Open access • Journal Article • DOI:10.1145/2331130.2331131

Object-Oriented Techniques for Sparse Matrix Computations in Fortran 2003

Salvatore Filippone, Alfredo Buttari

Institutions: University of Rome Tor Vergata, Centre national de la recherche scientifique

Published on: 01 Aug 2012 - ACM Transactions on Mathematical Software (ACM)

Topics: Sparse matrix, Linear algebra, Object-oriented design, Software design pattern and Modularity (networks)

Object-Oriented Techniques for Sparse Matrix Computations in Fortran 2003

Salvatore Filippone, Alfredo Buttari

Object-Oriented Techniques for Sparse Matrix Computations in Fortran 2003

SALVATORE FILIPPONE, University of Rome "Tor Vergata"
and ALFREDO BUTTARI, CNRS-IRIT Toulouse

The efficiency of a sparse linear algebra operation heavily relies on the ability of the sparse matrix storage format to exploit the computing power of the underlying hardware. Since no format is universally better than the others across all possible kinds of operations and computers, sparse linear algebra software packages should provide facilities to easily implement and integrate new storage formats within a sparse linear algebra application without the need to modify it; it should also allow to dynamically change a storage format at run-time depending on the specific operations to be performed. Aiming at these important features, we present an Object Oriented design model for a sparse linear algebra package which relies on Design Patterns. We show that an implementation of our model can be efficiently achieved through some of the unique features of the Fortran 2003 language. Experimental results show that the proposed software infrastructure improves the modularity and ease of use of the code at no performance loss.

Categories and Subject Descriptors: D.2.11 [Software Engineering]: Software Architectures—Data abstraction, information hiding; G.1.3 [Numerical Analysis]: Numerical Linear Algebra—Sparse, structured, and very large systems (direct and iterative methods); G.4 [Mathematical Software]: Algorithm design and analysis

General Terms: Mathematics of computing, Algorithms, Design

Additional Key Words and Phrases: Sparse Matrices, Object-Oriented Design

1. INTRODUCTION

Sparse linear algebra kernels account for a significant percentage of all scientific computations; the solution of sparse linear systems is an essential step in the solution of partial differential equations (PDEs), and the computation of eigenvalues of sparse matrices appears in fields as varied as vibration modeling, network connectivity analysis, webpage ranking and document-terms semantic analysis.

It is therefore not surprising that techniques for storing sparse matrices on digital computers have occupied researchers for a long time, not the least because it seems essentially impossible to settle on a single storage format suitable for all purposes.

We became involved with sparse matrices a number of years ago, focusing especially on linear system solvers and preconditioners for use on parallel machines [Filippone and Colajanni 2000; Buttari et al. 2007; D'Ambra et al. 2007; 2010]. From this experience we have been encouraged to reconsider the serial kernels and data structures on which the parallel solvers are founded, in the light of the language developments that are now included in widely used compilers; specifically we have applied object-oriented techniques in a more systematic fashion, exploiting the language support available in Fortran 2003 and subsequent standard levels [Metcalf et al. 2011].

In our discussion we focus ourselves on the data structures and techniques that are needed and useful for the purpose of implementing our libraries of parallel Krylov methods; of course this means we are looking at a specific cross-section of methods and operations, but the applications we cover are sufficiently important to warrant our effort. Achieving more generality than this, e.g providing full support for the equivalent of a computer algebra system, is outside the scope of the present work.

Author's addresses: Salvatore Filippone, Department of Mechanical Engineering, University of Rome "Tor Vergata", via del Politecnico 1, I-00133 Rome, Italy, salvatore.filippone@uniroma2.it. Alfredo Buttari, CNRS-IRIT, 118 Route de Narbonne, F-31062 Toulouse, France, alfredo.buttari@enseehit.fr

2. GENERAL OVERVIEW

We start our exposition by detailing the motivations and constraints of our development work; to begin with, we define an $N \times N$ matrix A to be "sparse" when

> The percentage of entries a_{ij} which are (identically) zero is large enough that it is convenient to avoid storing them in computer memory

Of course the specific storage format has to take into account the elementary operations needed to implement whatever computation is desired by the user. The above definition is attributed to Wilkinson; apparently [Davis 2007] he gave the dual of this definition, i.e. stating that "a matrix is dense if the percentage of zero elements is such as to make uneconomic to take advantage of their presence"; note that both these definitions are dependent on the algorithms the user wishes to employ.

In many important cases, e.g. for matrices originating from the discretization of PDEs, the average number of non-zeros per row k is much smaller than and independent of the matrix size N, and is determined by features of the originating problem such as the local topology of the discretization mesh.

In the following we will use the terms "representation" and "storage format" interchangeably to describe the detailed layout of the matrix coefficients a_{ij} in the computer memory, together with the mechanism(s) to identify their row and column indices.

The basic ideas of this paper revolve around the following facts:

— Sparse matrices do not have a "built in" representation in common programming languages;
— Different storage formats may be needed to match different computing architectures (e.g. scalar, super-scalar, vector, parallel etc.)
— Different storage formats entail different speeds in certain operations, such as matrix-vector products, and the difference may be quite large; these storage formats will usually achieve their efficiency by trading off memory space and pre-processing time for computational speed;
— No given storage format is likely to be uniformly better than the others across all possible kinds of operations, even when limited to a single underlying computing architecture;
— Scientific applications often need to perform different kinds of operations on sparse matrices depending on their different phases;
— Scientific applications may have a long lifetime; thus, given the progress in computer engineering, the underlying computing system and the associated optimal sparse matrix representation may change;
— A scientific application employing sparse matrices can usually be partitioned into the logical phases of handling the data distribution on a parallel computing environment, building the coefficients of the linear algebra problem to be solved, defining the solution strategy, computing the solution, post-processing the results.

From the above considerations we may extract the following set of requirements:

(1) It should be possible to dynamically choose the representation of the sparse matrix to fit the machine being employed with no changes in the application structure or behavior, except perhaps in the amount of memory to be paid to attain the desired execution speed;
(2) It should be possible to add a new storage format with no more effort than to compile the code for its object contents and for the methods operating on it, and link it in the application;
(3) It is desirable to dynamically change the storage format of a given matrix object during the execution of an application, so that in a sequence of operations each one can be performed with the best possible efficiency;

(4) The matrix representation should be opaque, except perhaps for one storage format which will be used to exchange data to/from the application side.

The above requirements pose a set of challenges to the developer, thus it should not be too surprising that the solution is quite complex.

A solution to at least some of these problems does not necessarily require the use of an object-oriented language; indeed, PETSc is written in C and version 2 of PSBLAS is written in Fortran 95 [Balay et al. 1995; Filippone and Colajanni 2000]. There is a long history of usage of object-based design in non object-oriented languages, including notable examples such as the X-windows system; therefore, the reader might wonder about the rationale for our foray into the usage of an object-oriented language. On the one hand this is absolutely correct; indeed, this idea might be taken further by saying that it is possible to implement an object-oriented language in another, or even in assembler, which is what a compiler is doing after all. What is actually very difficult is to satisfy simultaneously three contradictory requirements:

— It should be easy to add variations to storage formats (data types) already existing;
— The code that the user (or the library writer) is called upon to write should be centered around the desired mathematical operators, not the support infrastructure (as much as possible);
— The underlying language type checks, constraints and optimization capabilities should be preserved.

A typical strategy for handling objects in a non object-oriented language is to rely on "handles" of a predefined simple type, normally integers; this is used e.g. in X-windows and in one of the proposals for sparse BLAS [Duff et al. 2002]. However this forfeits the capability of type-checking implicit in the language, unless the application layer reconstructs these checks "a posteriori", a kind of reinvention of the wheel. If efficiency and run-time checks are preserved, as in our own Fortran 95 software, a different issue arises as to the amount of work needed to maintain the library over time; indeed, in this case we have to somehow reimplement the mechanism of dynamic dispatching, which means having (possibly in multiple places in the code), a list of dynamic types to choose our action from, according to the specific object instance at hand. This makes it impossible to add a new data structure without changing and recompiling the library itself; avoiding changes and recompilation is desirable, not just for a user testing new things, but also for the library developers themselves, in that it provides a "medium" level of integration for adding things incrementally over time, without the need to rework in depth for every change. An instance of this problem can be seen in the guts of our previous version were the dispatch of methods according to the contents of the various objects is implemented through flow-control mechanisms, which of course have to be updated in multiple places.

This work offers a two-fold contribution:

(1) We present an object-oriented model that allows users to handle sparse matrix objects independently of their internal state. Here the state has to be intended as the specific storage format used to represent the sparse matrix data although the proposed approach may be easily applied or extended to the case where the state is, for example, defined by mathematical properties of a matrix. The specific techniques we developed to achieve this objective can be interpreted in terms of Design Patterns [Gamma et al. 1995], a set of concepts well known in the software engineering community. More specifically, the reader will find plenty of similarities with the Builder, Prototype, Mediator and State design patterns;
(2) We show how it is possible to implement the Design Pattern concepts through the object-oriented features of the Fortran 2003 language that only recently have become accessible thanks to the availability of some complying compilers. We also show how

some original features of the Fortran 2003 standard (e.g., the `source` argument for data allocation described in Section 4), not available in other object-oriented languages such as C++, can be employed for this purpose.

This work is certainly not the first effort in sparse matrix formats [Duff et al. 1997; Filippone and Colajanni 2000; Duff et al. 2002]; our contribution seeks to lift existing sparse matrix techniques into a general and reusable framework that is easily extensible under the current generation of programming languages.

3. A SPARSE MATRIX CLASS

To begin tackling our challenge, we first define the objects that will contain the actual sparse matrix data, and whose methods will do the actual work required by the application. Because of the later encapsulation, we will occasionally refer to the objects of this section as the "internal representation" of a sparse matrix.

The sparse matrix objects are organized in a three-level inheritance hierarchy; this hierarchy has been designed taking into account our final goal of building iterative solvers for sparse linear systems. From a mathematical point of view a matrix is a representation of a linear operator mapping a vector space into itself; to the same operator there correspond different matrices depending on the choice of basis for the underlying vector spaces. In the context of the algorithms we are tackling, we are not usually interested in applying general transformations of basis, as these would destroy the sparsity patterns of the matrices, and therefore we assume a given basis for the underlying vector space.

If we consider the uses to which we put our objects, we are mostly interested in reusing methods as much as possible; therefore we identify attributes that are common to the different variety of fields onto which a matrix may operate:

— At the highest level of the inheritance hierarchy, a matrix is an object having certain features which can be represented in the same way for all possible base vector fields, such as number of rows/columns (i.e. dimension of vector space), number of non-zeros, possibly being triangular, with a unit diagonal, and so on;
— At a second level the derived classes are differentiated by the number field on which the vector spaces are based; in our implementation we have finite precision real and complex fields, and in both cases we have short and long precisions;
— At the third level of the hierarchy we finally have a "concrete" representation of a sparse matrix, holding in some format the set of nonzero coefficients.

A diagram of the class hierarchy is depicted in Fig. 1 and will be discussed in detail in the forthcoming sections; in this figure, according to the UML standard for Class Diagrams, the Composition relation is represented by a full diamond and the Inheritance relation by an empty triangle; further details on UML may be found e.g. in [Burkhardt 1997]. Before delving into the details of the class hierarchy we briefly recall here the basic syntactic features of the Fortran 2003 language used in the examples:

(1) The `type` keyword introduces the class definition; it is modified by the `extends` attribute to specify inheritance;
(2) The `contains` keyword separates the specification of attributes from the specification of methods;
(3) The `procedure` keyword introduces the specification of a binding (method, see below); the `pass` attribute identifies the dummy argument corresponding to the object for which the method was called; this is normally defined in the method interface, and in absence of the keyword it is the first argument. It is possible also to specify `nopass` to avoid passing the object variable to methods that do not need access to the object (such as the `get_fmt` method which prints a descriptive name of the dynamic type, see fig. 3).

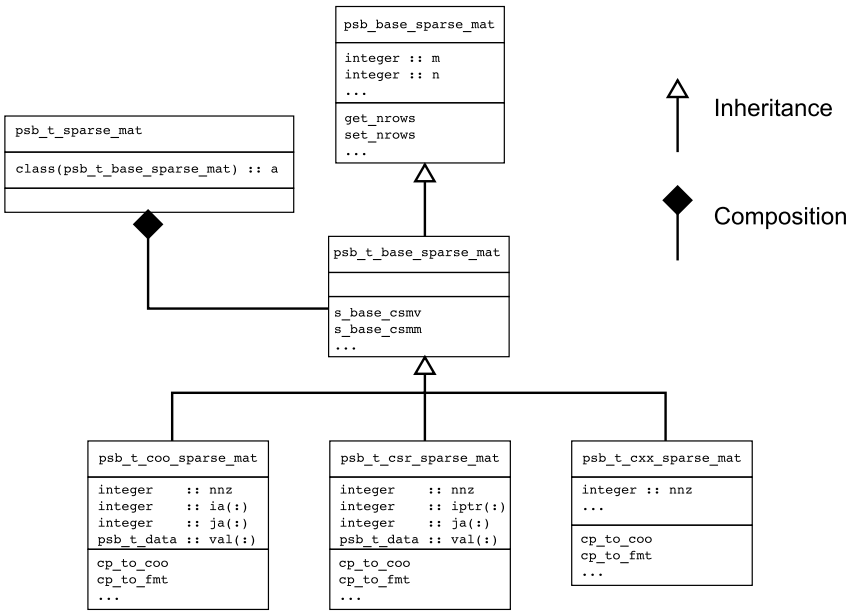

Fig. 1. UML class diagram

(4) A "binding" is the association of a method to an object; a binding allows the separation between the name by which the method is invoked through the object and the name of the subroutine implementing the method itself for the current (possibly derived) class; the name and the specific subroutine are linked by the `=>` syntactic element (see also Fig. 5);
(5) A binding can be generic, i.e. it can be an overloading set that is resolved by the compiler based on the language rules; the `generic` keyword adds a list of specific bindings to a generic set (see also Fig. 2).

Extensive discussions of the Fortran 2003 language features can be found in [Metcalf et al. 2011] and [Adams et al. 2009].

3.1. The `psb_base_sparse_mat` class

At the root of the type hierarchy there is a very basic object holding some information common to all type variants, such as number of rows and columns, whether the matrix is supposed to be triangular (upper or lower) and with a unit (i.e. assumed) diagonal, together with some state variables; see Fig. 2. Note that this base class is in common among all variants of real/complex, short/long precision; as such, it only contains information that is inherently integer in nature.

The methods associated with this class can be grouped into three sets:

— Fully implemented methods: some methods can be fully implemented at the `psb_base_sparse_mat` level. Two simple examples are:
  get_nrows. Extracts the number of rows (which is a `private` component of the data type, and therefore cannot be accessed directly);
  set_triangle. Record the fact that we are dealing with a triangular matrix.
— Partially implemented methods: Some methods have an implementation that is split between this level and the leaf level. A good example of this is the matrix transpose method `transp`: its implementation implies swapping the number of rows with the number of

```
type  :: psb_base_sparse_mat
  integer, private     :: m, n
  integer, private     :: state, duplicate
  logical, private     :: triangle, unitd, upper, sorted
contains
  ! methods that are fully implemented at this level
  procedure, pass(a) :: get_nrows;        procedure, pass(a) :: set_nrows
  procedure, pass(a) :: get_ncols;        procedure, pass(a) :: set_ncols
  procedure, pass(a) :: set_triangle
  ...
  ...

  ! methods that are partially implemented at this level
  generic, public    :: transp => base_transp_1mat, base_transp_2mat
  generic, public    :: transc => base_transc_1mat, base_transc_2mat
  ...
  ...

  ! methods whose interface can be defined at this level
  procedure, pass(a) :: get_neigh
  procedure, pass(a) :: allocate_mnnz;
  procedure, pass(a) :: reallocate_nz
  generic,   public  :: allocate => allocate_mnnz
  ...
  ...
end type psb_base_sparse_mat
```

Fig. 2. The `psb_base_sparse_mat` class

columns and complementing the lower/upper setting, and these operations clearly belong at the base level. However the transpose cannot be fully implemented without the knowledge of the exact storage details; therefore the derived class will implement the coefficient swapping, while calling the base class version of transpose to fix the common items[1]. This is facilitated by a unique Fortran 2003 feature: each extended type inherits (implicitly) a component having the parent type, and can thus invoke the parent's methods even if they are overridden.

— Other methods: There are a number of methods that are defined (i.e their interface is defined) but not implemented at this level. Two easy examples are the `allocate` and `free` methods, which clearly have to have detailed knowledge of storage components needed to hold the matrix coefficients. A more complicated example is the `csgetptn`: this is a method that returns a list of non-zero positions in a specified set of rows, i.e. a pattern with no coefficients attached. Since it only returns integer data it can be defined at a level common to all fields (real/complex), but it can only be implemented at the actual storage level. All methods in this set are coded at the base level to return a meaningful error message: if one such method has been invoked, it can only mean that one of the extended classes is missing an overriding implementation, and this is a programming error to be reported to the user.

Although sparse matrices will be declared as objects of type `psb_t_sparse_mat` (see Section 4) in the user's workspace, the `psb_base_sparse_mat` class provides a mean of simplifying the code development as methods that are defined at this level can be reused for all the descendant classes as well as inside the associated methods.

[1]Let us note explicitly that the transpose operator is intended for special cases where the actual transpose storage is really needed; this is opposed to the normal situation of specifying a transpose argument in the matrix-vector product, in a BLAS-like fashion, e.g. in the implementation of BiCG.

```
type, extends(psb_base_sparse_mat) :: psb_t_base_sparse_mat
contains
  procedure, pass(a) :: t_base_csmv
  procedure, pass(a) :: t_base_csmm
  generic, public    :: csmm => t_base_csmm, t_base_csmv
  ...
  procedure, nopass  :: get_fmt
  procedure, pass(a) :: t_csgetrow
  procedure, pass(a) :: t_csgetblk
  generic, public    :: csget => t_csgetrow, t_csgetblk
  ...
  procedure, pass(a) :: cp_to_coo;     procedure, pass(a) :: cp_from_coo
  procedure, pass(a) :: cp_to_fmt;     procedure, pass(a) :: cp_from_fmt
  procedure, pass(a) :: mv_to_coo;     procedure, pass(a) :: mv_from_coo
  procedure, pass(a) :: mv_to_fmt;     procedure, pass(a) :: mv_from_fmt
  procedure, pass(a) :: t_base_cp_from
  generic, public    :: cp_from => t_base_cp_from
  procedure, pass(a) :: t_base_mv_from
  generic, public    :: mv_from => t_base_mv_from

end type psb_t_base_sparse_mat

...

subroutine t_base_csmm(alpha,a,x,beta,y,info,trans)
  use psb_error_mod
  implicit none
  class(psb_t_base_sparse_mat), intent(in) :: a
  psb_t_data, intent(in)          :: alpha, beta, x(:,:)
  psb_t_data, intent(inout)       :: y(:,:)
  integer, intent(out)            :: info
  character, optional, intent(in) :: trans

  call psb_get_erraction(err_act)
  ! This is the base version. If we get here
  ! it means the derived class is incomplete,
  ! so we throw an error.
  info = 700
  call psb_errpush(info,name='t_base_csmm',a_err=a%get_fmt())

  if (err_act /= psb_act_ret_) then
    call psb_error()
  end if
  return

end subroutine t_base_csmm
```

Fig. 3. The `psb_t_base_sparse_mat` class

3.2. The `psb_t_base_sparse_mat` object

From the `base_sparse_mat` class we derive the classes for the various combination of real/complex data, with short/long precision; in our naming scheme the `t` in `psb_t_base_sparse_mat` is a placeholder for the four variations of short/long precision, real/complex data `s`, `d`, `c` and `z`, similarly to the naming scheme of LAPACK; the generic `psb_t_data` type will translate accordingly to the Fortran short/long precision, real/complex intrinsic data types.

Notice that at this point we have not yet defined any storage format, hence the class object itself does not have any additional members with respect to those of the base class; its structure is shown in Fig. 3.

No methods can be fully implemented at this level, but we can define the interface for the computational methods requiring the knowledge of the underlying field, such as the matrix-vector product, as shown in Fig. 3.

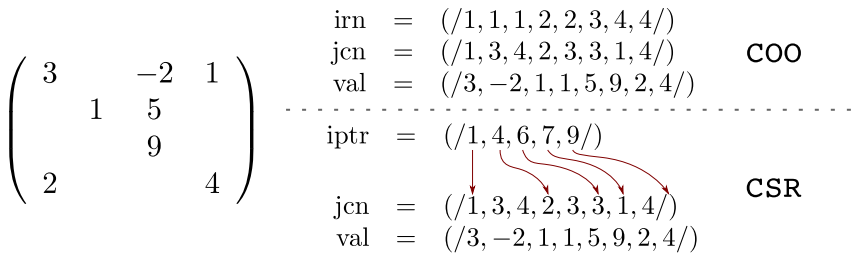

Fig. 4. The `COO` and `CSR` storage formats for sparse matrices.

```
type, extends(psb_t_base_sparse_mat) :: psb_t_coo_sparse_mat

  integer              :: nnz
  integer, allocatable :: ia(:), ja(:)
  psb_t_data, allocatable :: val(:)

contains

  procedure, pass(a) :: get_size => t_coo_get_size
  procedure, pass(a) :: get_nzeros => t_coo_get_nzeros
  procedure, pass(a) :: set_nzeros => t_coo_set_nzeros

  ! methods that overwrite those defined at the higher level
  procedure, pass(a) :: t_base_csmm => t_coo_csmm
  procedure, pass(a) :: t_base_csmv => t_coo_csmv
  ...
  ! methods that extend those defined at the higher level
  procedure, pass(a) :: base_transp_1mat => t_coo_transp_1mat
  procedure, pass(a) :: base_transc_1mat => t_coo_transc_1mat

end type psb_t_coo_sparse_mat
```

Fig. 5. The `psb_t_coo_sparse_mat` class

Again, this interface is defined, but is supposed to be overridden at the leaf level. The overloading capabilities of Fortran 2003 allow us to define a generic interface (e.g. `csmm`) which has the same "look" for all variants of data type/precision, as well as for products by rank-1 (i.e. vectors) or rank-2 (i.e. matrices) arrays[2] (see `t_base_csmm` and `t_base_csmv`).

3.3. The `psb_t_yyy_sparse_mat` object

We have finally arrived at the leaf of the inheritance hierarchy: we can now define a storage format; the `yyy` in the title will morph into a short symbolic name for the desired storage format in the actual class. The `COO`rdinates (`COO`) and `C`ompressed `S`parse `R`ow (`CSR`) storage formats will be used in the rest of this paper as reference examples; these two formats are illustrated in Fig. 4. In the `COO` we explicitly store each coefficient with a triple of coefficient value, row and column index, and we have a separate count of the nonzeros; with `CSR` we have an array of pointers to the boundaries of the rows into two other arrays containing the column indices and the values.

The class definition for `psb_t_coo_sparse_mat`[3] is shown in Fig. 5.

[2]Here the word "rank" is used in a technical sense from the Fortran language standard, meaning number of dimensions of an array, and does not refer to the mathematical concept of rank of a linear operator.

[3]Arguably the definition of a coordinate storage format could be made independently of the data type of the coefficients; however this would entail an inversion of the inheritance hierarchy, or a multiple inheritance chain, which is not supported in Fortran 2003.

As an example, in the `COO` storage format the implementation of the matrix transpose operator is quite simple: all that is needed is to swap the components `ia` and `ja` (and this can be done quite simply by using the `move_alloc` intrinsic), usually followed by a sorting of the entries in row-major order for efficiency reasons.

3.4. Conversion among different storage formats

To complete our work we still have to take into account one of the requirements for our model: we have to be able to perform conversions among different storage formats. We immediately see that providing conversion routines for all possible pairs of source/target storage formats is unfeasible for two reasons:

(1) A direct conversion for all pairs would generate a set of routines growing quadratically in size with the number of formats;
(2) The set of routines should be easily extensible, therefore we should not assume to know beforehand all possible storage formats in existence.

An efficient solution to these two problems is provided by the Mediator pattern [Gamma et al. 1995] whose implementation, in our case, consists in routing all conversions through one reference storage format, which thus gains the privileged status of being "generally known"; thus, the conversion among two different formats can be implemented in at most two steps via this reference format. The routines for format conversion from a source operand to a target operand are defined in eight variants, according to:

— Move vs. copy: the move operation will release the memory allocated for the source operand, which thus ceases to hold data, whereas the copy operation does the obvious thing; if at all possible, the move operation should avoid having both source and target memory simultaneously allocated at any given time;
— To vs. from: the "from" version is invoked via the target operand, whereas the "to" version is invoked via the source operand;
— Generic vs. reference storage format. Moves or copies can be perform to and from the reference format or a generic, unknown one. In the second case, a two-step procedure through the reference format will be used as described below.

The special storage format chosen for data exchange is the coordinate (`COO`): we store explicitly the number of non-zeros, and we have three vectors holding the nonzero coefficients, their row and column indices. This storage format is particularly convenient for the following operations:

— Renumbering the row/column indices;
— Adding/deleting entries.

Note that it is not necessary in this definition of the `COO` format for the entries to be in any particular order; however having them sorted may be critical in speeding up certain operations such as extracting a specific row or doing a triangular system solve.

How does the use of the `COO` storage solve the format conversion issue? The main problem here is that while we are writing the code for storage format `CXX` we have to prepare for conversion to storage format `CYY`, which will only be known at run-time. The only thing that we know about `CYY` is that, being an extention of the base type, it is required to provide methods for conversion to/from `COO`, just like the `CXX` code we are in the process of writing; therefore, if we employ a staging buffer, we can solve the problem with the code in Fig. 6.

In this way we can convert to `b`'s actual type (and storage format), because its specific details are known to its own implementation code invoked via the statement `call b%mv_from_coo(tmp,info)`.

It must be noted that the existence of this privileged format does not prevent the possibility to implement direct transitions among formats based on the actual type of `b` in those

```
subroutine t_mv_cxx_to_fmt(a,b,info)
  class(psb_t_cxx_sparse_mat), intent(inout) :: a
  class(psb_t_base_sparse_mat), intent(out)  :: b
  integer, intent(out)                        :: info

  type(psb_t_coo_sparse_mat) :: tmp

  call a%mv_to_coo(tmp,info)
  if (info == 0) call b%mv_from_coo(tmp,info)

end subroutine t_mv_cxx_to_fmt
```

Fig. 6. The code that moves a matrix `a` of type `CXX` into a generic format given by the argument `b`.

```
subroutine t_mv_coo_from_coo(a,b,info)
  implicit none
  class(psb_t_coo_sparse_mat), intent(inout) :: a
  class(psb_t_coo_sparse_mat), intent(inout) :: b
  integer, intent(out)                       :: info

  integer :: m,n,nz

  ...
  call a%psb_t_base_sparse_mat%mv_from(b%psb_t_base_sparse_mat)
  call a%set_nzeros(b%get_nzeros())

  call move_alloc(b%ia , a%ia  )
  call move_alloc(b%ja , a%ja  )
  call move_alloc(b%val, a%val )
  call b%free()
  ...

  return

end subroutine t_mv_coo_from_coo


subroutine t_mv_coo_to_fmt(a,b,info)
  implicit none
  class(psb_t_coo_sparse_mat), intent(inout) :: a
  class(psb_t_base_sparse_mat), intent(out)  :: b
  integer, intent(out)                       :: info

  ...
  call b%mv_from_coo(a,info)
  ...

  return

end subroutine t_mv_coo_to_fmt
```

Fig. 7. Format conversion methods

cases where the shortcuts provide performance gains significant enough to be worth pursuing. This requires separating the interface of methods from their actual implementation into different source files. In this case we are free to `USE` other modules in the source file containing the implementation without introducing cyclic dependencies between them. The task of including direct transition methods is considerably facilitated by a unique feature of the Fortran 2003 language: the `select type` statement which makes it possible to control the execution of instructions based on the dynamic type of a polymorphic entity.

```
subroutine t_cp_csr_to_coo(a,b,info)
  use psb_t_base_mat_mod
  implicit none

  class(psb_t_csr_sparse_mat), intent(in)  :: a
  class(psb_t_coo_sparse_mat), intent(out) :: b
  integer, intent(out)                     :: info

  integer, allocatable :: itemp(:)
  !locals
  logical              :: rwshr_
  Integer              :: nza, nr, nc,i,j,irw, idl,err_act

  info = 0
  nr  = a%get_nrows()
  nc  = a%get_ncols()
  nza = a%get_nzeros()

  call b%allocate(nr,nc,nza)
  call b%psb_t_base_sparse_mat%cp_from(a%psb_t_base_sparse_mat)
  do i=1, nr
    do j=a%irp(i),a%irp(i+1)-1
      b%ia(j)  = i
      b%ja(j)  = a%ja(j)
      b%val(j) = a%val(j)
    end do
  end do
  call b%set_nzeros(a%get_nzeros())
  return

end subroutine t_cp_csr_to_coo
```

Fig. 8. Format conversion methods

In Fig. 7 and 8 we show some of the conversion routines[4]. The methods in Fig. 7 are invoked through a `COO` object, therefore the first one is fully implemented, while the second one just throws the burden on the target object.

Requiring that each derived class must be able to construct a `COO` version of itself is not the absolute minimum necessary to implement the Mediator pattern; to implement this design pattern it would be sufficient to use a simple method that provides a view of a derived class; one such method is, for example, the `csget` mentioned in Fig. 3 and further described below. It must be noted that the resulting reduction in memory consumption comes at the price of a performance loss in terms of runtime; however, as discussed above, we are leaving the possibility for the user to implement a direct transition between any two formats, thus achieving memory and time efficiency at the cost of development effort.

We have yet to answer a fundamental question: How does the user let a sparse matrix come into existence? A very important design point was that storage formats should be opaque; this means that no knowledge of the internal representation should be needed outside the source code implementing the desired class. As a consequence it becomes necessary to provide methods for building a sparse matrix and for inspecting its contents relying on an exchange format. We have chosen to define the dual interfaces `csput` and `csget` assuming as a reference a simple list: given nz the number of nonzeros involved in a call, three vectors will hold for each $i \in (1 \ldots nz)$ the i-th coefficient, its row and its column indices, respectively. For `csput` these coefficients are fed into the matrix, whereas with `csget` we are inspecting the contents of a set of rows. Thus, `csput` and `csget` provide a `COO`-like virtual view of a given, opaque sparse matrix object.

[4]The actual library code is slightly more complex because of error handling procedures; at this level of discussion, they would only add clutter and have been removed from the code examples.

The alert reader will have noticed a possible redundancy in the previous methods with respect to the conversions to/from `COO` storage; this is not quite the case for the following reasons:

— The `csget` and `csput` methods are intended to be applied to a small subset of the rows/columns available, giving a purely "local" view;
— Some storage layouts can only be defined properly after having gathered "global" information on the whole sparse matrix;
— Having different interfaces for "global" and "local" conversions allows tuning for different usage conditions.

Of course there is nothing to prevent a format conversion from reusing the "local" methods if convenient; a draft example of how to build a sparse matrix with the local methods is shown in Fig. 10.

4. A CONTAINER CLASS FOR A SPARSE MATRIX

While the inheritance chain of the previous section achieves some of our objectives, it also leaves some questions unanswered, among them a major one: how to let a given sparse matrix object switch at run-time among different storage formats, some of them possibly unknown at the time the bulk of the library code is designed and implemented.

The solution is to add a layer of indirection: encapsulate the class of the previous section inside another class which is the one visible to the user, as already shown in Fig. 1 and detailed in Fig. 9. The only component of the user-level object is a polymorphic variable holding an actual sparse matrix; user-callable methods are simply implemented by calling the corresponding methods of the inner class (usually with a one-to-one correspondence).

The reader will recognize that this is essentially the "State" design pattern [Gamma et al. 1995]: the object has a well defined external appearance and set of methods, but the actual behavior changes according to the dynamic type of its (only) component.

The change in the internal state is accessible through the user-level `cscnv` method; thus the user may request a change in the storage format to suit the needs of the application. The method comes in two variants, according to whether it is working "in place" or whether it also performs a copy.

5. USAGE AND RESULTS

In this section we briefly describe the features unique to this Fortran 2003 version of our library; for general use considerations we refer the reader to the online documentation and to the code samples included.

5.1. How to add a new storage format

One of the key objectives was to make it easy for the library work with a new storage format, possibly developed by the end user (and thus completely unknown at the library writing time).

Naturally, the first order of business is to actually choose and implement a storage scheme, since there cannot be a way around the coding work required for implementing the various methods for operations such as the matrix-vector product[5]; as mentioned in the introduction one of our requirements is to make sure that this is (essentially) all there is to it. So, we assume that the user has chosen a storage format, has taken the code for one of the predefined formats[6] (e.g. `psb_t_csr_sparse_mat`) as a guide and has derived an entirely new class. How can the user plug this new class into the library framework?

[5]We are not discussing here the use of code generators; although quite possible, and indeed subject of related research [Martone et al. 2010], it is outside the scope of this paper.
[6]Not the `COO` format, since it has a privileged status in the MEDIATOR pattern.

```
type :: psb_t_sparse_mat

  ! The exact type of this component is only determined
  ! at runtime.
  class(psb_t_base_sparse_mat), allocatable  :: a

contains
!.... list of methods
..............
  procedure, pass(a) :: t_cscnv
  procedure, pass(a) :: t_cscnv_ip
  generic, public    :: cscnv => t_cscnv, t_cscnv_ip

end type psb_t_sparse_mat

...

subroutine t_cscnv_ip(a,info,type,mold,dupl)
  implicit none
  class(psb_t_sparse_mat), intent(inout) :: a
  integer, intent(out)                   :: info
  integer,optional, intent(in)           :: dupl
  character(len=*), optional, intent(in) :: type
  class(psb_t_base_sparse_mat), intent(in), optional :: mold

  class(psb_t_base_sparse_mat), allocatable  :: altmp

  ...
  if (count( (/present(mold),present(type) /)) > 1) then
    ! return an error
    call psb_error()
  end if

  if (present(mold)) then
    allocate(altmp, source=mold,stat=info)

  else if (present(type)) then
    select case (psb_toupper(type))
    case ('CSR')
      allocate(psb_t_csr_sparse_mat :: altmp, stat=info)
    case ('COO')
      allocate(psb_t_coo_sparse_mat :: altmp, stat=info)
    case default
      info = 136;  goto 9999
    end select
  else
    allocate(psb_t_csr_sparse_mat :: altmp, stat=info)
  end if
  ...

  call altmp%mv_from_fmt(a%a, info)
  call move_alloc(altmp,a%a)
  ...

  return

end subroutine t_cscnv_ip
```

Fig. 9. The `psb_t_sparse_mat` class

```
module psb_t_xxx_mat_mod
  use psb_t_base_mat_mod
  type, extends(psb_t_base_sparse_mat) :: psb_t_xxx_sparse_mat
       .... components of the type
  contains
    ....list of methods
  end type psb_t_xxx_sparse_mat
contains
   ......methods implementation
end module psb_t_xxx_mat_mod

program t_matgen
  use psb_sparse_mod
  use psb_t_xxx_mat_mod

  type(psb_t_sparse_mat)      :: a
  type(psb_t_xxx_sparse_mat) :: axxx

  ! Initialize an NRxNR matrix
  call a%csall(nr,nr,info)
  ! Fill in its contents
  do i=1, nr
     ....generate row I in IROW,ICOL,VAL
     call a%csput(nz,irow,icol,val,1,nr,1,nr,info)
  end do
  ! Assemble into XXX format
  call a%cscnv(info,mold=axxx)

  ! Do something with A

end program t_matgen
```

Fig. 10. Using the `psb_t_xxx_sparse_mat` class

Let us consider again the storage format conversion method `cscnv`, specifically the implementation of the "in place" version shown in Fig. 9; as we can see, there are two mutually exclusive optional arguments instructing the library code as to the desired storage format. The first selection mode, via the character string `type`, is used to select among pre-built storage variants that have already been defined at library writing time; as a minimum these include Coordinate (`COO`) and Compressed Storage by Rows (`CSR`), possibly others will be provided by default.

The interesting part of this example is the `mold` argument; the Fortran 2003 language provides the unique feature of sourced allocation, where the dynamic type of the allocated polymorphic object is set to be the same as the dynamic type of the source object specified in the `allocate` statement; this is a case of the "Prototype" design pattern having been included in the underlying language[7]. So, if the user has defined his/her own new `XXX` storage format by extending the `psb_t_base_sparse_mat` class, it is sufficient to declare in the user code a "mold" variable and pass it to the `cscnv` method, as shown in the code in Fig. 10. Note that, although a sourced allocation copies the contents of the source variable into the allocated one, for our purposes we are only using the dynamic type; thus the mold variable does not need to have any data associated with it, hence the name. When available, sourced allocation can be replaced with molded allocation, a feature of the Fortran 2008 standard already implemented in several compilers where the value of the mold variable is not copied into the allocated one.

Note that the various method invocations whose interfaces are specified by the `use psb_sparse_mod` statement have been pre-compiled, and thus have no knowledge of

[7]Note that this feature is not available natively in C++

the `psb_t_xxx_sparse_mat` derived type; the new type enters into the library space via the `mold` argument.

One final comment is in order: we have discussed so far how to build a sparse matrix from scratch; however it is often the case that the same pattern may be reused either in a different sparse matrix object, or in the same matrix at a later time with different coefficients. This is supported by the availability of both a cloning method and of a reinitialization method; if the reinitialization method is employed on an existing matrix in place of the allocation method `a%csall`, the filling loop in Fig. 10 will reuse the existing pattern while altering only the coefficients. Further details are available in the PSBLAS user's guide and accompanying sample programs.

5.2. Numerical results

The computational scientist interested in using these techniques will naturally wonder at this point how much of a performance impact will have to be paid for the flexibility acquired with the new framework.

Of course a precise answer to this question depends on the particular application and on the computing environment at hand, comprising both hardware and software, and care is needed to draw reliable conclusions.

In the context of this work we are helped by the fact that our code is an evolution of an existing one: indeed version 2 of PSBLAS is entirely implemented in Fortran 95, thus providing a convenient reference point. Moreover, the implementation of the computational kernels at the innermost level, such as the matrix-vector product in `CSR` or the sparse triangular system solution in `COO`, is essentially unchanged from version 2 to version 3; thus we are indeed as close as possible to measuring "just" the cost of the object model implementation. For this comparison to be meaningful it ought to be performed with the same compiler; thus we are tied to the compilers (and platforms) supporting all the features we have used. Here we will present results based on two such environments, the NAGware Fortran compiler version 5.2 for a 32 bit Linux installation, and to the IBM XLF version 13.1 compiler on an IBM POWER6; we should however notice that other compilers, including the Cray compiler version 7.3 and the GNU compiler version 4.6, also support enough features for our purposes.

Another aspect to be considered is that our library is designed for parallel computers and employs MPI; for the purposes of this comparison, however, we switch off the MPI code, thus leaving only empty stubs; we are effectively running in serial mode to avoid mixing communication issues into our performance data.

To evaluate our solution we have employed one of the standard test programs contained in our software, i.e. the `pargen` test program. The program embodies a stylized version of a PDE solver going through the following phases:

— Define the computation domain;
— Discretize the differential equation, transforming it in a linear system;
— Choose a (preconditioned) iterative method;
— Apply the iterative method to solve the equation.

The program is a "stylized" version in that the choices of computational domain and discretization technique are very simple: we are just applying finite differences on a unit cube with uniform step-size. Nonetheless, for the purpose of this analysis, the code structure is quite representative of a real application; translating this description into computational kernels, we have measurements for the following phases:

(1) Matrix build, comprising:
    (a) Allocation of data structures;
    (b) Generation of matrix coefficients;

Table I. CSR timings (in seconds) with the NAG compiler

	Version 2 (Fortran 95)				Version 3 (Fortran 2003)			
Size	Build	Prec	Solve	Iter	Build	Prec	Solve	Iter
8000	0.0110	0.0049	0.0197	0.0018	0.0106	0.0040	0.0180	0.0016
15625	0.0289	0.0109	0.0605	0.0047	0.0212	0.0096	0.0519	0.0040
27000	0.0419	0.0168	0.1101	0.0079	0.0317	0.0178	0.0967	0.0069
42875	0.0673	0.0269	0.1888	0.0118	0.0512	0.0296	0.1847	0.0115
64000	0.0876	0.0397	0.3147	0.0175	0.0794	0.0443	0.3842	0.0213
91125	0.1274	0.0590	0.5374	0.0256	0.1404	0.0689	0.6166	0.0294
125000	0.1753	0.1155	0.8793	0.0382	0.1558	0.0868	0.8938	0.0389
216000	0.3100	0.1547	1.7789	0.0684	0.2661	0.1467	1.7975	0.0691
343000	0.4928	0.2295	3.1669	0.1056	0.4325	0.2421	3.1617	0.1054
512000	0.7603	0.3384	5.0337	0.1480	0.6423	0.3574	5.2843	0.1554
729000	1.1148	0.4783	7.9254	0.2086	1.0530	0.5277	8.0699	0.2124
1000000	1.4296	0.6556	11.6826	0.2849	1.2382	0.6885	11.4146	0.2784

   (c) Assembly of sparse matrix;
(2) Preconditioner build;
(3) System solve.

The first set of results has been obtained with the NAG compiler version 5.2, on a machine equipped with an Intel Core2 T5600 CPU, with a 32 bit installation of Linux at the University of Rome "Tor Vergata". All linear systems have been solved with the BICGSTAB method and an ILU(0) preconditioner, with a tolerance of 10^{-9} for the relative (preconditioned) residual; in all cases the number of iterations and the residual on exit were the same for both versions, as is to be expected since the numerical properties of the inner kernels were not altered. All measurements were based on elapsed times, taken as "single shots"; as such they may suffer from a certain variability that is inherent in any such experiment.

In Table I we show the timings for the CSR storage format, specifically the times for matrix and preconditioner build, the total solve time and time per iteration. Comparing columns 4 and 5 with columns 8 and 9 we see clearly that the difference in solution time is very small, and thus the price paid for the OO design in the application of the computing kernels is quite acceptable. The second and sixth columns refer to the matrix build phase; here we have the somewhat surprising result that the new code is actually faster; in repackaging the code, we streamlined some components, and this was beneficial. The third and seventh columns contain the time to build the preconditioner, and here the Fortran 2003 code is only very slightly worse. If we look at the measurement for the COO storage format shown in Table II, we see a similar overall picture; one notable difference is that the preconditioner build times are consistently higher than the corresponding ones for CSR. This is due to a "dirty trick" in the code: when the input matrices are in CSR format we take a shortcut that enables faster execution for a very common case. Similar shortcuts can be added in various parts of the library if/when usage patterns warrant the effort. On the other hand, the Fortran 2003 preconditioner build times for COO are better than the Fortran 95 ones.

The second set of results has been obtained with the IBM XLF compiler version 13.1, on a POWER6 machine with the AIX Version 5.3 operating systems installed at the IDRIS (Institut du Développement et des Ressources en Informatique Scientifique) institute; all other parameters were identical to the previous set of test cases. Tables III and IV give the corresponding results. Again, the difference in performance is very small, but the distribution is somewhat different, because the Fortran 2003 code tends to be slightly slower in the matrix build, and somewhat faster in the preconditioner setup.

Our overall assessment is thus that the performance penalty for the Fortran 2003 implementation is close to non-existent, and is unlikely to be relevant in the context of a full, complex application; thus, we are reaping the benefits in code clarity and maintainability while keeping performance, a very nice outcome!

Table II. `COO` timings (in seconds) with the NAG compiler

	Version 2 (Fortran 95)				Version 3 (Fortran 2003)			
Size	Build	Prec	Solve	Iter	Build	Prec	Solve	Iter
8000	0.0116	0.0096	0.0290	0.0026	0.0101	0.0077	0.0260	0.0024
15625	0.0217	0.0179	0.0663	0.0051	0.0193	0.0152	0.0597	0.0046
27000	0.0410	0.0330	0.1252	0.0089	0.0336	0.0269	0.1153	0.0082
42875	0.0601	0.0539	0.2555	0.0160	0.0554	0.0488	0.2217	0.0139
64000	0.0930	0.0746	0.3913	0.0217	0.0780	0.0673	0.4089	0.0227
91125	0.1455	0.1170	0.6373	0.0303	0.1183	0.0982	0.6697	0.0319
125000	0.2201	0.1553	1.0101	0.0439	0.1567	0.1322	0.9575	0.0416
216000	0.3047	0.2491	1.9369	0.0745	0.2605	0.2203	1.8973	0.0730
343000	0.4963	0.4175	3.2125	0.1071	0.4230	0.3588	3.4458	0.1149
512000	0.7383	0.5979	5.6391	0.1659	0.6466	0.5410	5.8638	0.1725
729000	1.1197	0.8573	9.2316	0.2429	0.8811	0.7773	9.9702	0.2624
1000000	1.5056	1.2200	12.9558	0.3160	1.2673	1.0596	12.9526	0.3159

Table III. `CSR` timings (in seconds) with the XLF compiler

	Version 2 (Fortran 95)				Version 3 (Fortran 2003)			
Size	Build	Prec	Solve	Iter	Build	Prec	Solve	Iter
8000	0.0089	0.0072	0.0107	0.0010	0.0090	0.0055	0.0102	0.0009
15625	0.0169	0.0127	0.0244	0.0019	0.0167	0.0105	0.0237	0.0018
27000	0.0278	0.0255	0.0459	0.0033	0.0291	0.0192	0.0460	0.0033
42875	0.0454	0.0378	0.0878	0.0052	0.0456	0.0343	0.0926	0.0054
64000	0.0663	0.0572	0.1388	0.0077	0.0669	0.0448	0.1421	0.0079
91125	0.0935	0.0814	0.2382	0.0113	0.0897	0.0623	0.2438	0.0116
125000	0.1333	0.1090	0.4146	0.0180	0.1289	0.0907	0.4439	0.0193
216000	0.2322	0.1879	1.1538	0.0444	0.2253	0.1612	1.2090	0.0465
343000	0.3740	0.3137	2.2823	0.0761	0.3749	0.2572	2.1600	0.0720
512000	0.5781	0.4716	3.8668	0.1137	0.5683	0.4011	3.9524	0.1162
729000	0.8028	0.6677	6.2049	0.1633	0.8442	0.5699	6.1486	0.1618
1000000	1.1258	0.9369	9.4012	0.2293	1.1316	0.8169	9.9135	0.2418

Table IV. `COO` timings (in seconds) with the XLF compiler

	Version 2 (Fortran 95)				Version 3 (Fortran 2003)			
Size	Build	Prec	Solve	Iter	Build	Prec	Solve	Iter
8000	0.0082	0.0094	0.0109	0.0010	0.0086	0.0051	0.0102	0.0009
15625	0.0152	0.0173	0.0254	0.0020	0.0159	0.0100	0.0237	0.0018
27000	0.0267	0.0305	0.0487	0.0035	0.0273	0.0177	0.0460	0.0033
42875	0.0432	0.0467	0.0932	0.0055	0.0455	0.0304	0.0917	0.0054
64000	0.0645	0.0699	0.1473	0.0082	0.0632	0.0431	0.1422	0.0079
91125	0.0888	0.1016	0.2554	0.0122	0.0912	0.0614	0.2515	0.0120
125000	0.1230	0.1380	0.4798	0.0209	0.1243	0.0879	0.4307	0.0187
216000	0.2225	0.2451	1.2660	0.0487	0.2266	0.1647	1.2082	0.0465
343000	0.3541	0.4005	2.4657	0.0822	0.3873	0.2787	2.2623	0.0754
512000	0.5674	0.6196	4.2347	0.1246	0.5803	0.4165	4.0494	0.1191
729000	0.7897	0.8769	6.8445	0.1801	0.8163	0.5651	6.2538	0.1646
1000000	1.1076	1.1967	10.4982	0.2561	1.1461	0.7751	9.8937	0.2413

6. RELATED WORK

The attempt to standardize serial sparse computations is one of the tasks in the activities of the BLAS Technical Forum[8]; the interested reader is referred to [Duff et al. 1997; Duff et al. 2002].

An immense amount of research work has been devoted to the implementation of iterative solvers for sparse linear systems on distributed memory machines. One of the most complete works is the PETSc programming environment [Balay et al. 1995] which provides a C language implementation of an object-based design. The PETSc approach has many

[8]URL: `http://www.netlib.org/blast`

interesting features, such as the provision for setup and support routines, and ease of implementation of new iterative methods. We feel PSBLAS is somewhat complementary to PETSc, in that it addresses more closely the needs of the Fortran user community.

The Trilinos project [Heroux et al. 2005] aims at facilitating the development and maintenance of mathematical software by providing a unifying context for the usage of independent software packages. This is achieved by means of an object oriented software infrastructure where the Epetra class plays a fundamental role. This class provides an high level of abstraction for the definition of mathematical objects such as matrices or vectors. PSBLAS shares with Trilinos the effort to use advanced object oriented techniques to provide the user a mean of handling matrices regardless of their actual storage format. In particular, the Epetra row matrix classes `Epetra_RowMatrix`, `Epetra_BasicRowMatrix`, `Epetra_CrsMatrix` are organized in a hierachy that has a number of similarities with our hierachy for the internal objects `psb_base_sparse_matrix`, `psb_t_base_sparse_matrix`, `psb_t_csr_sparse_matrix` respectively; indeed, the Epetra user is encouraged by the documentation to take one of the classes that extend `Epetra_BasicRowMatrix`, and specifically `Epetra_JadMatrix` (in the documentation of version 10.6), as a guide to satisfy the requirements at hand. This is similar to our recommendation to employ `psb_t_csr_sparse_matrix` as a skeleton for implementing a new data structure tailored to the target machine architecture. There are two main differences that we can see. The first difference is that our library provides an external encapsulation layer enabling the implementation of the State design pattern, and therefore allowing any given sparse matrix class to change it's storage seamlessly, and to plug in the external format via the `mold` argument. The second difference is that the Epetra class takes into account the parallel data distribution, whereas our class hierachy is purely devoted to the implementation of the serial kernels; the kernels are already used in parallel, while we are currently investigating possible changes and extensions to the existing parallel structure.

Aztec [Hutchinson et al. 1998] is another software package developed at Sandia National Laboratories that simplifies the parallelization of sparse linear systems solution. Aztec's approach to domain partitioning, conversion from global to local data and keeping track of partitioning information is essentially identical to that of PSBLAS. The most important difference is that Aztec is designed primarily for C and Fortran 77 users, and does not provide object-oriented Fortran 90 capabilities; object oriented capabilities have been added by writing wrapper code in Trilinos.

Unlike the Aztec approach, object-oriented design methodologies are at the basis of the Diffpack project [Bruaset and Langtangen 1997]. Its authors propose the use of C++ instead of C and demonstrate that object-oriented numerical software is easy to use, flexible and maintainable, and can be computationally efficient even in the field of sparse matrix solvers.

Recent work by Rouson et al. [Rouson et al. 2010] describes the definition of new design patterns for multiphysics modeling and presents both a Fortran 2003 and a C++ approach to their implementation; sophisticated interfacing of Fortran 2003 with C/C++ is explored in [Rouson et al. 2010].

7. CONCLUSIONS

This paper presented a software framework for enabling extensible, efficient and portable implementations of sparse matrix computations.

Future developments will undoubtedly include the extension of this framework to other architectures; in particular, support for GPUs is the subject of related work [Barbieri et al. 2011], and multi-core support will probably require additional techniques, possibly using automatic code generation tools.

A side effect of our development has been the involvement with the GNU compiler development community; the GNU Fortran compiler version 4.6 has sufficient support for

Fortran 2003 to allow building and running our code. The PSBLAS library discussed in this paper is available from `http://www.ce.uniroma2.it/psblas`.

ACKNOWLEDGMENTS

John Holden of NAG Ltd. kindly provided a demo license for the NAGware 5.2 compiler; Jim Xia of IBM helped with issues related to the IBM XLF compiler.

We also thank the editor and the anonymous referees who provided us with many comments and suggestions, helping to improve our presentation.

REFERENCES

Adams, J. C., Brainerd, W. S., Hendrickson, R. A., Maine, R. E., Martin, J. T., and Smith, B. T. 2009. The Fortran 2003 Handbook. Springer.

Balay, S., Gropp, W., McInnes, L. C., and Smith, B. 1995. PETSc 2.0 user manual. Tech. Rep. ANL-95/11 - Revision 2.0.22, Argonne National Laboratory.

Barbieri, D., Cardellini, V., Filippone, S., and Rouson, D. 2011. Design patterns for scientific computations on sparse matrices. In Proceedings of EuroPar 2011, Parallel Processing Workshops. Bordeaux, France.

Bruaset, A. and Langtangen, H. 1997. Object-oriented design of preconditioned iterative methods in Diffpack. ACM Trans. Math. Softw. 23, 1, 50–80.

Burkhardt, R. 1997. UML: Unified Modeling Language. Addison-Wesley.

Buttari, A., D'Ambra, P., di Serafino, D., and Filippone, S. 2007. 2LEV-D2P4: a package of high-performance preconditioners for scientific and engineering applications. Appl. Algebra Engrg. Comm. Comput. 18, 3, 223–239.

D'Ambra, P., di Serafino, D., and Filippone, S. 2007. On the development of PSBLAS-based parallel two-level Schwarz preconditioners. Appl. Numer. Math. 57, 11-12, 1181–1196.

D'Ambra, P., di Serafino, D., and Filippone, S. 2010. MLD2P4: a package of parallel algebraic multilevel domain decomposition preconditioners in Fortran 95. ACM Trans. Math. Softw. 37, 3.

Davis, T. 2007. Wilkinson's sparse matrix definition. NA Digest 07, 12.

Duff, I. S., Heroux, M. A., and Pozo, R. 2002. An overview of the sparse basic linear algebra subprograms: The new standard from the blas technical forum. ACM Trans. Math. Softw. 28, 2, 239–267.

Duff, I. S., Marrone, M., Radicati, G., and Vittoli, C. 1997. Level 3 basic linear algebra subprograms for sparse matrices: a user-level interface. ACM Trans. Math. Softw. 23, 3, 379–401.

Filippone, S. and Colajanni, M. 2000. PSBLAS: a library for parallel linear algebra computations on sparse matrices. ACM Trans. on Math Softw. 26, 527–550.

Gamma, E., Helm, R., Johnson, R., and Vlissides, J. 1995. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley.

Heroux, M. A., Bartlett, R. A., Howle, V. E., Hoekstra, R. J., Hu, J. J., Kolda, T. G., Lehoucq, R. B., Long, K. R., Pawlowski, R. P., Phipps, E. T., Salinger, A. G., Thornquist, H. K., Tuminaro, R. S., Willenbring, J. M., Williams, A., and Stanley, K. S. 2005. An overview of the Trilinos project. ACM Trans. Math. Softw. 31, 3, 397–423.

Hutchinson, S., Prevost, L., Shadid, J., Tong, C., and Tuminaro, R. July 1998. Aztec User's Guide. Version 2.0. Tech. rep., Sandia National Laboratories.

Martone, M., Filippone, S., Tucci, S., Paprzycki, M., and Ganzha, M. 2010. Utilizing recursive storage in sparse matrix-vector multiplication, preliminary considerations. In Proceedings of CATA 2010. International Society for Computers and Their Applications, Honolulu, HI, USA.

Metcalf, M., Reid, J., and Cohen, M. 2011. Modern Fortran Explained. Oxford University Press.

Rouson, D. W. I., Adalsteinsson, H., and Xia, J. 2010. Design patterns for multiphysics modeling in Fortran 2003 and C++. ACM Trans. Math. Softw. 37, 1, 1–30.

Rouson, D. W. I., Lemaster, M. N., and Morris, K. 2010. Large-scale integration of object-oriented Fortran 2003 and C++ via ForTrilinos and CTrilinos. In International Conference on Computational Science. Amsterdam, Netherlands.