8944 (electronic copy)

Sandia National Laboratories