SAND2005–1382
Unlimited Release
Printed March 2005

MLAPI: A C++ Framework for Multilevel Preconditioners

Marzio Sala
Computational Math & Algorithms
Sandia National Laboratories
P.O. Box 5800
Albuquerque, NM 87185-1110

Abstract

We discuss the design of MLAPI, an object oriented framework that enables the development and usage of efficient, scalable and portable implementations of multilevel preconditioners for general distributed sparse matrices, in both serial and parallel environments.

The main feature of this framework is the use of several programming paradigms for the different implementation layes, with a strong emphasis on object oriented classes and operator overloading for the top layer, and optimized FORTRAN and C code for the layers underneath. In particular, MLAPI takes advantage of ML [21], the algebraic multilevel preconditioning package of Trilinos [13].

The outline the most important features of MLAPI, and we describe its usage with several examples.

Note: An improved version of this paper, containing numerical comparisons between MLAPI and ML, has been submitted to ACM-TOMS.

(page intentionally left blank)

Contents

1 Introduction

The parallel solution of large linear systems of type

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

where A is a (distributed) large, real sparse matrix and x and b two real multi-vectors, is the computational kernel of many applications. The solution of such systems is a fundamental, and often the most time consuming, part of many simulation codes. Because of the size of A, iterative solvers of Krylov type are generally adopted [3], the best known methods being the conjugate gradient [15] and GMRES [18].

It is well known that the convergence of Krylov methods depends on the spectral properties of the linear system matrix A [1, 17, 16]. Often, A is very ill-conditioned, so the original system (1) is replaced by

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

(left-preconditioning), or by

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

(right-preconditioning), using a linear transformation P^{-1}, called *preconditioner*. P^{-1} represents a sequence of operations that somehow approximates the effect of A^{-1} on a vector. Loosely stated, a preconditioner is any kind of transformation applied to (1) which makes it easier to solve, in terms of iterations and CPU time; see for instance [9, 8, 23]. More precisely, 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;

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.

A very successful class of preconditioners is represented by multilevel (or multigrid) methods. A multilevel method tries to approximate the original PDE problem of interest on a hierarchy of grids and use 'solutions' from coarse grids to accelerate the convergence on the finest grid. Multilevel and multigrid methods were introduced in the late 70's, and their success and development is testified by the enormous literature and the several international conferences organized since then. At least three approaches have been presented in literature to define the multilevel hierarchy: to use a sequence of grids, as done in geometric multigrid [4, 10, 12] to coarsen on each level by identifying a set of coarser-level nodes (the so-called C-nodes) and finer-level nodes (F-nodes) (see for instance [5, 11]), or to coarsen on each level by grouping the nodes into contiguous subsets, called aggregates, as done

```
MultiLevelStartUp(A, MaxLevels, k)
{
  if (k == MaxLevels)
    return;
  else {
    P[k] = BuildP();
    R[k] = BuildR();
    A[k] = R[k] * A[k] * P[k];
  }
}
```

Figure 1: Pseudo-code of a the multilevel start-up procedure.

```
MultiLevelSolve(A, f, x, k)
{
  if (k == CoarsestLevel)
    u = CoarseSolver(A[k]), b);
  else {
    u = Smoother(A[k], f, u);
    r = R[k] * (f - A[k] * u);
    v = 0;
    MultiLevelSolve(A[k + 1], v, k + 1);
    u = u + P[k] * v;
    u = Smoother(A[k], f, u);
  }
}
```

Figure 2: Pseudo-code of a multilevel application procedure.

in smoothed aggregation [25, 26, 27]. Multilevel methods are well understood on model problems, while their application to more general, non-symmetric, problems still requires further developments.

Like most preconditioners, a multilevel preconditioner requires a startup procedure (a common version of which is reported in Figure 1) and an application procedure (reported in Figure 2). In the Figures, `k` is the current level, `A` is an array of matrices, with `A[k]` the operator for level `k` and `A[0]` containing the matrix of the linear system (1), `R[k]` is the restriction operator from level `k` to level `k + 1`, and `P[k]` is the prolongator operator from level `k + 1` to level `k`, `CoarseSolver()` is a generic direct solver, and `Smoother()` is a generic smoother (that is, an approximate solver whose goal is to reduce the high frequencies of the error).

The effectiveness of the multilevel algorithm heavily depends on how the operators `A[k], P[k], R[k]` are defined. To leverage software development, it is preferable to code the multilevel preconditioner in a general framework, based on abstract interfaces. Eventually, a wide range of preconditioners based on these abstract interfaces should be available so that a method that matches the difficulty of the problem and the computational architecture

available can be adopted.

Note that two distinct sets of operations can be identified in the multilevel procedures:

1. operations that require operators as unique identities (for example, the application of an operator, `y = A * x`);
2. operations that require the specific knowledge of the structure of an operator and its internal data (for instance, the definition of the prologator operator `P[k]`).

Our aim is to simply the coding of a multilevel algorithm, by allowing an intuitive syntax for all operations in group (1), and by condensing operations in group (2) in well defined functions (or classes).

Ideally, the code should not look too different from what we have just presented, which in turn is just a modest change with respect to the mathematical standpoint used to define a multilevel method. Unfortunately, this is not what happens in most codes. Traditionally, coding is made complex by "details" inessential to the algorithm, like, for example, the dimension of the input and output vector, of flags for the smoother or the matrix-matrix product, or the parallel data layout.

Since these "details" are necessary to compile and run the program, our aim is the following: *let the compiler take care of the details, and let the programmer-developer to focus on the algorithm.* For example, any well designed implementation of a linear operator already contains the number of rows and columns, and the implementation of a vector contains the number of elements in the vector. By using a light layer of C++ and taking advantage of operator overloading (see, for example, [22]), one can instruct the compiler on how to look for all the necessary information, so that all operations are properly executed.

In this paper, we want to show that it is possible to obtain intuitive, easy-to-read and easy-to-develop codes, that are *at the same time* efficient and scalable. This paper is organized as follows. First, in Section 2 we will outline why an object-oriented interface can be useful to develop multilevel preconditioners (and, more general, multilevel solvers). Section 3 introduces the most important MLAPI classes, whose usage is presented in Section 4. The MATLAB © interface is detailed in Section 4.3. Section 5 outlines the conclusions.

2 Software Design

We have defined the MLAPI library to handle the definition of multilevel preconditioners for sparse large linear systems of type (1) on distributed memory computers. Our aim was to design develop a general framework, that is portable and straightforward to use, while being both flexible and efficient. The design of the library is based on the following principles:

Portability. Implementations of numerical algorithms should be directly portable across machine platforms. MLAPI is written in ANSI C++. The STL library is employed to increase efficiency. For message passing, we adopted MPI, which is the *de-facto* standard, and as such widely available and accepted by users of parallel applications. As a result, MLAPI has been successfully compiled on Linux, SGI Origin, DEC-alpha, SUN, and MAC OS X.

Clarity. Implementations of numerical algorithms should resemble the mathematical formulation on which they are based. This is in contrast to FORTRAN and C, which can require complicated subroutines or function calls, with a long parameter list. A key design for MLAPI was a user interface that is intuitive. Our intention is that it should be possible for this library to be used by those who have only a basic knowledge of MPI and C++. Ideally, the structure of all MLAPI kernels should be as close as possible to the that of MATLAB. We have developed two types of C++ interfaces to basic kernels. The first type utilizes the binary operators `*, +, -`, overloaded using the C++ capabilities. The second type is a set of interfaces (methods, functions) which can group triads or perform more complex operations.

Flexibility. MLAPI is not based on any particular matrix format. This is particularly convenient since several matrix formats are currently in used. MLAPI supports any data format that can offer a `getrow()` function, which returns the column ID and the value of all nonzero elements for any locally hosted row. C users can provide this using the `ML_Operator` structure, while C++ users can derive a class from the pure virtual `Epetra_RowMatrix` class of the EPETRA package. Similarly, all operators defined by MLAPI are wrapped as `Epetra_RowMatrix` and `ML_Operator`, so that users can access their nonzero elements without worrying about the actual data storage used by MLAPI .

Extensibility. Multilevel algorithms are far from being completely understood for all classes of problems. It is important for the multilevel library to be easily extended, to validate new approaches. To that aim, encapsulation is used to hide details in specific classes. Besides, a set of function based on abstract interfaces is provided, to generate the necessary multilevel operators. Polymorphism allows the user to derive classes to implement new features. All MLAPI classes and methods automatically use a set of default parameters, that can be tuned by specifying the desired parameters in a parameter list.

High performance. A good numerical package that utilizes OO programming must exhibit a computational efficiency that is comparable to that of FORTRAN and C codes. This puts severe limitations to the OO design. It is certain easy to generate elegant but inefficient C++ code. We implemented MLAPI as a light layer of C++, on the top of a mixture of FORTRAN77 kernels (BLAS [7] and LAPACK [6]) for all dense matrix operations and vector updates, C functions for all sparse matrix operations (like distributed sparse matrix-matrix product), and C++, for memory management. The layer structure of MLAPI is shown in Figure 3. Lower layers of the library indicates an encapsulation relationship with upper layers. As a result, MLAPI is (almost) as efficient as the C or FORTRAN library underneath.

Memory Management. One of the most powerful feature of C and C++ if the capability of allocating memory. Unfortunately, this is also the area where most bugs are found – not to mention memory leaks. We have adopted smart pointers to manage memory [2]. MLAPI objects should never be allocated using `new`, and therefore never free them using `delete`. The code automatically deletes memory when it is no longer referenced by any object. Besides, functions or methods that need to return MLAPI objects, should always return an instance of the required object, and not a pointer or a reference.

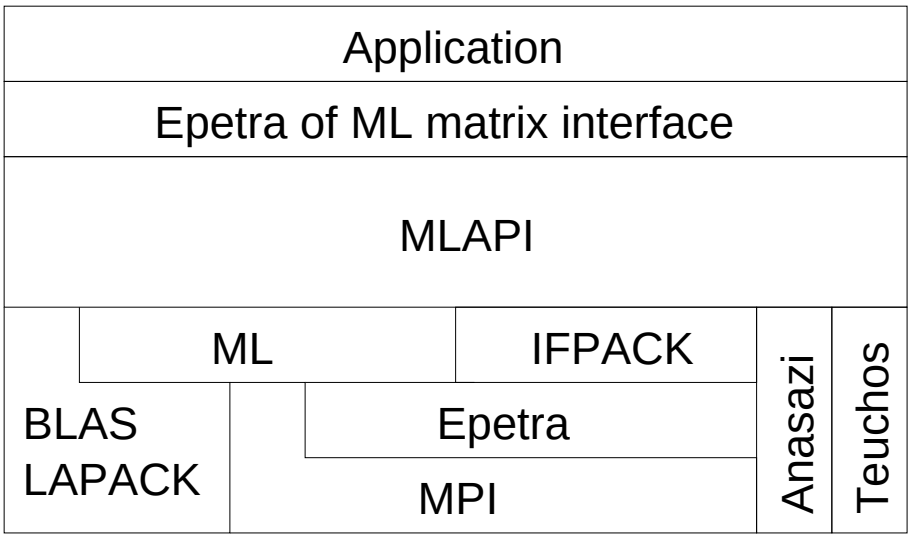

Figure 3: Layer structure of MLAPI .

At a first glance, this may appear as a major limitation for optimal performances. Dealing with an instance of an object, and not with pointers or references, signifies that the new instances have to be created and copied, usually an expensive operation. This is *not* what happens with MLAPI. In fact, all MLAPI objects are defined as light containers of pointers, automatically allocated and managed as necessary.
Let us consider three generic MLAPI objects. The assignment `A = B` means the following: all smart pointers contained by `B` are copied in `A`, both `A` and `B` point to the same memory location. However, `A` and `B` are not aliases: we can still write `B = C`, meaning that `A` contains what was contained in `B`, and both `B` and `C` point to the same memory location. Should we need to create a copy of `C` in `B`, we will use the instruction `B = Duplicate(C)`, which is instead an expensive operation, as new memory needs to be allocated, then all elements in `C` need to be copied in `B`.

Operator overloading. Operator overloading is an interesting capability of C++ that has been only partially used in the scientific computing community. We adopt expression templates (see for instance [24]) to increase the performances of MLAPI.

3 MLAPI Classes

In this section we introduce the most important classes of MLAPI : `Space` (analyzed in Section 3.1), `MultiVector` (analyzed in Section 3.2), `Operator` (described in Section 3.3) and `InverseOperator` (in Section 3.4). Furthermore, MLAPI furnishes two matrices classes, `SerialMatrix` and `DistributedMatrix`, to set the matrix elements in a very intuitive way.

The inheritance diagram is reported in Figure 4. All classes are derived from a basic class, `BaseObject`. Entities that are mathematically equivalent to operators are derived from `BaseOperator` class, which basically contains only method `Apply()`. Two concrete implementations are `Operator` and `InverseOperator`. Classes `Operator`, `InverseOperator`,

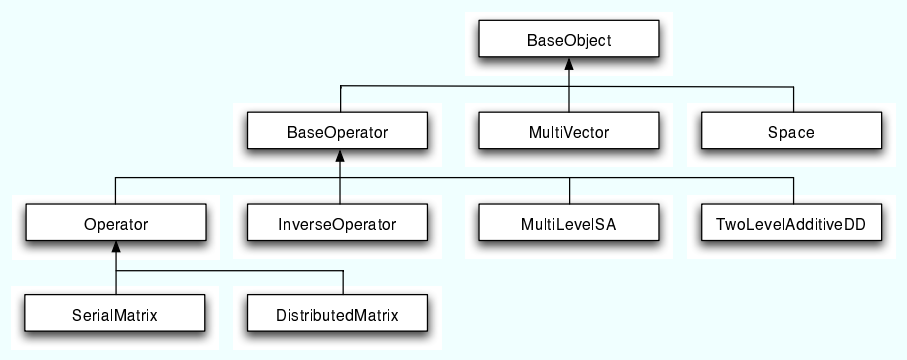

Figure 4: Inheritance diagram for MLAPI classes.

`MultiVector` all derived also from `CompObject` and `TimeObject`. These two classes, not described here, are used to count flops and track the time spent in given methods.

We note that MLAPI preconditioners can be easily wrapped as Epetra objects, and therefore used by any library of application that understand Epetra (for example, Krylov accelerators such as AztecOO, eigensolvers such as Anasazi, physics-based preconditioners as Meros, and the Trilinos API, TSF).

3.1 The `Space` Class

A `Space` is the fundamental MLAPI class. All distributed objects must have underlying spaces, which define the global dimension of the object (for example, the total number of elements in a vector) as well as the parallel layout (i.e., the distribution of these elements across the processors). The easiest way to define a `Space` is by specifying the number of global elements and/or the number of local elements,

```
int NumGlobalElements = 128;
int NumLocalElements  = 16;
Space S(NumGlobalElements);
Space S2(-1,NumLocalElements);
```

In both cases, a linear distribution is implicitly assumed[1]. If a non-linear distribution is required, one can simply write

```
int NumMyElements = 128;
int* MyGlobalElements;
// define here MyGlobalElements
Space S1(-1,NumMyElements,MyGlobalElements);
```

where `MyGlobalElements[i]` is the global ID or local node `i`.

The global ID of local node `i` is returned using the operator ():

```
int LID = 2;
int GID = S(LID);
```

[1]That is, processor 0 hosts all elements with a global ID from 0 to `NumLocalElements - 1`, processor 1 from `NumLocalElements` to `2 * NumLocalElements - 1`, etc.

As all memory is managed using smart pointers, the users can let a object go out scope even if this object has been used to define new objects. This is typically the case with `Space`'s. Let us consider the following function:

```
MultiVector foo(int NumGlobalElements)
{
  Space S(NumGlobalElements);
  MultiVector V(S);
  // do something on V, for example set all elements to 1.0
  V = 1.0;
  return(V)
}
```

Clearly, object `S` will go out of scope after returning from `foo`. However, the code will do the following: as `V` is created, a `Space` object, say `V2`, is defined within `V`, so that `V2 = V` (light-weight copy). All smart pointers in `V` are copied within `V2`, and `V` can go out of scope without damaging the returned vector (and without memory leaks).

3.2 The `MultiVector` Class

`MultiVector`'s are the MLAPI class for distributed double-precision vectors. Once a space (say, `S`) has been defined, vectors can be created as

```
MultiVector x(S), y(S); // specify the space in ctor...
MultiVector z;          // ... or create empty vector
```

The number of local elements in a vector is returned by method `GetMyLength()`, and the global number of elements by `GetGlobalLength()`. To set all the elements of a `MultiVector` to the same value, say 2, simply type `x = 2.0`. To modify a given (local) element, one can proceed as follows:

```
for (int i = 0 ; i < y.GetMyLength() ; ++i)
  y(i) = 1.0 * x(i);
```

that is, a reference to the $i-$element of the vector is returned using operator `()`. The sum of two vectors is simply

```
z = x + y; z = 2.0 * x - 3.0 * y;
```

The scalar product between `x` and `y` is `x * y`. The 2-norm is returned by method `Norm2()`; `Random()` populates the vector with random values; `Reciprocal()` replaces each element with its inverse. Method `GetValues()` returns the pointer to the internally stored double array. Note that efficient BLAS functions are used for copy and DAXPY operations.

Often, it is necessary to define a set of vectors, all sharing the same space. This can be obtained as follows:

```
int NumVectors = 3;
MultiVector w(S,NumVectors);
w.Random();
for (int i = 0 ; i < w.GetMyLength() ; ++i)
```

```
  for (int j = 0 ; j < w.GetNumVectors() ; ++j)
    cout << w(i,j) << endl;
```

All vectors in a multivector are stored in the same double array, consecutively.

It is also possible to create a `MultiVector` by providing an already allocated double array:

```
Space S(-1, NumMyElements);
double* ptr = new double[NumMyElements];
// here populate ptr as necessary
MultiVector v(S, ptr, ownership);
assert (v(i) == ptr[i]);
```

By setting parameter `ownership` to `true`, then `V` will take care of deleting memory when no objects refer to it[2]. By setting `ownership` to `false`, instead, the user will take care of deleting `ptr`.

3.3 The `Operator` Class

An `Operator` is a (linear or nonlinear) map between two `Space`'s, the domain space and the range space. An `Operator` object can be created by passing a pointer to an already `FillComplete()`'d `Epetra_RowMatrix`,

```
Epetra_RowMatrix* Epetra_A;
// create and FillComplete() here Epetra_A
Operator A(DomainSpace, RangeSpace, Epetra_A, ownership);
```

or by passing a pointer to an `ML_Operator` struct:

```
ML_Operator* ML_A;
// create and populate here ML_A
Operator A(DomainSpace, RangeSpace, ML_A, ownership);
```

Finally, `Operator`'s can be defined by manipulating already existing objects. If `A` and `B` are two existing `Operator` objects with suitable spaces, a new `Operator C` can be defined, for example, as `C = A + B`, `C = 1.0 * A + 3.0 * B`, `C = A * B`. For the particular case of triple matrix-matrix product, one can write `D = GetRAP (A, B, C)`.

As ML, MLAPI is not based on any particular matrix format. Rather, its matrix interface basically requires only a capability to query for the nonzero elements in a given local row. Therefore, it is easy to interface a given application with MLAPI.

An `Operator` can be applied to a `MultiVector`,

```
Space DomainSpace(5);
Space RangeSpace(5);
MultiVector x(DomainSpace); x = 2;
MultiVector y(RangeSpace);
Operator I = Identity(DomainSpace, RangeSpace);
y = I * x;
```

[2]It is supposed that `ptr` has been allocated using `new`.

For a symmetric positive definite matrix, the A-norm of a vector can simply be defined as `sqrt(z * (A * z))`. Some basic operations on matrices are also supported. For example, one can extract the diagonal of a matrix as a vector, then create a new matrix, containing this vector on the diagonal

```
// A is an Operator
MultiVector z = GetDiagonal(A);
Operator D = GetDiagonal(z);
```

Function `Eig()` can be used to compute the eigenvalues of an `Operator` (for serial runs only). This function calls LAPACK, therefore the `Operator` should be "small".

```
MultiVector ER, EI, V;
Eig(A, ER, EI, V);

for (int i = 0 ; i < ER.GetMyLength() ; ++i)
  for (int j = 0 ; j < ER.GetNumVectors() ; ++j)
    cout << "ER(" << i << ", " << j << ") = " << ER(i,j) << endl;
```

3.4 The `InverseOperator` Class

A `InverseOperator` is a (linear or nonlinear) map between two spaces, the domain space and the range space, whose application to a given `MultiVector` is meant to approximate the action of a given `Operator`. The most important difference between an `Operator` and an `InverseOperator` is that the latter has no definition of nonzero structure of matrix elements, while `Operator` does. The only mathematical method implemented by `InverseOperator` is `Apply()`.

`InverseOperator`'s usually define smoothers and coarse solvers in a multilevel preconditioner. At present, point relaxation smoothers (of Jacobi, Gauss-Seidel and symmetric Gauss-Seidel), several incomplete factorizations and a complete LU factorization can be used as smoothers and coarse solver. A simple example of usage here follows:

```
Operator A; // define the elements of A
Teuchos::ParameterList List;
List.set("smoother: sweeps", 2);
List.set("smoother: damping factor", 0.67);
InverseOperator invA(A, "symmetric Gauss-Seidel",List);
```

where 2 and 0.67 are the number of sweeps and the damping factor, respectively. A coarse solver can be defined as

```
InverseOperator coarse(A, "Amesos-KLU");
```

which means that the KLU solver of Amesos will be adopted to compute the LU factorization.

To apply the inverse of A using LU factorization one can write

```
InverseOperator invA(A,"Amesos-KLU");
```

To verify that $x = A^{-1}Ax$,

```
x = invA * (A * x) - x;
double NormX = sqrt(x * x);
```

3.5 The `SerialMatrix` and `DistributedMatrix` Classes

MLAPI offers two convenient matrix classes, one for serial computations only, and the other for serial or parallel computations. Both are derived from the `Operator` class, and overload the `()` operator to facilitate the insertion and modification of elements. The first class is `SerialMatrix`. An example of code to create a tridiagonal matrix is as follows:

```
Space S(n); // n is the size of the matrix
SerialMatrix A(S, S);
for (int i = 0 ; i < n ; ++i) {
  A(i,i) = 2.0;
  if (i)          A(i, i - 1) = -1.0;
  if (i != n - 1) A(i, i + 1) = -1.0;
}
```

The second class, `DistributedMatrix` can be used to create distributed matrices. Using this class, any process can set *any* element of the matrix (that is, both elements in the locally hosted rows, as well as elements hosted by another process). The following example creates a distributed tridiagonal matrix:

```
Space S(n); // n is the global size of the matrix
DistributedMatrix A(S, S);

// loop over all local elements
for (int i = 0 ; i < S.GetNumMyElements() ; ++i) {
  int j = S(i); // get global ID of local element i
  A(j, j) = 2.0;
  if (j)          A(j, j - 1) = -1.0;
  if (j != n - 1) A(j, j + 1) = -1.0;
}
```

After the insertion of all elements, the matrix structure must be "freezed" by calling method `FillComplete()`. This method computes the data structures required by the (distributed) matrix-vector product. Matrix elements cannot be added after the call to `FillComplete()`; though, the value of already existing elements can be modified.

By using the `SerialMatrix` class, the user can insert matrix elements at any time; however, the matrix-vector product has not been optimized. The `DistributedMatrix` class, instead, offers an optimized matrix-vector product, since the matrix is internally stored by using the Epetra library.

4 Using MLAPI

This section briefly explain how to use MLAPI[3]. We also refer to the Doxygen documentation for more details.

[3]Compilable codes can be found in `ml/examples/MLAPI`.

4.1 Initialize and Finalize the MLAPI Workspace

First, we need to initialize the MLAPI workspace, using `Init()` (which is automatically called by MLAPI is the user forgets to do so). The workspace should be cleaned using `Finalize()` to avoid memory leaks.

All MLAPI commands should be inserted in a `try/catch` block:

```
try {
  ... // here MLAPI stuff
}
catch (const int e) {
  cout << "Integer exception, code = " << e << endl;
}
catch (...) {
  cout << "problems here..." << endl;
}
```

4.2 Defining two-level Preconditioners

We now present how to define a 2-level additive preconditioner. Let `A` be the fine-level matrix. `C` the coarse level matrix, and `FineSolver` and `CoarseSolver` the fine level smoother and the coarse level solver, respectively. Let `P` be the prolongator from the coarse space to the fine space[4], then `R` is the transpose of `P`, and Galerkin projection is used to define `C`:

```
Operator A, C, P, R;
// define here A and P
R = GetTranspose(P);
C = GetRAP(R,A,P);
```

We will use symmetric Gauss-Seidel for the fine level, and LU for the coarse level:

```
InverseOperator FineSolver, CoarseSolver;
FineSolver.Reshape(A,"symmetric Gauss-Seidel");
CoarseSolver.Reshape(C,"Amesos-KLU");
```

The application of the preconditioner will read as follows:

```
void foo(MultiVector& b_f, MultiVector& x_f)
{
  x_f = FineSolver(b_f);     // smoother
  r_c = R * r_f;             // restriction to coarse
  r_c = CoarseSolver_ * r_c; // solver coarse problem
  x_f = x_f + P * r_c;       // sum correction
}
```

[4] As MLAPI is based on ML, the user can easily build a prolongator operator based on smoothed aggregation process.

4.3 Writing MLAPI Objects in MATLAB Format

It is often convenient to use MATLAB to analyze matrices and vectors. Distributed MLAPI objects can dumped to a single, MATLAB compatible ASCII file, by using class `MATLABStream`. Note that both serial and distributed objects are saved in just one file, which will contain the global object (for example, a distributed matrix will be dumped using global row and column ordering). `MATLABStream` behaves like a "normal" stream. Objects can be saved in file by using the operator `<<`.

Let `S, V`, and `A` be a `Space`, a `MultiVector` and an `Operator`. First, we have to define a `MATLABStream` object by specifying the file name,

```
MATLABStream matlab("mlapi.m");
```

Then, we specify the label of each operator, as this label will be used to define the name of the object in the output file,

```
S.SetLabel("S");
V.SetLabel("V");
A.SetLabel("A");
```

Finally, we can simply write

```
matlab << "% a string comment is allowed\n";
matlab << S;
matlab << V;
matlab << A;
```

We can also mix objects with MATLAB commands,

```
matlab << "plot(eig(A))\n";
```

The output file is automatically closed. Note that it is not possible to write on file an `InverseOperator` object, as this class only defines the action of the inverse of an operator on a given vector.

5 Concluding Remarks

Using C++ can greatly enhance clarity, reuse, and portability of numerical libraries. In our C++ library, MLAPI, the implementation details are completely hidden at the algorithm level, so that the resulting code greatly resemble the algorithm itself. This object oriented framework is based on a carefully designed set of classes and operator overloading, and allows the algorithm developer to write a parallel, efficient (C++) code in a MATLAB-like style. Operations on MLAPI classes are then automatically translated by a (standard) compiler into efficient code. The advantage of this approach is that the developer can focus on the algorithm itself, without spending time on details that can be fixed by the compiler. This reduces the time spent in writing and debugging the algorithm, at the only expense of a minimal increase in the CPU time. We believe that this is a negligible factor in the development and testing of new ideas, especially considering the time saved by the developer to write the code itself.

Acknowledgments

We thank all the ML developers, since MLAPI could not exist without ML, and could not be efficient and flexible if ML would not be so. MLAPI also relies of several other Trilinos [13] packages: Epetra [14], Ifpack [20], Amesos [19], Teuchos, AztecOO and Triutils.

References

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

[2] R. Barlett, *Teuchos::RefCountPtr beginner's guide: An introduction to the trilinos smart reference-counted pointer class for (almost) automatic dynamic memory management in C++*, Tech. Rep. SAND-3268, Sandia National Laboratories, June 2004.

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

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

[5] W. L. Briggs, V. E. Henson, and S. McCormick, *A multigrid tutorial, Second Edition*, SIAM, Philadelphia, 2000.

[6] J. Demmel, *LAPACK: A portable linear algebra library for supercomputers*, in Proceedings of the 1989 IEEE Control Systems Society Workshop on Computer-Aided Control System Design, IERR Press, Piscataway, NJ, 1989.

[7] J. Dongarra, J. D. Croz, S. Jammarlling, and I. Duff, *A set of level 3 basic linear algebra subprograms*, ACM Trans. Math. Softw., 1 (1990), pp. 1–17.

[8] G. Golub and H. van der Vorst, *Closer to the solution: Iterative linear solvers*, in The State of the Art in Numerical Analysis, I. Duff and G. Watson, eds., vol. 63, Oxford, 1997, pp. 63–92.

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

[10] W. Hackbusch, *Multi-grid Methods and Applications*, Springer-Verlag, Berlin, 1985.

[11] W. Hackbusch, *Multigrid Methods and Applications*, vol. 4 of Computational Mathematics, Springer–Verlag, Berlin, 1985.

[12] W. Hackbusch, *Iterative Solution of Large Sparse Linear Systems of Equations*, Springer-Verlag, Berlin, 1994.

[13] M. A. Heroux, *Trilinos home page.* http://software.sandia.gov/trilinos.

[14] M. A. Heroux, *Epetra Reference Manual*, 2.0 ed., 2002. http://software.sandia.gov/trilinos/packages/epetra/doxygen/latex/EpetraReferenceManual.pdf

[15] M. Hestenes and E. Steifel, *Method of conjugate gradients for solving linear systems*, Journal of Research, 20 (1952), pp. 409–436.

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

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

[18] Y. Saad and M. H. Schultz, *GMRES: a generalized minimal residual algorithm for solving nonsymmetric linear systems*, SIAM Journal on Scientific and Statistical Computing, 7 (1986), pp. 856–869.

[19] M. Sala, *Amesos 2.0 reference guide*, Tech. Rep. SAND-4820, Sandia National Laboratories, September 2004.

[20] M. Sala and M. Heroux, *Robust algebraic preconditioners with IFPACK 3.0*, Tech. Rep. SAND-0662, Sandia National Laboratories, February 2005.

[21] M. Sala, J. Hu, and R. Tuminaro, *ML 3.1 smoothed aggregation user's guide*, Tech. Rep. SAND-4819, Sandia National Laboratories, September 2004.

[22] B. Stroustrup, *The C++ programming language*, Addison-Wesley, Reading, MA, 1991.

[23] H. van der Vorst, *Parallel iterative solution methods for linear systems arising from discretized PDE's*, tech. rep., Special Course on Parallel Computing in CFD, no. R–807 in AGARD Reports, AGARD, Neuilly-sur-Seine, France, 1995, pp. 3–1–3–39., 1995.

[24] D. Vanderboorder and N. M. Josuttis, *C++ Templates: the complete guide*, Addison Wesley, Boston, 2003.

[25] P. Vanek, *Acceleration of Convergence of a Two Level Algorithm by Smooth Transfer Operators*, Appl. Math., 37 (1992), pp. 265–274.

[26] ——, *Fast Multigrid Solvers*, Appl. Math., 40 (1995), pp. 1–20.

[27] P. Vanek, J. Mandel, and M. Brezina, *Algebraic Multigrid Based on Smoothed Aggregation for Second and Fourth Order Problems*, Computing, 56 (1996), pp. 179–196.