SANDIA REPORT

SAND2005-0662
Unlimited Release
Printed February 2005

Robust Algebraic Preconditioners using IFPACK 3.0[a]

Marzio Sala, Michael Heroux

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

Sandia is a multiprogram laboratory operated by Sandia Corporation,
a Lockheed Martin Company, for the United States Department of Energy's
National Nuclear Security Administration under Contract DE-AC04-94-AL85000.

[a]Compatible with Trilinos Releases 5.0 and 6.0.X

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.doe.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

SAND2005-0662
Unlimited Release
Printed February 2005

Robust Algebraic Preconditioners using IFPACK 3.0[†]

Marzio Sala and Michael Heroux
Computational Mathematics and Algorithms Department
Sandia National Laboratories
P.O. Box 5800
Albuquerque, NM 87185-1110

Abstract

IFPACK provides a suite of object-oriented algebraic preconditioners for the solution of preconditioned iterative solvers. IFPACK constructors expect the (distributed) real sparse matrix to be an Epetra_RowMatrix object. IFPACK can be used to define point and block relaxation preconditioners, various flavors of incomplete factorizations for symmetric and non-symmetric matrices, and one-level additive Schwarz preconditioners with variable overlap. Exact LU factorizations of the local submatrix can be accessed through the AMESOS packages.

IFPACK, as part of the Trilinos Solver Project, interacts well with other Trilinos packages. In particular, IFPACK objects can be used as preconditioners for AZTECOO, and as smoothers for ML. IFPACK is mainly written in C++, but only a limited subset of C++ features is used, in order to enhance portability.

[†]Compatible with Trilinos Releases 5.0 and 6.0.X

Acknowledgments

The authors would like to acknowledge the support of the ASCI and LDRD programs that funded development of Trilinos.

Robust Algebraic Preconditioners using IFPACK 3.0[1]

Contents

1 Introduction

The parallel solution of large linear systems of type

$$Ax = b, \tag{1}$$

where A is a (distributed) large, sparse matrix and x and b two real multi-vectors, is often achieved using iterative solvers of Krylov type (see for instance [2]). It is well known that the convergence of Krylov methods depends on the spectral properties of the linear system matrix A [1, 12, 9]. Often, A is very ill-conditioned, so the original system (1) is replaced by

$$P^{-1}Ax = P^{-1}b$$

(left-preconditioning), or by

$$AP^{-1}Px = b$$

(right-preconditioning), using a linear transformation P^{-1}, called *preconditioner*, in order to improve the spectral properties of the linear system matrix. In general terms, a preconditioner is any kind of transformation applied to (1) which makes it easier to solve, in terms of iterations and CPU time.

The general (and challenging) problem of finding an efficient preconditioner is to identify a linear operator P with the following properties:

1. P **is a good approximation of** A **is some sense**. Although no general theory is available, we can say that P should act so that $P^{-1}A$ is near to being the identity matrix and its eigenvalues are clustered within a sufficiently small region of the complex plane (see for instance [7]);

2. P **is efficient**, in the sense that the iteration method converges much faster, in terms of CPU time, for the preconditioned system. In other words, preconditioners must be selected in such a way that the cost of constructing and using them is offset by the improved convergence properties they permit to achieve;

3. P **or** P^{-1} **can be constructed in parallel**, to take advantage of the architecture of modern supercomputers.

The choice of P varies from "black-box" algebraic techniques which can be applied to general matrices to "problem dependent" preconditioners which exploit special features of a particular class of problems. Although problem dependent preconditioners can be very powerful, there is still a practical need for efficient preconditioning techniques for large classes of problems. IFPACK aims to fill the need for general, black-box preconditioners, by providing a set of robust algebraic preconditioners for parallel large scale applications.

Single-level algebraic preconditioners can be classified as follows:

1. **Relaxation schemes**, like Jacobi, Gauss-Seidel and symmetric Gauss-Seidel (point or block versions) [18]. These schemes seldomly provide satisfactory performances as stand-alone preconditioner, but can be very effective if used as smoothers in multilevel methods (like, for example, ML [15]);

2. **Polynomial preconditioner**, like Neumann, Least-Square, and Chebyshev [12].

3. **Incomplete Factorizations preconditioner**, like IC(k), ILU(k), ILUT(k) [12];
4. **One-level domain decomposition preconditioners of Schwarz type**, with minimal or wider overlap among the subdomains [16, 10]. The local linear problems can be solved with exact factorizations, incomplete factorizations, or other techniques.
5. **Sparse Approximate Inverses** (like SPAI, AINV).

IFPACK aims to define preconditioners belonging to groups 1, 3 and 4. Preconditioners of class 2 can be accessed through AztecOO. Libraries like ParaSails or SPAI are available to define preconditioners of class 5 (see for instance [8, 3]).

Remark 1. *Single-level preconditioners can be used as stand-alone preconditioners, on in conjunction with multilevel preconditioners. In this latter case, the single-level preconditioner is reinterpreted as a smoother for the multilevel hierarchy. Three families of multilevel (or multigrid) methods have been proposed in the literature: geometric multigrid [4], the classical Ruge-Stüben algebraic multigrid (AMG) [11], or smoothed aggregation (SA) [17]. The preconditioning package Hypre can be used to define AMG preconditioners, while the Trilinos package ML can be used to build SA preconditioners.*

The goal of this document is to provide an overview of all IFPACK preconditioners. Several examples are reported to illustrate how to define and use IFPACK objects. The manuscript is organized as follows. Section 2 briefly outlines the theoretical background. A general description of IFPACK preconditioners is reported in Section 3. The IFPACK factory class is detailed in Section 4. Several examples of usage are reported in Section 5. Parameters for IFPACK preconditioners are reported in Section 6. The analysis tools of IFPACK are reported in Section 7. Configuration and building are detailed in Section 8.

Further details can be found on the Doxygen documentation. Document [13] explains the design of IFPACK. The usage of the Python interface is described in document [14].

2 Theoretical background

The aim of this section is to define concepts associated with algebraic preconditioning and establish our notation. This section is not supposed to be exhaustive, nor complete on this subject. The reader is referred to the existing literature for a comprehensive presentation.

2.1 Point Relaxation Scheme

IFPACK contains a set of simple preconditioners based on point relaxation methods. Beginning with a given approximate solution, these methods modify the components of the approximation, one or a few at a time and in a certain order, until convergence is reached. Although still popular in some application areas, these preconditioners are now rarely used as stand-alone preconditioner; however, they can provide successful smoothers for multilevel methods.

All IFPACK point preconditioners are based on the decomposition

$$A = D - E - F, \tag{2}$$

where D is the diagonal part of A, $-E$ the strict lower part, and $-F$ the strict upper part. It is always assumed that the diagonal entries of A are all nonzero.

2.1.1 Point Jacobi Preconditioner

Given a starting solution $x^{(0)}$, the (damped) Jacobi method determines the $i-$th component of solution of (1) at step $k \geq 1$ as

$$a_{i,i}x_i^{(k)} = \omega \left[-\sum_{j \neq i} a_{i,j}x^{(k-1)} + b_i \right]$$

where ω is the damping parameter[1] and $a_{i,j}$ the (i,j) element of matrix A. This component-wise equation can be rewritten in a vector form as

$$x^{(k)} = \omega \left[D^{-1}(E+F)x^{(k-1)} + D^{-1}b \right],$$

or, equivalently,

$$x^{(k)} = x^{(k-1)} + \omega D^{-1}(b - Ax^{(k-1)}) = x^{(k-1)} + \omega D^{-1}r^{(k-1)}, \tag{3}$$

where $r^{(k-1)} = b - Ax^{(k-1)}$ is the residual at step $k-1$.

This preconditioner is symmetric.

2.1.2 Point Gauss-Seidel Preconditioner

The (damped) Gauss-Seidel method at step $k \geq 1$ can be written as

$$a_{i,i}x_i^{(k)} = \omega \left[-\sum_{j<i} a_{i,j}x^{(k)} - \sum_{j>i} a_{i,j}x^{(k-1)} + b_i \right]$$

where ω is the damping parameter. In vector form, one has

$$x^{(k)} = x^{(k-1)} + \omega(D-E)^{-1}(b - Ax^{(k-1)}), \tag{4}$$

which requires, at each step k, the solution of a (lower) triangular linear system. This preconditioner is non-symmetric.

2.1.3 Point SOR Preconditioner

The Successive Over Relaxation (SOR) method computes the k-th step using the relaxation sequence

$$x^{(k)} = \omega x_{GS}^{(k-1)} + (1-\omega)x^{(k-1)}, \tag{5}$$

where $x_{GS}^{(k-1)}$ is the $k-1$ step of a (non-damped) Gauss-Seidel iteration. SOR is based on the splitting

$$\omega A = (D - \omega E) - (\omega F + (1-\omega)D),$$

so that (5) can also be written as

$$(D - \omega E)x^{(k)} = \left[\omega F + (1-\omega)D\right] x^{(k-1)} + \omega b.$$

This preconditioner is non-symmetric.

[1] Clearly, is used as a solver, ω must be set to 1 to converge to the solution of (1).

2.1.4 Point SSOR Preconditioner

A Symmetric SOR (SSOR) step consist of the SOR step (5) followed by a backward SOR,

$$\begin{aligned}(D-\omega E)x^{(k-1/2)} &= [\omega F-(1-\omega)D]x^{(k-1)}+\omega b\\ (D-\omega F)x^{(k)} &= [\omega E-(1-\omega)D]x^{(k-1/2)}+\omega b.\end{aligned} \tag{6}$$

This preconditioner is symmetric. When $\omega = 1$, we obtain

$$\begin{aligned}(D-E)x^{(k-1/2)} &= Fx^{(k-1)}+b\\ (D-F)x^{(k)} &= Ex^{(k-1/2)}+b.\end{aligned} \tag{7}$$

This method is often called symmetric Gauss-Seidel.

2.2 Block Relaxation Schemes

Block relaxation schemes of Jacobi and Gauss-Seidel type generalize their point counterpart by updating a set of variables at the same time. Consider to partition the matrix A, the right-hand side and the solution vector as follows:

$$A=\begin{pmatrix} A_{1,1} & A_{1,2} & A_{1,3} & \dots & A_{1,m}\\ A_{2,1} & A_{2,2} & A_{2,3} & \dots & A_{2,m}\\ A_{3,1} & A_{3,2} & A_{3,3} & \dots & A_{3,m}\\ \vdots & \vdots & \vdots & \ddots & \vdots\\ A_{m,1} & A_{m,2} & A_{m,3} & \dots & A_{m,m}\end{pmatrix},\quad x=\begin{pmatrix} x_1\\ x_2\\ x_3\\ \vdots\\ x_m\end{pmatrix},\quad b=\begin{pmatrix} b_1\\ b_2\\ b_3\\ \vdots\\ b_m\end{pmatrix}, \tag{8}$$

in which the partitioning of x and b into m blocks is compatible with the partitioning of A. Also, it is supposed that the diagonal blocks $A_{i,i}$ are square and assumed nonsingular.

Splitting (2) can still be used to define block Jacobi and block Gauss-Seidel algorithms, with the following definitions of D, E and F:

$$D=\begin{pmatrix} A_{1,1} & & & & \\ & A_{2,2} & & & \\ & & A_{3,3} & & \\ & & & \ddots & \\ & & & & A_{m,m}\end{pmatrix}, \tag{9}$$

$$E=-\begin{pmatrix} O & & & & \\ A_{2,1} & O & & & \\ A_{3,1} & A_{3,2} & O & & \\ \vdots & \vdots & & \ddots & \\ A_{m,1} & A_{m,2} & A_{m,3} & \dots & O\end{pmatrix}, \tag{10}$$

$$F=-\begin{pmatrix} O & A_{1,2} & A_{1,3} & \dots & A_{1,m}\\ & O & A_{2,3} & \dots & A_{2,m}\\ & & O & & \\ & & & \ddots & \\ & & & & O\end{pmatrix}, \tag{11}$$

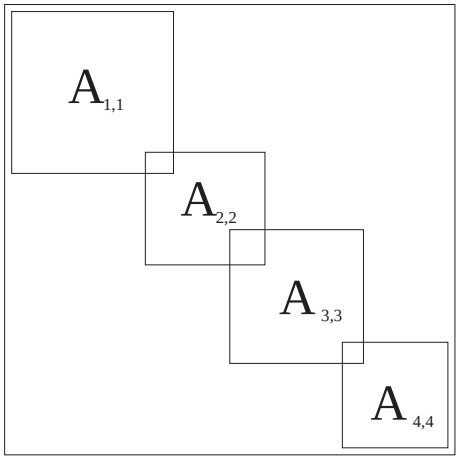

Figure 1. The block Jacobi matrix with overlapping blocks.

Using definitions (9), (10) and (11), the block Jacobi method is simply as reported in equation (3). Analogously, the block Gauss-Seidel is still described by equation (4), and the symmetric Gauss-Seidel by equation (7).

An alternative definition of block relaxation scheme is as follows. Let us suppose to partition the set of rows of the matrix into m sets $S_i, i = 1, \ldots, m$, such that

$$S_i \subseteq S, \quad \cup_i S_i = S.$$

Let V_i be a boolean $n \times n_i$ matrix (where $n_i = card(S_i)$), whose entries are defined as

$$V_{i,j} = \begin{cases} 1 & \text{if } i \in S_j \\ 0 & \text{otherwise} \end{cases}$$

A general (damped) block Jacobi iteration can be defined as follows:

$$\text{On each processor, for each block } i\text{, Do} \tag{12}$$
$$x^{(k)} = x^{(k-1)} + \omega V_i^T A_{i,i}^{-1} V_i (b - A x^{(k-1)}). \tag{13}$$

Figure 1 graphically describes the block Jacobi with variable overlap among blocks.

The (damped) block Gauss-Seidel algorithm easily derives from (13), by immediately updating the solution vector to compute the residual. The algorithm is as follows:

$$\text{On each processor, for each block } i\text{, Do} \tag{14}$$
$$x^{(k)} = x^{(k-1)} + \omega V_i^T A_{i,i}^{-1} V_i (b - A x^{(k)}). \tag{15}$$

2.3 Incomplete Factorization Preconditioners

A broad class of effective preconditioners is based on incomplete factorization technique. Such preconditioners are often referred to as incomplete lower/upper (ILU) preconditioners. ILU preconditioning techniques lie between direct and iterative methods and provide a balance between reliability and numerical efficiency. ILU preconditioners are constructed in the factored form $P = \tilde{L}\tilde{U}$, with $\tilde{L}$ and $\tilde{U}$ being a lower and an upper triangular matrices, respectively. Solving with P involves two triangular solutions.

ILU preconditioners are based on the observation that, although most matrices A admit an LU factorization $A = LU$, where L is (unit) lower triangular and U is upper triangular, the factors L and U often contain too many nonzero terms, making the cost of factorization too expensive in time and memory use. The simplest type of ILU preconditioner is ILU(0), which is defined as proceeding through the standard LU decomposition computations, but keeping only those terms in $\tilde{L}$ that correspond to nonzero terms in the lower triangle of A and similarly keeping only those terms in $\tilde{U}$ that correspond to nonzero terms in the upper triangle of A. Although effective for certain classes of problems, in some cases the accuracy of the ILU(0) may be insufficient to yield an adequate rate of convergence. More accurate factorizations will differ from ILU(0) by allowing some *fill-in*–that i, some of the elements produced by the Gaussian elimination are kept in order to make $\tilde{U}$ and $\tilde{U}$ "closer" to the exact factorization L and U. The level-of-fill should be indicative of the size of the element: the higher the level-of-fill, the smaller the elements. Several strategies have been proposed in the literature to define the level-of-fill. One strategy is to compute the graph of $A^{(\ell+1)}$ (ℓ being the level-of-fill) and use this sparsity pattern in the construction of $\tilde{L}$ and $\tilde{U}$. The resulting class of methods is called ILU(k), where k is the level-of-fill. Other strategies consider dropping by value – for example, dropping entries smaller than a prescribed threshold. Alternative dropping techniques can be based on the numerical size of the element to be discarded. Numerical dropping strategies generally yield more accurate factorizations with the same amount of fill-in as level-of-fill methods. The general strategy is to compute an entire row of the $\tilde{L}$ and $\tilde{U}$ matrices, and then keep only a certain number of the largest entries. In this way, the amount of fill-in is controlled; however, the structure of the resulting matrices is undefined. These factorizations are usually referred to as ILUT(k).

When solving a single linear system, ILUT(k) methods can be more effective than ILU(k). However, in many situations a sequence of linear systems must be solved where the pattern of the matrix A in each system is identical but the values of changed. In these situations, ILU(k) is typically much more effective because the pattern of ILU(k) will also be the same for each linear system and the overhead of computing the pattern is amortized.

2.4 Condition Number Estimates

The condition of a matrix B, called $\kappa_p(B)$, is defined as $\kappa_p(B) = \|B\|_p\|B^{-1}\|_p$ in some appropriate norm p. $\kappa_p(B)$ gives some indication of how many accurate floating point digits can be expected from operations involving the matrix and its inverse. A condition number approaching the accuracy of a given floating point number system, about 15 decimal digits in IEEE double precision, means that any results involving B or B^{-1} may be meaningless.

The computation of $\kappa_p(B)$ is in general an expensive operation, and it is therefore to be avoided unless necessary. (The Aztec library can be used to compute accurate estimates of $\kappa_2(B)$ by using CG

or GMRES.) On the other side, a crude estimate of $\kappa_\infty(B)$ can be cheaply computed as follows. The ∞-norm of a vector y is defined as the maximum of the absolute values of the vector entries, and the ∞-norm of a matrix B is defined as $\|B\|_\infty = \max_{\|y\|_\infty=1} \|By\|_\infty$. A crude lower bound for the $\kappa_\infty(B)$ is $\|B^{-1}e\|_\infty$ where $e = (1, 1, \ldots, 1)^T$. It is a lower bound because $\kappa_\infty(B) = \|B\|_\infty \|B^{-1}\|_\infty \geq \|B^{-1}\|_\infty \geq |B^{-1}e\|_\infty$.

2.4.1 *A-priori* Diagonal Perturbations

A well-known disadvantage of incomplete factorizations is that the two factors can be unstable. This may occur, for instance, if matrix A in (1) is badly scaled, or if one of the pivotal elements occurs to be very small. Computing the condition number estimate using method `Condest()` can help to detect ill-conditioned factorizations. The procedure is as follows. If this condition estimate is larger than machine precision, say greater than 10^{15}, then it is possible that numerical errors in the application of the preconditioner can destroy convergence of the iterative solver (that is, the iterative solver starts to diverge, stagnates, or aborts because it detects ill-conditioning). In this case, *a-priori* diagonal perturbations may be effective.

Diagonal perturbations operate as follows: instead of using the matrix A of (1), we perform the factorization on a modified matrix B, whose elements are defined as

$$\begin{array}{l} B_{i,j} = A_{i,j} \qquad i \neq j \\ B_{i,i} = \alpha \ \ sgn(A_{i,i}) + \rho A_{i,i}, \end{array} \tag{16}$$

where α and ρ are two real parameters, to be determined by the user. α represents an absolute threshold added to the matrix, while ρ is a relative threshold (that is, the actual diagonal value of the matrix to be factored is ρ times the original value). Note that B is never built, since the code modifies the `ExtractMyRowCopy()` method, and updates the diagonal value.

This has the effect of forcing the diagonal values to have minimal magnitude of α and to increase each by an amount proportional to ρ, and still keep the sign of the original diagonal entry.

Although no general theory to define α and ρ is available, it can be convenient to adopt the strategy outlined in Figure 2.

Remark 2. *Note that if the condition estimate of the preconditioner is well below machine precision and one is not achieving convergence, then diagonal perturbation will probably not be useful. Instead, one should try to construct a more accurate factorization by increasing the level-of-fill.*

2.5 Additive Schwarz Preconditioners

IFPACK makes very easy to define and use domain decomposition preconditioners of (overlapping) Schwarz type.

The basic idea of DD methods is to decompose the computational domain Ω into M smaller parts Ω_i, $i = 1, \ldots, M$, called subdomains, such that $\cup_{i=1}^{M} \overline{\Omega_i} = \overline{\Omega}$. Next, the original problem can be reformulated within each subdomain Ω_i, of smaller size. This family of subproblems is coupled one to another through the values of the unknown solution at subdomain interface. This coupling is then removed at the expense of introducing an iterative process which involves, at each step, solutions on the Ω_i with additional interface conditions on $\partial\Omega_i \setminus \partial\Omega$ [10, 16],

1. Set the absolute threshold $\alpha = 0.0$ and the relative threshold $\rho = 1.0$ (equivalent to no perturbation).
2. Define perturbed diagonal entries as $d_i = sign(d_i)\alpha + d_i\rho$ and compute the incomplete factors L and U.
3. Compute $condest = \|(LU)^{-1}e\|_\infty$ where $e = (1, 1, \ldots, 1)^T$.
4. If failure ($condest > 10^{15}$ or convergence is poor), set $\alpha = 10^{-5}$, $\rho = 1.0$. Repeat Steps 2 and 3.
5. If failure, set $\alpha = 10^{-5}$, $\rho = 1.01$. Repeat Steps 2 and 3.
6. If failure, set $\alpha = 10^{-2}$, $\rho = 1.0$. Repeat Steps 2 and 3.
7. If failure, set $\alpha = 10^{-2}$, $\rho = 1.01$. Repeat Steps 2 and 3.
8. If still failing, continue alternate increases in the two threshold values.

Figure 2. Simple *a priori* Threshold Strategy

In overlapping Schwarz preconditioner, the computational domain is subdivided into *overlapping* subdomains, and local Dirichlet-type problems are then solved on each subdomain. The communication between the solutions on the different subdomains is here guaranteed by the overlapping region.

The additive Schwarz preconditioner can be written as:

$$P_{AS}^{-1} = \sum_{i=1}^{M} P_i A_i^{-1} R_i, \tag{17}$$

where M is the number of subdomains (that is, the number of processors in the computation), R_i is an operator that restricts the global vector to the vector lying on subdomain Ω_i, P_i is an operator that prolongate from subdomain Ω_i to Ω, and

$$A_i = R_i A P_i. \tag{18}$$

IFPACK supports two major cases:

- Minimal-overlap (here referred to as "zero-overlap"): each subdomain is identified by the set of local rows of the preconditioned matrix; The operators R_i's and P_i's are not implemented, since the required components of the residual vector are already local. Besides, matrix (18) can be easily extracted from the local matrix, by dropping all nonzeros corresponding to non-local columns with no communications between processors.

- Wider overlap: each subdomain is identified by the set of local rows of a suitable overlapping matrix. Each application of R_i and P_i may require the importing or exporting of off-process data, and the construction of (18) requires communications.

In both cases, each processor is responsible for exactly one subdomain.

Remark 3. *Additive Schwarz preconditioners as reported in equation (17) are not scalable: their convergence rate deteriorates as the number of subdomains (that is, of the processors) increases. Algebraic techniques exist to add an algebraic coarse level correction to (17) to make the preconditioner scalable; see for example the documentation of the ML package [15].*

3 General Description of IFPACK Preconditioners

All IFPACK preconditioners described in this document are reported in Table 1. They are all derived from the `Ifpack_Preconditioner` class.

`Ifpack_Preconditioner` is a pure virtual class, derived from `Epetra_Operator`, that standarizes the construction and usage of IFPACK preconditioners. In fact, all IFPACK preconditioners are supposed to behave as follows:

1. The object is constructed, passing as only input argument the pointer of the matrix to be preconditioned, say `A`. `A` has already been `FillComplete()`'d[2].

2. All the parameters, stored in a TEUCHOS parameters list, are set using method `SetParameters()`. If `SetParameters()` is not called, default values will be used.

3. The preconditioner is initialized by calling method `Initialize()`. In this phase, all operations that do not require the matrix values of `A` are performed (that is, only the structure of `A` is used).

4. The preconditioner is constructed by calling method `Compute()`. In this phase, all the operations that require the matrix values of `A` are performed [3]. It calls `Initialize()` if not already done by the user.

5. Method `ApplyInverse()` applies the preconditioner. Any class that uses `ApplyInverse()` to apply the preconditioner can take advantage of an `Ifpack_Preconditioner` derived object[4] Note that `Compute()` must have been successfully called before using `ApplyInverse()`.

6. Method `IsInitialized()` returns `true` is the preconditioner has been successfully initialized, `false` otherwise.

7. Method `IsComputed()` returns `true` is the preconditioner has been successfully computed, `false` otherwise.

[2] It is supposed that the `OperatorDomainMap()`, the `OperatorRangeMap()` and the `RowMatrixRowMap()` of the matrix all coincide, and that each row is assigned to exactly one process.

[3] For example, in a time dependent setting, if the structure of `A` does not change from a given time step to the next but its values do, the user can call `Initialize()` only once before the first time step, then `Compute()` at each time step.

[4] For example, `AztecOO` objects can use `Ifpack_Preconditioner` objects as preconditioners.

Class name	Overlap	Description
`Ifpack_PointRelaxation`	0	Point (damped) relaxation preconditioners (Jacobi, Gauss-Seidel, symmetric Gauss-Seidel). Users can specify the number of Jacobi steps (sweeps), and the damping factor. See Section 2.1.1
`Ifpack_BlockRelaxation`	0	Block relaxation preconditioner (Jacobi, Gauss-Seidel, symmetric Gauss-Seidel). Users can store the diagonal blocks as dense or sparse. In the latter case, any IFPACK preconditioner can be used to apply the inverse of the diagonal block. See Section 2.2.
`Ifpack_AdditiveSchwarz`	user	Generic additive Schwarz preconditioner. Allows for generic additive Schwarz preconditioners, with minimal or wider overlap. In the latter case, the user must provide the overlapping matrix. Any IFPACK preconditioner can be used to solve the local problems. See Section 2.5.
`Ifpack_IC`	0	Incomplete Cholesky factorization, with dropping based on the level-of-fill of the graph.
`Ifpack_ICT`	0	Incomplete Cholesky factorization, with dropping based on threshold.
`Ifpack_ILU`	0	Incomplete LU factorization, with dropping based on the level-of-fill of the graph.
`Ifpack_ILUT`	0	Incomplete LU factorization, with dropping based on threshold.

Table 1. Description of all the IFPACK preconditioners reported in this document. In the Table, 'Overlap' indicates the overlap (with 0 being the minimal overlap case, 'any' means that the code can construct the overlapping matrix for any given positive value).

8. Method `Condest()` returns an estimation of the condition number of the preconditioner P (and not of the preconditioner system), see Section 2.4. Accurate (and expensive) computations of the condition number of the preconditioned system can be obtained by calling `Condest(Ifpack_CG)` or `Condest(Ifpack_GMRES)`[5]. See Section 2.4.1 for more details.

9. Methods `NumInitialize()`, `NumCompute()` and `NumApplyInverse()` return the number of calls to each phase.

10. Methods `InitializeTime()`, `ComputeTime()` and `ApplyInverseTime()` return the number of CPU-time spent in each phase.

11. Methods `InitializeFlops()`, `ComputeFlops()` and `ApplyInverseFlops()` return the number floating point operations (FLOPS) occurred in each phase.

12. Method `AreFlopsComputed()` return `true` is the preconditioner counts the flops, `false` otherwise.

Remark 4. *Some* IFPACK *preconditioners may require to copy the input* `List` *object given in input to* `SetParameters()`. *In any case, the user-provided list can go out of scope before* `Compute()` *is called. Note that changes to user-provided list after the call to* `SetParameters()` *will not affect the preconditioner, unless* `SetParameters()` *is re-called.*

Remark 5. *Each* `Ipfack_Preconditioner` *object overloads the* `<<` *operator. Basic information about a given preconditioner can be obtained by simply using an instruction of the type:* `cout << Prec`.

4 The Factory Class

The easiest way to define preconditioners of type (17) in IFPACK is through its factory class. Let us consider the following fragment of code, which constructs an ILU(5) preconditioner, with minimal overlap.

```
#include "Ifpack.h"
...
Epetra_RowMatrix* A; // A is already FillComplete()'d
...
Ifpack Factory;
Ifpack_Preconditioner* Prec;
string PrecType = "ILU";
// create the preconditioner using Create()
Prec = Factory.Create(PrecType, A);
assert (Prec != 0);

// specify parameters for ILU
```

[5] We note that using CG or GMRES to compute and estimated condition number is an expensive operations; see discussion in Section 2.4.

PrecType	Description
IC	Incomplete Cholesky factorization on each subdomain.
ICT	Incomplete Cholesky factorization with threshold on each subdomain.
ILU	Incomplete LU factorization on each subdomain.
ILUT	Incomplete LU with threshold on each subdomain.
Amesos	Complete LU factorization on each subdomain. Requires IFPACK support for AMESOS.

Table 2. List preconditioners supported by the Factory class.

```
Teuchos::ParameterList List;
List.set("fact: level-of-fill", 5);

Prec->SetParameters();
Prec->Initialize();
Prec->Compute();
...
// Let Problem be an Epetra_LinearProblem
AztecOO Solver(Problem);
Problem.SetPrec(Prec);
// now we can solve with AztecOO
```

The difficulty with this type of preconditioner is that it tends to become less robust and require more iterations as the number of processors used increases. This effect can be offset to some extent by allowing *overlap*. Overlap refers to having processors redundantly own certain rows of the matrix for the ILU factorization. Level-1 overlap is defined so that a processor will include rows that are part of its original set. In addition, if row i is part of its original set and row i of A has a nonzero entry in column j, then row j will also be included in the factorization on that processor. Other levels of overlap are computed recursively. IFPACK supports an arbitrary level of overlap. However, level-1 is often most effective. Seldom more than 3 levels are needed.

To increase the overlap among processors, one can simply call method `Create()` as follows:

```
int OverlapLevel = 2;
Prec = Factory.Create(PrecType, A, OverlapLevel);
```

The list of options for `PrecType` is reported in Table 2. Note that only one word in the above fragment of code has to be changed to define, for instance, the Gauss-Seidel preconditioner.

5 Examples of Usage

This section contains several examples of usage of IFPACK preconditioners. A detailed list of IFPACK parameters is reported in section 6.

5.1 Point Relaxation Schemes

An example of usage of point relaxation preconditioners (in this case, Gauss-Seidel) is as follows:

```
#include "Teuchos_ParameterList.hpp"
#include "Ifpack_PointRelaxation.h"
```

Let `A` be a pointer to an `Epetra_RowMatrix` derived object, and let `Problem` be a pointer to an `Epetra_LinearProblem`. We suppose that `A` and `Problem` are properly set, and method `FillComplete()` has been called. At this point, we can create the preconditioner as

```
Teuchos::ParameterList List;
List.set("relaxation: type", "Gauss-Seidel");

Ifpack_PointRelaxation Prec(A);

IFPACK_CHK_ERR(Prec.SetParameters(List));
IFPACK_CHK_ERR(Prec.Initialize());
IFPACK_CHK_ERR(Prec.Compute());
```

Now, we can set the IFPACK preconditioner for AztecOO:

```
AztecOO AztecOOProblem(Problem);
AztecOOProblem.SetPrecOperator(Prec);
```

as call `AztecOO.Iterate()` as required.

Macro `IFPACK_CHK_ERR()` can be used to check return values. If the return value if different from 0, the macro prints out a warning message on `cerr`, and returns.

Remark 6. *Point relaxation schemes are implemented in* IFPACK *for general Epetra_RowMatrix's and they therefore require several calls to method* `ExtractMyRowCopy()`. *This approach make point relaxation schemes quite flexible, but potentially slower than implementations tuned for a particular matrix format.*

5.2 Block Relaxation Schemes

From the point of view of the implementation, block preconditioners are sensibly more complex than their point counterpart:

1. A strategy to define the blocks has to be chosen (for instance, a linear partitioner, or a graph decomposition algortithm);

2. block Jacobi and block Gauss-Seidel algorithms require the application of the inverse of each diagonal block $A_{i,i}$. Blocks of small dimension should be stored as dense matrices, while larger blocks require sparse storage. In this latter case, to apply the inverse of the block can be reformulated as applying a preconditioner for matrix $A_{i,i}$. The code must allow for different choices of block preconditioners.

Let us start with the definition of the blocks. IFPACK provides the following options:

- a linear partitioning, using class `Ifpack_LinearPartitioner`;
- a simple greedy algorithm, using class `Ipfack_GreedyPartitioner`;
- an interface to METIS, using class `Ifpack_METISPartitioner`.

It is important to note that all blocks are *local* – that is, all partitioner schemes will *always* decompose the local graph only[6].

All IFPACK partitioners are derived from the pure virtual class `Ifpack_Partitioner`, and all require in the constructor phase an `Ifpack_Graph` object. `Ifpack_Graph`'s can be easily created (as light-weigth) conversions from `Epetra_RowMatrix`'s and `Epetra_CrsGraph`'s, as follows At this point, we can create the preconditioner as

```
#include "Ifpack_Graph.h"
#include "Ifpack_Graph_Epetra_CrsGraph.h"
#include "Ifpack_Graph_Epetra_RowMatrix.h"

// use either CsrA or RowA, depending on your application
Epetra_CrsMatrix* CrsA;
Epetra_RowMatrix* RowA;

Ifpack_Graph CrsGraph* CrsGraph =
  new Ifpack_Graph_CrsGraph(&(CrsA->Graph()));

Ifpack_Graph RowGrap* RowGraph  =
  new Ifpack_Graph_RowMatrix(RowA);
```

Note that the `Partitioner` object will decompose the graph (either `CrsGraph` or `RowGraph`) into non-overlapping sets (that is, each graph vertex is assigned to exactly one set).

The following fragment of code shows how to use a greedy partitioner to define 4 local blocks for a given `Ifpack_Graph`.

```
#include "Ifpack_Graph.h"
#include "Ifpack_GreedyPartitioner.h"
#include "Ifpack_BlockRelaxation.h"
#include "Teuchos_ParameterList.hpp"
...
```

[6] If used in conjuction with class `Ifpack_AdditiveSchwarz`, blocks can span more than one processor.

```
Ifpack_Graph* Graph;
// Graph is created here

Teuchos::ParameterList List;
List.set("partitioner: local parts", 4);
Ifpack_Partitioner* Partitioner = new Ifpack_GreedyPartitioner(Graph);

// set the parameters (in this case the # of blocks only)
Partitioner->SetParameters(List);

// compute the partition
Partitioner->Compute();
```

Once an `Ifpack_Partitioner` is created, we are ready to compute the block preconditioner. This requires the extraction of all the diagonal blocks of equation (9). In IFPACK, the user can choose to store the $A_{i,i}$ as dense matrices, or a sparse matrices. In the former case, the inverse of each block is applied using LAPACK[7]. In the latter, the user can specify any valid `Ifpack_Preconditioner`.

As an example, we now create a block Jacobi preconditioner for a given `Epetra_RowMatrix`, say `A`, with damping parameter of 0.67, and 2 sweeps. Each diagonal block is stored as a dense matrix.

```
#include "Ifpack_BlockRelaxation.h"
#include "Ifpack_DenseContainer.h"
...

Ifpack_Partitioner* Partitioner;
// Partitioner is created here

Ifpack_Preconditioner* Prec =
  new Ifpack_BlockRelaxation<Ifpack_DenseContainer>(A);

Teuchos::ParameterList List;
List.set("relaxation: sweeps", 2);
List.set("relaxation: damping parameter", 0.67);
Prec->SetParameters(List);
Prec->Compute();
```

The previous example makes use of a dense containers to store the diagonal blocks. In IFPACK, a *container* is an object that contains all the necessary data to solve the linear system with any given $A_{i,i}$. `Ifpack_DenseContainer` stores each $A_{i,i}$ as `Epetra_SerialDenseMatrix`. Alternatively, one can use `Ipfack_SparseContainer` to store each block as an `Epetra_CrsMatrix`. Sparse containers are templated with an `Ifpack_Preconditioner`, so that the user can specify which IFPACK preconditioner has to be used to apply the inverse of each sparse block.

[7] LAPACK is used to factorize the matrix, then each application of $A_{i,i}^{-1}$ results in a dense linear system solution.

The following fragment of code illustrates how to use the direct factorization of Amesos (through class `Ifpack_Amesos`[8]) with sparse containers. The preconditioner will be a block Gauss-Seidel one.

```
#include "Ifpack_BlockRelaxation.h"
#include "Ifpack_SparseContainer.h"
#include "Ifpack_Amesos.h"
...

Ifpack_Partitioner* Partitioner;
// Partitioner is created here

Ifpack_Preconditioner* Prec =
  new Ifpack_BlockRelaxation<Ifpack_SparseContainer<Ifpack_Amesos> >(A);

Teuchos::ParameterList List;
List.set("relaxation: sweeps", 2);
List.set("amesos: solver type", "Amesos_Klu");
Prec->SetParameters(List);
Prec->Initialize();
Prec->Compute();
```

Option `amesos: solver type` specifies the AMESOS solver that has to be adopted. If the selected solver is not available, then `Ifpack_Amesos` will create an `Amesos_Klu` solver[9]. As *any* IFPACK preconditioner can be used, one can also adopt, for instance, a point Gauss-Seidel algorithm in each block:

```
Ifpack_Preconditioner* Prec =
  new Ifpack_BlockRelaxation<Ifpack_SparseContainer<Ifpack_GaussSeidel> >(A);
```

A call to `SetParameters(List)` will set the parameters for the block preconditioner.

5.3 Additive Schwarz Preconditioners

Once matrices (18) have been formed, the user still need to define a strategy to apply the inverse of A_i in (17). At this purpose, any IFPACK preconditioner can be adopted. Common choices are:

- To solve exactly on each subdomain with an complete LU factorization, using the `Ifpack_Amesos` preconditioner. This is shown in Section 5.3.1.
- To solve using an incomplete LU factorization (ILU), as presented in Section 5.3.2.
- To furtherly decompose the local domain into smaller subdomains, then apply a block Jacobi or block Gauss-Seidel preconditioner. This is outlined in Section 5.3.3.

[8] This requires IFPACK to be configured with option `--enable-amesos`.

[9] KLU is compiled by default with AMESOS. Please consult the AMESOS documentation for more details.

5.3.1 Additive Schwarz with Exact Local Solves

The following fragment of code shows the use of additive preconditioners. The local subproblems with matrix A_i are solved using a (complete) LU factorization through AMESOS.

```
#include "Ifpack_AdditiveSchwarz.h"
#include "Ifpack_Amesos.h"

Epetra_RowMatrix* A;
// Here the elements of A are filled, and FillComplete() is called.

int OverlapLevel = 0;
Ifpack_Preconditioner* Prec =
  new Ifpack_AdditiveSchwarz<Ifpack_Amesos>(A, OverlapLevel);

Teuchos::ParameterList List;
IFPACK_CHK_ERR(Prec->SetParameters(List));
IFPACK_CHK_ERR(Prec->Initialize());
IFPACK_CHK_ERR(Prec->Compute());
```

Remark 7. *Complete factorizations can be expensive to compute, especially for problems arising from discretizations on 3D grids. The user should consider complete factorizations if the local problems are small, or when other, cheaper preconditioners fail.*

5.3.2 Additive Schwarz with ILU

The following fragment of code shows the use of additive preconditioners. The local subproblems with matrix A_i are solved using an incomplete factorization.

```
#include "Ifpack_AdditiveSchwarz.h"
#include "Ifpack_ILU.h"

Epetra_RowMatrix* A;
// Here the elements of A are filled, and FillComplete() is called.

int OverlapLevel = 0;
Ifpack_Preconditioner* Prec =
  new Ifpack_AdditiveSchwarz<Ifpack_ILU>(A, OverlapLevel);

Teuchos::ParameterList List;
List.set("fact: level of fill", 2);
IFPACK_CHK_ERR(Prec->SetParameters(List));
IFPACK_CHK_ERR(Prec->Initialize());
IFPACK_CHK_ERR(Prec->Compute());
```

The user can access the factorization of the local matrix produced by templating `Ifpack_AdditiveSchwarz` with classes `Ifpack_IC`, `Ifpack_ICT`, `Ifpack_ILU` and `Ifpack_ILUT` in the following way:

```
Ifpack_Preconditioner* Prec =
  new Ifpack_AdditiveSchwarz<Ifpack_ILU>(A, OverlapLevel);

Ifpack_ILU* Inverse = Prec->Inverse();
```

Then, the total number of nonzeros in the L and U factors can be queried as follows:

```
int NumGlobalNonzerosLU = Inverse->NumGlobalNonzeros();
```

The L and U factors are stored as `Epetra_CrsMatrix`'s, whose pointers can be obtained as follows[10]:

```
const Epetra_CrsMatrix& L = Inverse->L();
const Epetra_CrsMatrix& U = Inverse->U();
```

5.3.3 Additive Schwarz with Local Block Preconditioners

Another possible technique to apply the inverse of A_i in (17) is to adopt a block preconditioner, like block Jacobi or block Gauss-Seidel (see Section 2.2). This requires a bit more work, as we have to specify the partitioner, and the container. Let us start with dense containers.

The required include files are:

```
#include "Ifpack_AdditiveSchwarz.h"
#include "Ifpack_BlockPreconditioner.h"
#include "Ifpack_Graph_Epetra_RowMatrix.h"
#include "Ifpack_DenseContainer.h"
```

Let `A` be an `Epetra_RowMatrix`. We suppose that `FillComplete()` has been called.

As always, we create a parameters list, that will be used for all IFPACK objects:

```
Teuchos::ParameterList List;
```

At this point we can create the block Jacobi preconditioner as follows:

```
Ifpack_Preconditioner* Prec =
  new Ifpack_AdditiveSchwarz<
    Ifpack_BlockPreconditioner<Ifpack_DenseContainer> >(A);

Prec->SetParameters(List);
Prec->Initialize();
Prec->Compute();
```

As we have used `Ifpack_DenseContainer`, blocks are stored are dense matrices, and LAPACK is used to apply the inverse of each block. This can be a limiting factor for large blocks. In this latter case, it is preferable to store the blocks are sparse matrices, and use a sparse solver to apply their inverse. This can be done by resorting to `Ifpack_SparseContainer`. Sparse containers can be used with minor modifications. The only difference is that we also have to specify how to apply the inverse of each block, for instance using the exact factorizations of AMESOS:

[10] For classes `Ifpack_IC` and `Ifpack_ICT` the user shall use method `H()`.

```
Ifpack_Preconditioner* Prec =
  new Ifpack_AdditiveSchwarz<Ifpack_BlockPreconditioner
    <Ifpack_SparseContainer<Ifpack_Amesos> > >(A);
```

Should the user want to use a block Gauss-Seidel preconditioner (where each block is defined by partitioning the local graph of the overlapping matrix), he/she could proceed as follows:

```
Teuchos::ParameterList List;
List.set("relaxation: damping factor", .67);
List.set("relaxation: sweeps",5);
List.set("partitioner: local parts", 4);
List.set("partitioner: overlap", OverlapLevel);

Epetra_RowMatrix* A; // A is FillComplete()'d.

Ifpack_Preconditioner* Prec =
  new Ifpack_AdditiveSchwarz<Ifpack_BlockPreconditioner
      <Ifpack_SparseContainer<Ifpack_Amesos> > >(A,OverlapLevel);

IFPACK_CHK_ERR(Prec->SetParameters(List));
IFPACK_CHK_ERR(Prec->Compute());
```

6 Parameters for IFPACK preconditioners

The parameters that affect the IFPACK preconditioners are reported below. It is important to note that parameters for all IFPACK preconditioners must be spelled as indicated: misspelled parameters will be ignored, parameters are case sensitive, and words are separated by one space only.

For more details about the TEUCHOS parameters list we refer to the TEUCHOS documentation. Table 3 briefly reports the most important methods of this class. IFPACK requires just a very basic usage of the parameters list. Input parameters are set via method `set(Name,Value)`, where `Name` is a string containing the parameter name, and `Value` is the specified parameter value, whose type can be any C++ object or pointer.

`set(Name,Value)`	Add entry `Name` with value and type specified by `Value`. Any C++ type (like int, double, a pointer, etc.) is valid.
`get(Name,DefValue)`	Get value (whose type is automatically specified by `DefValue`). If not present, return `DefValue`.
`subList(Name)`	Get a reference to sublist `List`. If not present, create the sublist.

Table 3. Some methods of Teuchos::ParameterList class.

`relaxation: type`	[`string`] Relaxation scheme. Valid choices are: `Jacobi`, `Gauss-Seidel`, `symmmetric Gauss-Seidel`. Default: `Jacobi`.

`relaxation: sweeps`	[`int`] Number of sweeps of the point relaxation preconditioner. Default: `1`.
`relaxation: damping factor`	[`double`] Value of ω in formulae (3), (4), (5) and (6). Default: `1.0`.
`relaxation: min diagonal value`	[`double`] Replace diagonal values whose absolute value is less than the specified value by this value (for point relaxation methods only). Default: `1e-9`.
`relaxation: zero starting solution`	[`bool`] If `true`, the input values in the preconditioned vector will be used as starting solution (for relaxation methods only). Default: `true`.
`partitioner: type`	[`string`] Defines how to build the local blocks (for block relaxation methods). Valid choices are: `linear` (use a simple linear decomposition), `greedy` (use a greedy algorithm to partition the local graph), or `metis` (call METIS on the local graph). Default: `linear`.
`partitioner: local parts`	[`int`] Number of (local) subgraphs (for block relaxation methods only). Default: 4.
`partitioner: overlap`	[`int`] Overlap among blocks. Only for the block Jacobi method. Default: 0.
`partitioner: root node`	[`int`] Root node, for greedy algorithm only. Default: 0
`schwarz: combine mode`	[`Epetra_CombineMode`]. It can assume one of the following values: `Add`: Components on the receiving processor will be added together; `Zero`: Off-processor components will be ignored; `Insert`: Off-processor components will be inserted into locations on receiving processor replacing existing values. `Average`: Off-processor components will be averaged with existing; `AbsMax`: Magnitudes of Off-processor components will be maxed with magnitudes of existing components on the receiving processor. Note that, for non-zero overlap values, the preconditioner is in general non-symmetric, due to the handling of the overlapping region. Set this parameter to `Insert` if a symmetric preconditioner is required. Default: `Zero`.

`amesos: solver type`	[`string`]. Defines the Amesos solver to be used by class Ifpack_Amesos. Valid values are: `Amesos_Lapack`, `Amesos_Klu`, `Amesos_Umfpack`, `Amesos_Superlu`, `Amesos_Mumps`, `Amesos_Dscpack`. Default: `Amesos_Klu`.
`fact: level-of-fill`	[`int`] Level-of-fill for IC and ILU.
`fact: ict level-of-fill`	[`double`] Level-of-fill for ICT.
`fact: ilut level-of-fill`	[`double`] Level-of-fill for ILUT.
`fact: relax value`	[`double`] Relaxation value.
`fact: absolute threshold`	[`double`] Value of α in equation (16).
`fact: relative threshold`	[`double`] Value of ρ in equation (16).

7 Analysis Tools

IFPACK contains the following tools to analyze a linear system matrix:

- Function `Ifpack_Analyze()` reports some information about the structure of the matrix, its diagonal elements, and others.
- Function `Ifpack_PrintSparsity()` prints on a PostScript file the sparsity pattern of a given `Epetra_RowMatrix`.
- Function `Ifpack_PrintSparsitySimple()`, to be used only with small matrices, prints on a screen the sparsity pattern of a given `Epetra_RowMatrix`.

8 Configuring and Building IFPACK

We recommend to configure and build IFPACK as part of the standard TRILINOS build and configure process. In fact, IFPACK is built by default if you follow the standard TRILINOS configure and build directions. Please refer to the TRILINOS documentation for information about the configuration and building of other TRILINOS packages.

To configure and build IFPACK through TRILINOS, you may need do the following (actual configuration options may vary depending on the specific architecture, installation, and user's need). It's assumed that shell variable `$TRILINOS_HOME` identifies the TRILINOS directory, and, for example, that we are compiling under LINUX and MPI.

```
% cd $TRILINOS_HOME
% mkdir LINUX_MPI
% cd LINUX_MPI
% $TRILINOS_HOME/configure  --with-mpi-compilers \
    --prefix=$TRILINOS_HOME/LINUX_MPI
% make
% make install
```

IFPACK is configured and built using the GNU autoconf [5] and automake [6] tools. IFPACKconfiguration and compilation can be tuned by several flags. The user may type

```
% configure --help
```

in the IFPACK source directory for a complete list. Here, we briefly report the list of packages (included or not in Trilinos) that are supported by IFPACK:

`--enable-amesos`	Enables support for the AMESOS package, which can be used to solve the local subproblems in Schwarz-type preconditioners, or in block Jacobi and block Gauss-Seidel preconditioners.
`--enable-aztecoo`	Enable support for the AZTECOO package. AZTECOO is used in several tests and examples.
`--enable-teuchos`	Enable support for the TEUCHOS package, whose parameters list is used by several IFPACK classes.
`--enable-triutils`	Enable support for the TRIUTILS package, which is used in some examples and test to generate the linear system.
`--enable-ifpack-metis`	Enable support for the METIS package, version 4.0 or later. METIS can be used to create block preconditioners.

Remark 8. IFPACK *cannot be compiled without the* EPETRA *library.*

References

[1] O. Axelsson. *Iterative Solution Methods*. Cambridge University Press, Cambridge, 1994.

[2] R. Barrett, M. Berry, T.F. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H. Van der Vorst. *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods, 2nd Edition*. SIAM, Philadelphia, PA, 1994.

[3] M. Benzi and M. Tuma. A sparse approximate inverse preconditioner for nonsymmetric linear systems. *SIAM Journal on Scientific Computing*, 19(3):968–994, 1998.

[4] A. Brandt. Multi-level Adaptive Solutions to Boundary-Value Problems. *Math. Comp.*, 31:333–390, 1977.

[5] Free Software Foundation. Autoconf Home Page. http://www.gnu.org/software/autoconf, 2004.

[6] Free Software Foundation. Automake Home Page. http://www.gnu.org/software/automake, 2004.

[7] A. Greenbaum. *Iterative Methods for Solving Linear Systems*. Frontiers in Applied Mathematics 17. Society for Industrial and Applied Mathematics, Philadelphia, PA, 1997.

[8] M.J. Grote and T. Huckle. Parallel preconditioning with sparse approximate inverses. *SIAM Journal on Scientific Computing*, 18(3):838–853, 1997.

[9] A. Quarteroni, R. Sacco, and F. Saleri. *Numerical Mathematics*. Springer-Verlag, New York, 2000.

[10] A. Quarteroni and A. Valli. *Domain Decomposition Methods for Partial Differential Equations*. Oxford University Press, Oxford, 1999.

[11] J. Ruge and K. Stuben. Algebraic multigrid (AMG). In S. McCormick, editor, *Multigrid Methods*. Frontiers in Applied Mathematics, 1987.

[12] Y. Saad. *Iterative Methods for Sparse Linear Systems*. PWS, Boston, 1996.

[13] M. Sala. On the design of algebraic one level domain decomposition preconditioners. Technical report, Sandia National Laboratories, June 2005. In preparation.

[14] M. Sala. PyTrilinos tutorial. Technical report, Sandia National Laboratories, June 2005. In preparation.

[15] M. Sala, J. Hu, and R. Tuminaro. ML 3.0 smoothed aggregation user's guide. Technical Report SAND-2195, Sandia National Laboratories, May 2004.

[16] B.F. Smith, P. Bjorstad, and W.D. Gropp. *Parallel Multilevel Methods for Elliptic Partial Differential Equatio ns*. Cambridge University Press, Cambrige, 1996.

[17] P. Vanek, M. Brezina, and J. Mandel. Convergence of Algebraic Multigrid Based on Smoothed Aggregation. Technical Report report 126, UCD/CCM, Denver, CO, 1998.

[18] R. Varga. *Matrix Iterative Analysis, Second Edition*. Springer-Verlag, Berlin, 2000.

Distribution List for “Robust Algebraic Preconditioners using IFPACK 3.0:”

- MS 9018 Central Technical Files, 8945-1
- MS 0899 Technical Library, 9616
- MS 0123 LDRD Donna Chavez, 1011