A Key for Reusable Parallel Linear Algebra Software

Eric Noulard [a,b] and Nahid Emad [b]

[a] *Société ADULIS – 3, rue René Cassin – F-91742 Massy Cedex – France*

[b] *Université de Versailles St-Quentin-en-Yvelines – Laboratoire PRiSM – Bâtiment Descartes – F-78035 Versailles Cedex – France*

Abstract

We propose an object oriented design which enables very good code reuse for both sequential and parallel linear algebra applications. A linear algebra class library called LAKe is implemented using our design method. We introduce a new reuse mechanism called *matrix shape* which enables us to derive the implementation of both the sequential and the parallel version of the iterative methods of LAKe. We show that polymorphism is insufficient to achieve our goal and that both genericity and polymorphism are needed. We propose a new design pattern as a part of the solution. Some numerical experiments validate our approach and show that efficiency is not sacrified.

Key words: OO design, genericity, parallel and sequential code reuse, Krylov subspace methods.

Introduction

In the area of numerical computing many people would like to use parallel machines in order to solve large problems. Parallel machines like modest sized SMPs or workstations clusters are becoming more and more affordable, but no easy way to program these architectures is known today. A message passing library like MPI[6] brings us a standard for parallel message passing programming. However code written with MPI is hard to understand for non parallel programming experts and consequently hard to maintain and reuse.

In order to evaluate the object oriented design as a mean to reuse most of the sequential and parallel software components we develop our previous study [15]. This goal is reached in two related steps, first by encapsulating parallelism in such a way that the user may manipulate parallel objects as if she/he were

working with sequential ones and then by finding a code reuse scheme which enables the application developer to use either the sequential objects or their parallel counterpart. In the domain of linear algebra, the family of Krylov subspace methods allows to solve either eigenproblems [16] or linear systems [17]. These methods share many properties and are good candidates to code reuse and parallelization.

In section 1, we present the block Arnoldi method as a particular case of block Krylov methods and recall the usual way to parallelize such a method. We list their needed elementary operations for either a sequential or a parallel implementation. In section 2, we first develop our goals in terms of code reuse and then present different design solutions with illustrative implementation in C++. We show the limit of polymorphism and dynamic binding as a reuse scheme when compared to genericity. Finally section 4 presents some numerical experiments.

1 The Block Krylov Methods

The orthogonal projection methods are often used to compute a few eigenelements of a large sparse matrix A (may be noted A). They approximate r eigenelements of A by those of a matrix of order m obtained by orthogonal projection onto a m-dimensional subspace $K_{m,V}$ of $\mathbb{R}^n$ with $r \leq m \ll n$. These methods approximate a solution (λ_i, u_i) of the eigenproblem:

$$\mathsf{A}u = \lambda u, \text{ with } \mathsf{A} \in \mathbb{R}^{n \times n}, \lambda \in \mathbb{C}, u \in \mathbb{C}^n \tag{1}$$

by a pair $\lambda_i^{(m)} \in \mathbb{C}$, $y_i^{(m)} \in \mathbb{C}^m$ satisfying :

$$(\mathsf{H}_m - \lambda_i^{(m)} I) y_i^{(m)} = 0 \tag{2}$$

where the $m \times m$ matrix H_m is defined by $\mathsf{H}_m = \mathsf{V}_m^H \mathsf{A} \mathsf{V}_m$ with V_m the $n \times m$ matrix whose columns are an orthogonal basis of $K_{m,V}$, V_m^H the conjugate transpose of V_m and $u_i^{(m)} = \mathsf{V}_m y_i^{(m)}$. Thus, in order to solve the problem (1), we first build an orthogonal basis of $K_{m,V}$ and then solve the problem (2). The couple $(\lambda_i^{(m)}$, $u_i^{(m)})$ is an approximated solution of (1) and is called a pair of Ritz elements (Ritz value and Ritz vector) of A on the subspace $K_{m,V}$. A *Krylov subspace method* is a one for which $K_{m,V}$ is defined by $K_{m,V} = Span(V, AV, \cdots, A^{m-1}V)$ where V is spanned by a set $\{v_1, \cdots, v_s\}$ of initial guesses. Among these methods the *Block Krylov subspace methods* come from the choice $s > 1$.

In general, the accuracy of the computed Ritz elements is not satisfactory. In order to obtain desired accuracy, the computed Ritz vectors will be used to build a new set of initial guesses. This set of vectors is used to restart the above process in the `iterative` version of the projection method. The reader can find some restarting strategies in [16,17,21].

In this paper we consider a particular case of the block Krylov subspace methods called the iterative block Arnoldi method to compute some Ritz elements of a large non-symmetrical sparse matrix. This method is a block version of the iterative Arnoldi method (when $s = 1$) [16]. We briefly present the algorithm constituting the method in the next section and then analyze the elementary operations involved and their usual parallelization strategies.

1.1 The Block Arnoldi Method

The block Arnoldi method enables to find the eigenvalues whose multiplicity is less or equal than the block size s. This projection method first reduces the original matrix A into an upper block Hessenberg matrix H_m using a *Block Arnoldi Process*. Let V_1 be the orthonormal matrix whose columns are $v_1, \cdots, v_s$. The block Arnoldi process, given by algorithm 1.1, generates a set of orthonormal matrices $V_1, \cdots V_m$.

ALGORITHM 1.1 **(Block Arnoldi Process [MGS version])**

Iterate: For $j = 1, 2, \ldots, m$

(a) $W = \mathsf{A}V_j$

(b) *For* $i = 1, \ldots, j$

$H_{ij} = V_i^H W$

$W = W - V_i H_{ij}$

End For i

(c) $[V_{j+1}, H_{j+1,j}] = \mathtt{QR}(W)$

End For j

The $H_{i,j}$ s-size matrices computed by this process are the blocks of H_m. The eigenvalues of this matrix approach some of A. When the Block Arnoldi Process terminates, the matrices H_m, A and $\mathsf{V}_m = [V_1 \ldots V_m]$ verifies the Block Arnoldi Reduction equation (3).

$$\mathsf{A}\mathsf{V}_m = \mathsf{V}_m\mathsf{H}_m + V_{m+1}H_{m+1,m}E_m^H \tag{3}$$

The Block Arnoldi Process is only one step (projection step) of the iterative

Block Arnoldi method whose complete algorithm is given by algorithm 1.2, where V_m is the $n \times m \cdot s$-size matrix whose block of columns are $V_1, \ldots, V_m$.

ALGORITHM 1.2 **(Iterative Block Arnoldi Method)**

1) Initialization:

Choose r the number of wanted eigenvalues and m the size of the Block Krylov subspace.

2) Choose an orthogonal starting matrix V_1 of size $n \times s$.

3) Iterate:

(a) Projection step: execute a Block Arnoldi Process with algorithm 1.1. This step produces the Arnoldi reduction equation (3).

(b) Ritz computation step, compute:

- *the eigenelements $(\lambda_i^{(m)}, y_i^{(m)})$ of H_m,*
- *the r wanted Ritz elements $(\lambda_i^{(m)}, u_i^{(m)})$ of A.*

(c) compute the norm of the residuals of the wanted Ritz elements.

(d) compute a stopping criteria

(e) restart the method

- *go to step 2) with a new matrix if explicit restart is used*
- *update Arnoldi reduction and go to 3.a) if implicit restart is used*

1.2 Elementary Operations and Parallelization

We must mention that all the Krylov subspace methods share many properties such as using BLAS operations on sparse and/or dense matrices and may support different kind of restart during the iterative process ... Thus a careful modular design of the block Arnoldi method for eigenvalue problems may enable a lot of code reuse for building for example a GMRES solver [18] or easy variation of restart strategies. Moreover with a good design, iterative methods should be parallelized easily and most of the sequential code directly reused. Our target parallel machines are distributed memory architectures which could be either a supercomputer or any cluster. In this context, the classical way to parallelize Krylov subspace iterative methods is to distribute the large vectors and/or matrices and replicate the small ones on the processors. Then we decompose and distribute all the matrices of size n, that is the $n \times n$ sparse matrix A, Krylov subspace basis V_m of size $n \times m \cdot s$ and possibly the temporary variables like W of algorithm 1.1. But the matrix H_m and all the m-sized matrices used in step 3.b) algorithm 1.2 are replicated on

processors. From a design point of view it should be interesting to encapsulate parallel code in a basic parallel matrix class which has the same interface as the sequential one and implement the iterative method using those matrix classes. We will further develop this aspect in section 2.

We list hereafter the necessary elementary operations used by iterative methods. We recall that the Block Arnoldi Method is representative of Krylov subspace methods for the necessary elementary operations. From algorithms 1.1 and 1.2 and our parallelization strategy we may summarize those requirements as following (for $m, n, p \in \mathbb{N}$) :

(1) algebraic operations
   (a) Full Matrix SAXPY: $Y = \alpha X + \beta Y$ with $Y, X \in \mathbb{R}^{m \times n}$ and $\alpha, \beta \in \mathbb{R}$. The matrices may be distributed.
   (b) Full Matrix Product: $Y = \alpha A \cdot X + \beta Y$ with $Y \in \mathbb{R}^{m \times p}$, $A \in \mathbb{R}^{m \times n}$, $X \in \mathbb{R}^{n \times p}$ and $\alpha, \beta \in \mathbb{R}$. The matrices may be distributed.
   (c) Sparse Matrix by Full Matrix product: $Y = \alpha A \cdot X$ with $X, Y \in \mathbb{R}^{n \times p}$ and $A \in \mathbb{R}^{n \times n}$, A is **sparse**. These matrices may be distributed and A may be available only as a function which performs matrix product.
(2) sub-ranging or aliasing
   (a) sub-ranging: $Y = A(i_1 : i_2, j_1 : j_2)$, with $A \in \mathbb{R}^{m \times n}$ and $Y \in \mathbb{R}^{(i_2 - i_1 + 1) \times (j_2 - j_1 + 1)}$. The Matrices may be distributed.
   (b) point addressing: $A(i, j) = \alpha$ with $A \in \mathbb{R}^{m \times n}$ and $\alpha \in \mathbb{R}$. This operation is authorized only on full matrix.
(3) memory allocation
   (a) allocate, deallocate and (re-)distribute $A \in \mathbb{R}^{m \times n}$.

2 Designing a Reusable Software

The block Arnoldi method presented in the previous section led us to design a class library called LAKe (**L**inear **A**lgebra **Ke**rnels). The main goal of this library is the use of the *same code for the sequential and parallel* version of the iterative methods. In that way we only maintain a single code which is not cluttered with unreadable parallel code. The following sections will demonstrate how to achieve this goal. We first present the sequential design of LAKe, then we explain why polymorphism and dynamic binding are not sufficient to reach our goal. We finally demonstrate how genericity is the key of the solution. We point out that our design is the first one to reach such a reuse goal.

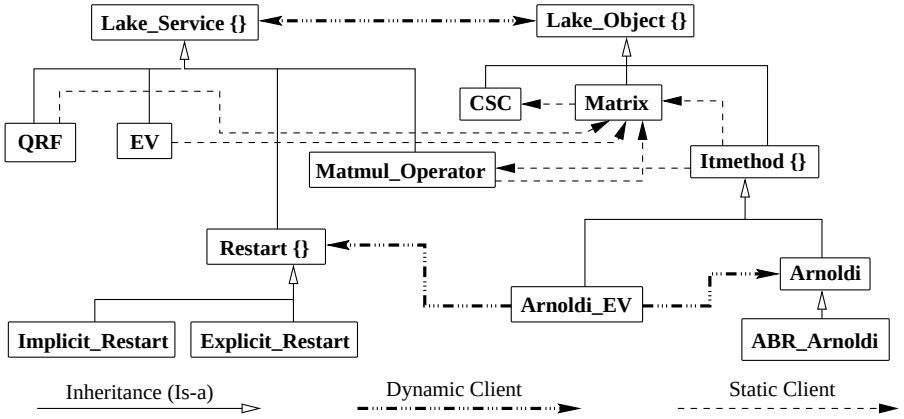

Fig. 1. The LAKe architecture

2.1 LAKe Architecture

The LAKe architecture is presented in figure 1. Each box represents a class, whose features have been omitted for the sake of clarity. Boxes are linked by arrows which describe the relation between the classes. Plain arrows stand for inheritance or the **is-a** relation [13, p. 811]. Each class name recall the role it play in algo. 1.1 and 1.2, for example the `Itmethod` class represents an abstract iterative method while the `Arnoldi` class implements the Block Arnoldi Process from algo. 1.1. Thus `Arnoldi` class inherits from `Itmethod` since Arnoldi *is an* iterative method. This means that each feature defined in `Itmethod` will be a feature of `Arnoldi` too. Dashed arrows represent the **client** relation. A class `A` is a **client** of another class `B` if it uses at least an object of type `B`. The `Arnoldi_EV` class implements the Iterative Block Arnoldi Method described in algo. 1.2 then it uses the `Arnoldi` class. The client relation is *dynamic* if the relation is established at runtime and is *static* if it may be established at compile time. As an example the `Arnoldi_EV` class is a dynamic client of the `Restart` whose daughter classes `Implicit_Restart` and `Explicit_Restart` implement varying strategies of restart.

Polymorphism [13, p. 28] is the uniform handling of different objects which share some parts of their interface (for example by inheriting from the same base class). Polymorphism associated with dynamic binding [13, p. 29] enables the polymorphically handled object to act differently at runtime. Dynamic binding may be illustrated by the `Arnoldi_EV` class which uses (is a *dynamic* client of) `Arnoldi`, but at runtime `Arnoldi_EV` may use an `ABR_Arnoldi` (algorithmic modification of `Arnoldi` in order to handle varying block size) object transparently.

Polymorphism seems an obvious way to parallelize iterative methods without touching the code. Each iterative method is a client of the `Matrix` class which implements all the operations described in §1.2. We only need to build a `DMatrix` class that is derived from `Matrix` which redefines the needed features in order to have a parallel implementation. Our iterative methods will then use the distributed matrix class `DMatrix` polymorphically, which means without modifying the code of the iterative method itself. We will now show why polymorphism is not sufficient to reach our aim. The problem in object-oriented language is the implicit assumption that inheritance defines a subtype relation. The following definition specifies the notion of subtype:

Definition 1 (Subtype) *A type T' is a* subtype *of type T, also noted $T' \leq T$ iff every function that expects an argument of type T may take an argument of type T'.*

Contravariance comes out when trying to subtype functions. The idea we must keep in mind when trying to subtype functions is: *g is a subtype of f iff everywhere f is expected one may use g.* The rule for function sub-typing is given in definition 2. This result is not so natural and detailed examples and theoretical references may be found in [10].

Definition 2 (Contravariance) *Let $TA \rightarrow TR$ be the type of a function taking an argument of type TA and returning an argument of type TR. The subtype rule for functions is: $TA' \rightarrow TR' \leq TA \rightarrow TR$ iff $TR' \leq TR$ and $TA \leq TA'$. We say that the output type of a function is* covariant *since it varies in the same way the type of the function does, but the type of the input arguments is* contravariant *since the subtype relation is inverted.*

In the following "`A::`" prefix means that the feature is defined in class `A`. The contravariance problem arises at step b) of algorithm 1.1 when we need to perform the algebraic operation $H_{ij} = V_i^H W$ on the distributed matrices V_i and W. Let `TA` be `Matrix` and `TB` be `DMatrix` a subtype of `TA`. The operation is performed by a call to the method `TA::tmatmul(TA*,TA*)` which has been redefined in the distributed matrix class as `TB::tmatmul(TB*,TB*)`. The call is shown at line 5–6 of figure 2. At this point the wrong method is called because the subtype relation on functions implies that `TB::tmatmul(TB*,TB*)` is **not** a subtype of `TA::tmatmul(TA*,TA*)` since the inputs arguments must be contravariant.

The only proper method redefinition is: `TB::tmatmul(TA*, TA*)`. Thus the type of the arguments must be checked dynamically. This process is called *dispatch* of the arguments: single, double and multiple dispatch when doing it for one, two or more arguments. The *multiple dispatch* problem is a classical

```
1 Arnoldi::gsloop(int jbeg, int jend) {
2   [...]
3   W_ = V1_.copy();
4   [...]
5   // call to Matrix::tmatmul(Matrix*,Matrix*)
6   Hij_->tmatmul(Vi_,W_);
7   [...]
8 }
```

Fig. 2. Bad dispatch call

OO problem and has been solved in the past. It is generally not integrated in OO languages since it is costly. It was noticed and solved *in the same linear algebra context* by F. Guidec in [9, pp. 96–99] for the Paladin linear algebra library. His solution relies on finding the dynamic type of each argument of the concerned function by using dynamic type control.

We propose an improved solution as a design pattern called *Service Pattern*. It has two advantages over the Paladin solution: the dispatch of an argument is only done when it changes and the dispatch may be done for several operations using the same arguments.

Remark 3 (Wrong Design) *One may argue that the problem comes from misconception in the* `Matrix` *class. Providing a low level interface, like pointwise access* $A(i,j)$ *to both* `Matrix` *and* `DMatrix` *would enable the implementation of* `tmatmul` *operation in* `Matrix` *and its reuse in* `DMatrix`*. This would be very inefficient in parallel environment since it would generate one message for each element access.*

Remark 4 (Anchored type) *Some languages like Eiffel offer another solution to the contravariance problem:* anchored type. *A variable has an anchored type if its type is specified to be* like *another type which may be currently defined. This authorizes covariant redefinition of member method. We do not consider this solution since C++, our target language, does not support it. Moreover anchored types have their own drawbacks [13, pp. 621–642].*

2.3 Service Pattern Solution

The *Service Pattern* reifies the method which must dispatch its argument: the `Matrix::matmul` method becomes a `Matmul_Operator` service class. The *Service pattern* is inspired from the *Visitor Pattern* [7, p. 331], and it can use either dynamic type control or the visitor pattern to dispatch the argument. The *Service Pattern* represents a set of operations which register (or connect) their arguments one by one. The service has an internal state which specifies which operations may be called on the service. Each operation offered by the

```
1 // Declaration
2 Matrix           *A, *B, *C; // pointer to matrix objects
3 Matrix           *Q, *R, *X; // pointer to matrix objects
4 DMatmul_Operator   Matmul;   // a distributed Matrix Multiplication service
5 QRF                QRFizer;   // a QR factorizer service object
6 int                rank;
7
8 // polymorphic assignment
9 A = new DMatrix(); B = new DMatrix(); C = new DMatrix();
10 Q = new Matrix(); R = new Matrix(); X = new Matrix();
11 [...]
12 // Compute A = B * C
13 Matmul.connect(A,"Y"); // register A as "Y"
14 Matmul.connect(B,"A"); // register B as "A"
15 Matmul.connect(C,"X"); // register C as "X"
16 Matmul.matmul();       // compute A = B * C
17 // Compute C = B^{T} * A
18 Matmul.disconnect("Y"); Matmul.connect(C,"Y");
19 Matmul.disconnect("X"); Matmul.connect(A,"X");
20 Matmul.tmatmul();      // compute C = B^{T} * A
21 Matmul.disconnect();
22 [...]
23 // Compute QR factorization of X
24 QRFizer.connect(X);    // register X as the default argument
25 QRFizer.factorize();
26 QRFizer.get_R(R);      // get the R factor of QR factorize
27 rank = QRFizer.rank(); // get the rank of QR factorization
28 QRFizer.compute_Q();   // compute Q factor which overrides X
29 QRFizer.disconnect();
```

Fig. 3. Examples of LAKe Services

service will check this internal state and raise an error if the service is not in a convenient state to serve this operation. This means that at runtime a method of the service can be called if the state of the service object authorizes it. Two examples which illustrate the use of services are shown on figure 3. The first one is the `DMatmul_Operator` service which performs the task $Y = A \cdot X$ on distributed matrices `DMatrix` and the other is `QRF` a QR Factorizer service which performs several tasks related to the QR factorization [2,8] of `Matrix`. An example of use of the state of the service is the call to `int QRF::rank()` method at line 27 of fig. 3, at this point the `QRF` service will check if the QR factorization is already computed and raise an error if not.

The advantages of the *Service Pattern* (compared to the solution adopted in Paladin) are that the related operations are grouped together in an object offering a complete service, and that the arguments are dispatched only

when needed. The only drawback is the unusual syntax which can be further improved by authorizing implicit connection/disconnection. For example if parameter ``A'' of `matmul` doesn't change a call to `Matmul.apply(X,Y)` would trigger a `X` and `Y` connection, a call to `matmul()` and a `X` and `Y` disconnection. The *Service Pattern* may be implemented with any object-oriented language providing dynamic type control, but we must mention that the same spirit of service is used in the Fortran 77 *Reverse Communication* mechanism [4]. It is not surprising that the main goal of Reverse Communication is to abstract away from iterative method code some matrix/vector operations.

The participants of the pattern are: a `Service` class, a base `Object` class and as many descendants of `Object` as needed. The `Object` class has no requirement other than having type information provided for it [1]. In the following `parm_name` stands for the name of the argument of the service, typically ``A'', ``X'', ``Y'' in fig. 3 which represents the role the arguments play for the `Matmul_Operator` service. The `Service` class must provide:

- one redefinable method `connect(Object* O, char* parm_name)` which finds the dynamic type of `O` and calls the specialized method corresponding to the specified `parm_name`. This method should be redefined by the descendant of the service to handle the concerned types,
- one method `connect_PARM_NAME(DType* O)` for each `parm_name` which makes sense for the service and for each dynamic type `DType` accepted for this parameter. The call to such a method will register the object into the service and update the state of the service,
- one redefinable method `disconnect(char* par_name)` which unregisters the specified parameter(s) and update the service state,
- one method `do_task()` for each computational task offered by the service. The service may have several methods of this kind. The service should check its state in order to see if it can answer to the `do_task()` calls. As an example the `QRF` service has the following computational task:
  - `void factorize()`: computes the QR factorization in compressed form,
    state condition: matrix to be factorized should be connected
  - `void get_R(Matrix*)`: retrieves the R factor,
    state condition: factorization should have been computed by `factorize()` and Q factor should not have been formed,
  - `int rank()`: returns the rank of the computed factorization,
    state condition: factorization should have been computed by `factorize()` and Q factor should not have been formed
  - `void compute_Q()`: explicitly forms Q factor,
    state condition: factorization should have been computed by `factorize()`
- a specification of the policy of the service which explains how the operations of the service should be called, ordered or not, how many parameters must

[1] In C++ this translates into having at least 1 non pure virtual method

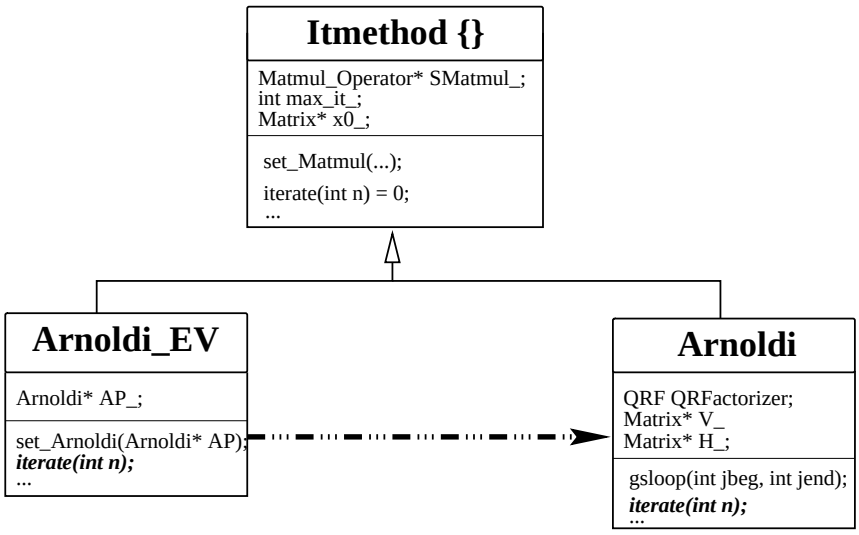

Fig. 4. Iterative methods interface

be registered for each task...

The multiple dispatch problem is solved but we still not fulfill all the conditions for a full reuse of iterative methods code. The preceding technique works for all the operations listed in §1.2 but the memory allocation. We explain the reason and the solution in the next section.

2.4 The Need of Genericity

Some classes or functions of LAKe must be able to allocate matrices whose size depends on input parameters of theses classes or functions. As an example the `Arnoldi` class shown in figure 4 must be able to allocate the block Hessenberg matrix H and the matrix of the Krylov subspace basis V. The dimensions of H and V can be deduced from two parameters needed to create an `Arnoldi` object: the Krylov subspace size m and the matrix $V_1 \in \mathbb{R}^{n \times s}$ (named `Itmethod::x0_` on Fig. 4). Algorithm 1.1 implies $\mathsf{H} \in \mathbb{R}^{(m+1)s \times ms}$ and $\mathsf{V} \in \mathbb{R}^{n \times (m+1)s}$.

In a sequential context this is not a problem since the `Matrix` used by `Arnoldi` must have a method to create any rectangular matrix. Now in a parallel context V_1 is distributed which implies that V should be distributed and H is usually not distributed (see §1.2) but the `Arnoldi` class has no way to allocate distributed matrices. In fact, it does not need to know if the matrices it uses are distributed or not. There is a simple solution to this problem: make all distributed object parameters of the `Arnoldi` class. Then H and V are now parameters for creating an `Arnoldi` object. We must add to this list the temporary variable W of algorithm 1.1. It must be distributed too because it is

involved in algebraic operations with distributed matrices. Finally to create an `Arnoldi` object, all the matrices must be passed as parameters. We have made a big step backwards since *our class has the same structure as a Fortran subroutine: it requires input/output parameters and workspace!*

Remark 5 (Classes or Functions) *It may seem a better choice to make the `Arnoldi` class a function as it is done in IML++ [5] or ITL [20]. We are convinced this is not a good choice since our goal is reusing the code implementing the Arnoldi algorithm. In fact if Arnoldi were a function every Arnoldi client would allocates the* H, V *and* W *matrices before using the function. If this client were given* V_1 *and* m *as parameters, like `Arnoldi_EV` is, he would require to be given* H, V *and* W *too in order to be able to forward them to its own Arnoldi function. In the end every related iterative method would become functions, each of them requiring several preallocated matrices arguments including workspace like* W*. Workspace is really needed since the iterative method does not know it uses distributed matrices. This approach breaks encapsulation since clients of Arnoldi should provide Arnoldi's private data and workspace. This is against reuse too since no client can polymorphically use a specialized Arnoldi function.*

Another solution is to use the *Service Pattern* or the *Abstract Factory* [7, pages 87–95] and/or *Factory Method* [7, pages 107-116] patterns to design a `Matrix_Allocator` service and pass it as parameters of `Arnoldi`, but it would make the code of `Arnoldi` uglier which is just what we want to avoid. The concept that solves all the issues is *genericity*.

Definition 6 (Genericity) *Genericity is the ability to parameterize a class with a type. We note the generic class `A<TB>` where the class `A` is parameterized by the formal generic type parameter `TB`.*

A classical example of the use of genericity is the `Container<TElem>` which defines a `Container` whose elements are of type `TElem`. To use this class with elements of type `double` we need to *instantiate* it with the actual type parameter: `Container<double>`. Conceptually, the compiler will take the code of the generic class and try to substitute the formal generic parameter with the actual one. This means that at compile time the generic instantiation creates a *new class* which represents a *new type*. This compile-time new type creation removes the contravariance problem. While writing the code of the methods of the generic class we implicitly assume that the formal generic parameter has some properties [2] like having the $+$, $-$, $*$ and / operators. We can require stronger constraints on the generic parameter since it may be a class on which we assume a given interface. If an actual type or class `AT` fulfills the constraints of the formal generic parameter `T` we say that `AT` conforms to `T`.

[2] This may be expressed by some languages like Ada as *constrained* genericity

The solution to the distributed allocation problem is to parameterize the matrix class with an opaque type `TShape` that gives informations about the *shape* of the matrix.

3 Shaped Matrix and Matrix Shape

In this section we explain the notion of *Shaped Matrix* and *Matrix Shape* and we give examples which illustrate their usefulness.

Definition 7 (Shaped Matrix) *A* Shaped Matrix *is a generic type of matrix, noted* `Matrix<TShape>` *in which the formal generic parameter* `TShape` *must conform to the specification of a* Matrix Shape. *The only necessary argument to create a* Shaped Matrix *object is a* Matrix Shape *of type* `TShape`. *A* Shaped Matrix *should be able to give the type of its* Matrix Shape *and a* Matrix Shape *object of this type representing its current shape.*

Before we go on with *Shaped Matrix* we present the specification of a *Matrix Shape*.

3.1 Matrix Shape

A *Matrix Shape* is a type which has several requirements. It furnishes the minimal set of operations and functions to make structural calculation on matrices, that is every (elementary) operation that can be done on a matrix can be done on a matrix shape. A *Matrix Shape* is a kind of reification of the Abstract Data Type matrix as specified by the elementary operations given in §1.2. The necessary operations on a *Matrix Shape* falls into 3 categories:

(1) Creator/Destructor
(2) Logical Operations
(3) Algebraic Operations

The functions of each category are shown in tables 1, 2, 3.

Now that the specification of a *Matrix Shape* is complete, we will show its use in the design of a *Shaped Matrix*.

Table 1
Creator/Destructor Operations on *Matrix Shape*

create()	$: \square \rightarrow TShape$
Default creation. This function creates an invalid shape. After this creation the validity check would answer that the shape is invalid.	
destroy(S)	$: TShape \rightarrow TShape$
Destroys a shape. This function destroys a shape, and makes it invalid. After this the validity check would answer that the shape is invalid.	
create(m,n)	$: int, int \rightarrow TShape$
Creates the default shape for a $m \times n$ matrix.	
create(S)	$: TShape \rightarrow TShape$
Creates a copy of a shape S.	
row_shape(S)	$: TShape \rightarrow TShape$
Creates a 1-row shape with the same column shape as S. The number of rows of the created shape is 1 and the number of columns is the same as S. If S represents an $m \times n$ matrix then the resulting shape would represent an $1 \times n$ matrix.	
column_shape(S)	$: TShape \rightarrow TShape$
Creates a 1-column shape with the same column shape as S. The number of columns of the created shape is 1 and the number of rows is the same as S. If S represents an $m \times n$ matrix then the resulting shape would represent an $m \times 1$ matrix.	
sub_shape(S,I1,I2)	$: TShape, Index, Index \rightarrow TShape$
Creates a shape which is the sub-shape of S ranging from index I1 to index I2. The type of the indexes I1 and I2 are unspecified here. Typically an index I should be a pair (i, j) if we are interested in contiguous sub-shape, a triplet (i, j, s) if we are interested in non contiguous (slice) sub-shape or any user defined index (for example block index).	
expand_shape(S,m,n)	$: TShape, int, int \rightarrow TShape$
Creates a shape which is the same as S with m times more rows and n times more columns. If S represents an $p \times q$ matrix the resulting shape would represent an $p \cdot m \times q \cdot n$ matrix.	

3.2 Shaped Matrix

From Def. 7 we know that a *Shaped Matrix* has a formal generic parameter `TShape` conforming to a *Matrix Shape*. The functional requirement of a *Shaped Matrix* conforming to the type `Matrix<TShape>` are:

Table 2
Logical Operations on *Matrix Shape*

!S	$: TShape \rightarrow boolean$
Invalidity checking. The function checks if the shape is valid, and return true if the shape is *invalid*. A shape may be invalid if a previous operation produced an invalid result.	
S1==S2	$: TShape, TShape \rightarrow boolean$
Strict equality. The function returns true if S1 is strictly equal to S2.	
S1_is_assignable_to_S2(S1,S2)	$: TShape, TShape \rightarrow boolean$
The function checks if we may assign a shape S1 to a shape S2 and returns true if it is possible and false otherwise. *Note that this may be the same as strict equality but this is not mandatory.* In the context of parallel shape S1 may be assignable to S2 even if S1 is not strictly equal to S2, for example if S1 is duplicated and S2 is not.	

Table 3
Algebraic Operations on *Matrix Shape*

nrow(S)	$: TShape \rightarrow int$
Number of rows of the shape. This represents the number of rows of the matrix.	
ncolumn(S)	$: TShape \rightarrow int$
Number of columns of the shape. This represents the number of rows of the matrix.	
transpose(S)	$: TShape \rightarrow TShape$
Matrix Transposition. This functions returns the shape corresponding to the transposed matrix. The resulting shape may be invalid if this kind of shape does not support transposition.	
S1+S2	$: TShape, TShape \rightarrow TShape$
Matrix Addition. This functions returns the shape corresponding to the addition of matrices whose shape are S1 and S2. The resulting shape may be invalid if the addition of the 2 matrices is not computable.	
S1*S2	$: TShape, TShape \rightarrow TShape$
Matrix Product. This functions returns the shape corresponding to the product of matrices whose shape are S1 and S2. The resulting shape may be invalid if the product of the 2 matrices is not computable.	

(1) the only necessary information to create (allocate) a `Matrix<TShape>` is an object of type `TShape`,
(2) a `Matrix<TShape>` must be able to give its current shape, i.e. an object of type `TShape`,
(3) a `Matrix<TShape>` must be able to provide the *type* of its shape, noted

```
 1 // TMatrix is the type of Shaped Matrix used by Arnoldi
 2 [...]
 3 TMatrix::shape_type    SW;
 4 SW = (SMatmul.shape()*x0.shape());
 5 #ifdef _LAKE_CHECK
 6  if (!SW)
 7    {
 8     error("Invalid Sparse Matrix Vector Multiply");
 9     error(SW.invalid_info()); // print invalid_info string of SW
10    }
11 #endif
12 // allocate W whose shape is the product of
13 // SMatmul and x0 shapes.
14 W.create((SMatmul.shape())*(x0.shape()));
15
16 TMatrix::shape_type   Sx0;
17 Sx0 = x0.shape();
18 bV.create(Sx0.expand_shape(1,max_it()+1));
19 TMatrix::shape_type   Sx0t;
20 Sx0t = Sx0;
21 Sx0t.transpose();
22 // allocate bH whose shape is the shape of
23 // x0^{T} times x0 expanded max_it()+1 times along the rows
24 // and max_it() times along the columns.
25 bH.create((Sx0t*Sx0).expand_shape(max_it()+1,max_it()));
```

Fig. 5. Using shape for allocating matrices

`Matrix<TShape>::shape_type`.

Note that even if we note a *Shaped Matrix* `Matrix<TShape>` it may be ANY user defined type which conforms to `Matrix<TShape>`.

The piece of code of the `Arnoldi` class in figure 5 illustrates the use of shape for allocating H, V and W. At line 1–14 we create W whose shape is the product of matmul operator shape and x_0 shape. At line 16–18 we define V whose shape is the shape of x_0 expanded along the columns $m+1$ times. Finally at line 19–25 we create the H block Hessenberg matrix whose shape is the shape of $x_0^T x_0$ duplicated m times along the columns and $m+1$ times along the rows. At line 3,16 and 19 we use the embedded shape *type*: `TMatrix::shape_type`. Lines 5–11 show an example of the use of the *Matrix Shape* for guard condition, the validity of the product is checked on the shape before any product occurred.

One may argue that we could have made the shape mechanism work directly on matrix object. If we had given the operator* the matrix multiplication semantic we would have had to handle temporaries generated by this operators

```
 1 int
 2 main(int argc, char* argv[])
 3 {
 4 // initialize LAKe
 5 LAKELL.Initialize(argc,argv);
 6 // retrieve the number of processors
 7 const int n_processors = Lakell::LAKE_COMM_WORLD.Size();
 8 [...]
 9 // declare a distributed matrix whose shape is a Distribution
10 Matrix<Distribution>   x0;
11 Distribution          Dx0; // a Distribution object
12 int n_rows    = 10000;
13 int n_columns = 5;
14
15 // create a row cyclic distribution for a
16 // matrix with n_rows and n_columns
17 Dx0.create_row_cyclic(n_rows,n_columns,n_processors);
18 // allocate the distributed matrix x0
19 x0.create(Dx0);
20
21 [...]
22 // pass x0 as parameter to Arnoldi ...
23 [...]
24 }
```

Fig. 6. Distributing matrices in the main program

which is something tricky [23] we were not interested in. Otherwise we should have defined algebraic operators that do not have their usual sense. Since shape objects are small objects we may accept some temporaries due to operators, moreover temporaries optimization may be added later.

The example of fig. 5 shows how a client of a *Shaped Matrix* may allocate matrices whose shape can be computed from matrices arguments passed to the client. In our example the client is the `Arnoldi` class which has been given the matrix parameter `x0` and the size of the Krylov subspace m (given by the `max_it()` method). Now allocation of the (eventually) distributed matrices in `Arnoldi` comes from the facts that (1) `x0` is a distributed matrix and (2) the algebraic rules of computation on shapes dictate how to create (eventually) distributed shape from `x0`'s shape. The shape mechanism does not prevent the user to initially distribute *himself* the parameters in the main program. In order to create a distributed matrix `x0` in the main program the user could have written something similar to the code presented in figure 6.

The methods used to create distributed shape are NOT part of the standard shape specifications. Every parallel matrix shape designer will give its own

facilities for building distributed shapes, along with the specified operations (see tables 1,2,3) on shape. In this way, only the main program would have to be changed for swapping from one parallel shaped matrix class to another. In this way matrix shape designers do not need to implements all possible distributions schemes but only the ones that are useful for their applications. The only constraint for a matrix shape class is to implement the specified standard shape operations.

3.2.1 Genericity solves Contravariance

We have just seen how the *Shaped Matrix* mechanism solves the distributed allocation problem. It may not seem obvious that genericity is really needed in this case, we may have made the *Shaped Matrix* class a client of an abstract *Matrix Shape* class and derive concrete *Matrix Shape* for sequential and parallel matrix class. But the contravariance problem would still be there in the `Arnoldi` class as shown in fig. 2. This problem is solved if we make `Matrix` a generic parameter of `Arnoldi` which becomes `Arnoldi<TMatrix>`, where `TMatrix` should be a *Shaped Matrix*. Now when compiling the `Arnoldi<TMatrix>` with an actual generic parameter `Sequential_Matrix` or `Parallel_Matrix` for `TMatrix` the compiler will instantiate the right call to the `TMatrix::tmatmul(TMatrix&, TMatrix&)` **at compile time**. Contravariance is solved by genericity for all the `Itmethod<TMatrix>` classes.

We made the *Shaped Matrix* class a generic class `Matrix<TShape>` because we want to reuse the code of the generic Matrix class such that `Sequential_Matrix` is in fact `Matrix<Sequential_Shape>` and `Parallel_Matrix` is in fact `Matrix<Parallel_Shape>`. This is just what we have done in LAKe. It must be noted that the `Parallel_Matrix` class is not equal to the `Matrix<Parallel_Shape>` class but it is derived from it. This last point is important since we need to know the exact shape of the matrix in order to implement `Matrix<TShape>::create(TShape S)` and all necessary elementary matrix operations from §1.2. Then `Parallel_Matrix` redefines all the methods inherited from `Matrix<Parallel_Shape>` that need to access the interface of `Parallel_Shape` that is not a part of the `Matrix Shape` interface.

3.3 Advantages of Shaped Matrix

We must quote several advantages of *Shaped Matrices* which make the client of a *Shaped Matrix* really independent of its implementations details.

- A client of *Shaped Matrix* may allocate temporaries without knowing how it is allocated, even in the distributed case.

- Operations on shape fix the rules for distributing the result of distributed operations on matrices.

  For example the result of the product of a column-wise distributed matrix by a row-wise distributed matrix should be a duplicated matrix. Those rules may be changed in the shape class itself and may influence the distribution of all matrices, **even the temporary storage** used by iterative methods. If the distribution rules change you won't need to change any single line of the code of the iterative method class.
- Shapes unify guard conditions for matrix operations.

  A method like `Matrix::matmul(...)` usually checks that its matrices arguments have the required dimensions to do matrix multiplication, those methods now use logical operators on the shape of the arguments. When doing $Y = A \cdot X$ we should have:
  `S1_is_assignable_to_S2(Y.shape(),A.shape()*X.shape()).`

  When adding two (potentially) distributed matrices A and B you may check that `!(A.shape()+B.shape())`. If A and B are not distributed in a compatible way, the sum of A and B shapes will be invalid.

3.4 Compile Time vs Run Time Polymorphism

Genericity may be viewed as a compile time polymorphism. Inheritance and dynamic binding are the support for run time polymorphism whereas conformance and genericity are the support for compile time polymorphism. Equivalence between genericity and inheritance has already been discussed [13,19]. A conclusion of these studies is that inheritance is a general purpose mechanism that may easily emulate genericity [3] but the converse is false. Genericity cannot emulate inheritance but genericity is a powerful mean to handle self-referencing.

We think that *both* compile time and run time polymorphism are useful and complementary. A convenient mean for us to chose between compile time and run time polymorphism is the notion of *dynamic* and *static* client relation.

Definition 8 (Dynamic/static Client) *A class `A` is a client of another class `B` if `A` uses at least one object of type `B`. The client relation is dynamic if an object of type `A` may use several class inheriting from `B` at runtime. The client relation is static if the type of the object inheriting from `B` used by `A` does not change during runtime.*

Then we recommend applying the following rule:

[3] This is the way Java supports genericity since there is no built-in support for genericity in Java. But this may change in the future [1].

Remark 9 (Genericity choice rule) *If a class* `A` *is a* static client *of class* `B` *make the class* `B` *be a generic parameter of class* `A`*. If a* `A` *is a dynamic client of* `B` *then the client relation should be implemented using run time polymorphism.*

From the implementation point of view the Service/Object architecture of the Service Pattern is made to fully support the dynamic client relation in which `B` is a service for `A`. Concerning genericity the implementation will be straightforward if the language fully supports genericity, on the contrary pre-processor or source-to-source compiler may be used to support genericity [4].

Applying the *Genericity choice rule* on the LAKe architecture of figure 1 on page 6 gives the following generic classes:

- `Matrix<TShape,TCSC>`
- `QRF<TMatrix>`
- `EV<TMatrix>`
- `Itmethod<TMatmul_Operator,TMatrix>`
- `Arnoldi<TMatmul_Operator,TMatrix>`
- `ABR_Arnoldi<TMatmul_Operator,TMatrix>`

Remark 10 (Other generic libraries) *IML++ (Iterative Method Library) [5] and MTL/ITL (Matrix Template Library/Iterative Template Library) [20] both define generic iterative methods. LAKe handles issues which are unresolved in those libraries:*

(1) they have not been used with distributed matrix classes. A parallel extension of MTL called PMTL is under development but at the time of the writing no tests have been done yet. Moreover it is not mentioned that ITL components will be able to use PMTL component without any change.
(2) the iterative methods are implemented as generic functions and not classes. This means that polymorphically reusing an Arnoldi process was not a goal of those libraries.
(3) the functions implementing iterative methods cannot handle the allocation of a distributed variable.

The generic LAKe library fulfills its requirements. The code of the iterative methods hierarchy is *strictly the same* when used with parallel or sequential matrices. Iterative methods are really building blocks which may hold and allocate their own data distributed or not. We reuse most of the code of the sequential matrix to implement the distributed one. We pointed out a methodology for choosing between run time and compile time polymorphism.

[4] This remains to be experienced since we used C++ which fully supports genericity

To go a step further with genericity we may borrow the idea behind the STL [14] also used in MTL which is using iterators on matrices in order to factorize generic algorithms like matrix addition or multiplication. This is currently under development and will enable us to have several specialized algorithm for distributed matrix multiplication. In this study we will examine the link between our design method and aspect-oriented programming [12] and generative programming [3] which are "natural" extensions to genericity.

If we look at the Fortran-side of the scientific computing world we see at least two relations. We have already pointed out the relation between the Service Pattern and reverse communication, but we may also draw a parallel between matrix shape and the distribution directive of HPF [11]. In fact the `DISTRIBUTE`, `TEMPLATE`, `ALIGN` directive of HPF are compiler instructions for allocating distributed matrices. The matrix shape plays the same role at runtime, so we may imagine a restructuring compiler that is "matrix shape" aware to do some optimization at compile time.

4 Numerical Experiments

We have implemented the LAKe library in C++ and used MPI through OOMPI [22] for the parallel classes. We used block Arnoldi method in order to find the 10 eigenvalues of largest modulus. The parameters of algorithm 1.2 were: $r = 10$, $s = 4$, $m = 15$. Iterations were stopped whenever the residual associated with the Ritz pair was less than 10^{-6}. The first matrix (CRY2500) is taken from the matrix market (CRYSTAL set of the NEP collection) and has 2500 rows and 12349 entries. The second matrix (RAEFSKY3) has 21200 rows and 1488768 entries. The matrices, stored in Compressed Sparse Column (CSC) format, were distributed block column wise on the specified number of processors. We did not pre-processed the sparse matrices with any graph partitionner since this was not a primary goal. Nevertheless we know that, this reordering should have improved both parallel efficiency and speed-up. Numerical experiments were done on the CRAY T3E of IDRIS [5] .

Speed-up are shown in figure 7. The solid line curve corresponds to theoretical speed-up and the dashed curve to measured speed-up. The speed-up corresponding to RAEFSKY3 begins with 2 processors since the code is unable to be run on one processor. For CRY2500 a number of processors $NPE = 0$

[5] Institut du Développement et des Ressources en Informatique Scientifique, CNRS, Orsay, France

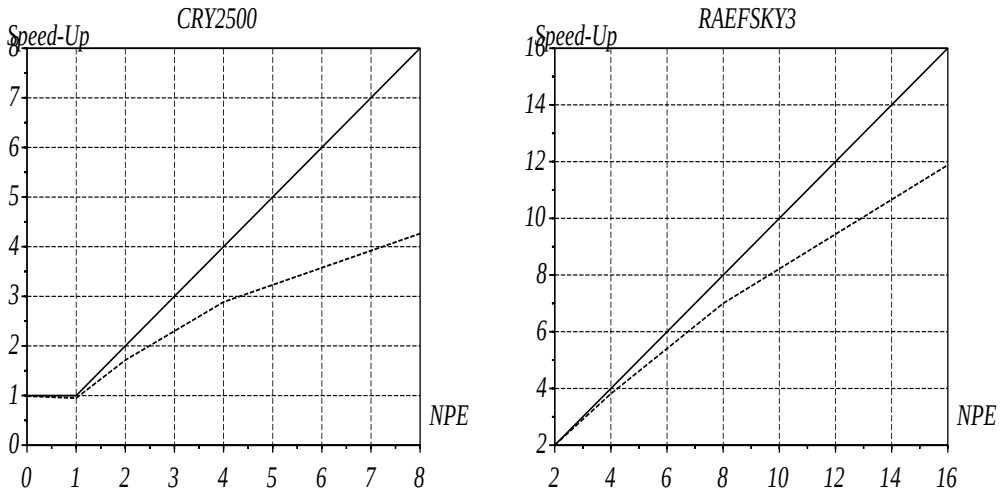

Fig. 7. Speed-up

corresponds to the sequential code and $NPE \geq 1$ corresponds to the parallel one. The speed-up are good as long as the number of processors is not too large in comparison with the size of the matrix. We note that the sequential and parallel code used for CRY2500 are derived from the *same* generic code. This means that for a data set that fits on a workstation we do not need to run the parallel version on one processor but we instantiate the sequential version. A raw comparison with a Fortran 77 code implementing the method in a *non-generic* way showed that the Fortran code was 2 times faster than the generic C++ one. We must note that the comparison is not fair since the Fortran 77 code is far from implementing every feature implemented in the C++ version.

Conclusion

We have presented how a coupled object-oriented and generic design enables the development of the *same* code for the sequential or parallel version of our linear algebra application. This is a key to parallel software maintenance and reuse. The basic idea is to parameterize the class which will become parallel by its abstract data type. We think the shaped matrix mechanism may be illustrative enough to give insight for other parallel applications. Experiments have shown that the same code is working for both sequential and parallel version with promising scalability. We pointed out that both genericity and polymorphism are useful. We think that our approach is to be related to generative programming techniques and that it is at the edge of compiling technique.

Acknowledgements

The authors want to thank Paul Feautrier and Jean-Yves Chatelier for their careful reading and improvement suggestions.

References

[1] Add generic types to the java programming language. On the Web., May 1999. http://java.sun.com/aboutJava/communityprocess/jsr/jsr_014_gener.html.

[2] E. Anderson, Z. Bai, C. Bishof, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, S. Ostrouchov, and D. Sorensen. LAPACK *User's Guide*. SIAM, 1992.

[3] Krzysztof Czarnecki. *Generative Programming: Principles and Techniques of Software Engineering Based on Automated Configuration and Fragment-Based Component Models*. PhD thesis, Dept. of Computer Science and Automation– Technical University of Ilmenau, 199x. preliminary (Oct. 1998) version kindly given by the author.

[4] Jack Dongarra, Victor Eijkhout, and Ajay Kalhan. Reverse communication interface for linear algebra templates for iteratives methods. Lapack Working Note 99, Oak Ridge National Laboratory, May 1995.

[5] Jack Dongarra, Andrew Lumsdaine, Roldan Pozo, and Karin. A. Remington. *Iterative Methods Library*, April 1996. Reference Guide.

[6] Message Passing Interface Forum. MPI: A message passing interface standard. Technical Report CS-94-230, University of Tennessee, Knoxville, TN, March 1994.

[7] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, 1995.

[8] Gene H. Golub and Charles F. Van Loan. *Matrix Computation*. The John Hopkins University Press, Baltimore London, 1989. Second Edition.

[9] Frédéric Guidec. *Un Cadre Conceptuel pour la Programmation par Objets des Architectures Parallèles Distribuées: Application à l'Algèbre Linéaire*. PhD thesis, Université de Rennes 1, Rennes, France, Juin 1995. PhD thesis edited by IRISA.

[10] Warren Harris. Contravariance for the rest of us. Technical Report HPL-90-121, Hewlett-Packard Software and Systems Laboratory, August 1990.

[11] High Performance Fortran Forum. *High Performance Fortran Language Specification*, January 1997. Version 2.

[12] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Christina Videira Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In *Proceeding of the European Conference on Object-Oriented Programming*, number 1241 in LNCS. Springer Verlag, 1997.

[13] Bertrand Meyer. *Object-Oriented Software Construction.* Prentice Hall, 2nd edition, 1997.

[14] David R. Musser and Alexander A. Stepanov. Algorithm-oriented generic libraries. *Software–Practice & Experience*, 24(7):623–642, July 1994.

[15] Eric Noulard and Nahid Emad. Object-oriented design for reusable parallel linear algebra software. In *Proceedings of EUROPAR'99*, 1999. Accepted to EUROPAR'99, 31 Aug. – 3 Sept. 99,.

[16] Yousef Saad. *Numerical Methods For Large Eigenvalue Problems.* Manchester University Press, 1991.

[17] Yousef Saad. *Iterative Methods for Sparse Linear Systems.* PWS Publishing Company, New York, 1996.

[18] Yousef Saad and Martin Schultz. GMRES : A generalized minimal residual algorithm for solving nonsymmetric linear systems. *SIAM Journal on Scientific and Statistical Computing*, pages 856–869, 1986.

[19] Ed Seidewitz. Genericity versus inheritance reconsidered: self-reference using generics. In *OOPSLA'94*, pages 153–163, 1994.

[20] Jeremy G. Siek, Andrew Lumsdaine, and Lie Quan Lee. Generic programming for high performance numerical linear algebra. In *SIAM Workshop on Interoperable OO Sci. Computing*, 1998. http://www.lsc.nd.edu/research/mtl/publications.htm.

[21] D.C. Sorensen. Implicit application of polynomial filters in a k-step Arnoldi method. *SIAM J. Matrix Anal. Appl.*, 13(1):357–385, 1992.

[22] Jeffrey M. Squyres, Brian C. McCandless, and Andrew Lumsdaine. *Object Oriented MPI (OOMPI): A C++ Class Library for MPI*, 1998. http://www.cse.nd.edu/~lsc/research/oompi.

[23] Todd Veldhuizen. Expression templates. *C++ Report*, 7:26–31, 1995.