Object Oriented Design for Reusable Parallel Linear Algebra Software

Eric Noulard[1,2] and Nahid Emad[2]

[1] Société ADULIS – 3, rue René Cassin – F-91742 Massy Cedex – France
E.Noulard@adulis.fr
[2] Laboratoire PRiSM – UVSQ – Bât. Descartes – F-78035 Versailles Cedex – France
Nahid.Emad@prism.uvsq.fr

Abstract. Maintaining and reusing parallel numerical applications is not an easy task. We propose an OO 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 show how the same code is used to implement both the sequential and the parallel version of the iterative methods implemented in 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.
Keywords: OO design, parallel code reuse, Krylov 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. Our main goal is to evaluate the object oriented design as a mean to reuse most of the sequential and parallel software components. We focus on the domain of linear algebra and particularly one the Krylov subspace methods. They solve either eigenproblems [5] or linear systems [6] and are good candidates to code reuse and parallelization.

In section 1, we present the block Arnoldi method and recall the usual way to parallelize such a method. We list the elementary neeeded 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 We show the limit of polymorphism and dynamic binding as a reuse scheme when compared to genericity. Finally section 3 presents some numerical experiments.

1 The Block Arnoldi Method

The Block Arnoldi method is a projection method that computes some eigenelements (u, λ) satisfying $Au = \lambda u$, of a large non-symmetric sparse matrix A of

P. Amestoy et al. (Eds.): Euro-Par'99, LNCS 1685, pp. 1385–1392, 1999.

size $n \times n$ where n is large. The very first step consists in reducing the projected matrix into an upper block Hessenberg matrix H_m of size $ms \times ms$ through the *Block Arnoldi Process*. For further details see [5].

1.1 Parallelizing Arnoldi Method

Our target parallel machines are distributed memory architectures. 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. We first decompose and distribute all the matrices of size n, the matrix A, the Krylov subspace basis $\mathsf{V}_m = [V_1, \cdots V_m]$ of size $n \times m \cdot s$ and possibly the temporary variable of size $m \cdot s$ we call W. The matrix H_m and all the m-sized matrices are replicated.

1.2 Necessary Elementary Operations

We list below the basic operations used by all the Krylov subspace methods including the block Arnoldi one. We have for $m, n, p \in \mathbb{N}$:

1. SAXPY: $Y = \alpha X + \beta Y$ with $Y, A, X \in \mathbb{R}^{m \times n}$ and $\alpha, \beta \in \mathbb{R}$. The matrices may be distributed.
2. 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.
3. Sparse 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 to do matrix product.
4. subranging: $Y = A(i_1 : i_2, j_1 : j_2)$, 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.
5. 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.
6. allocate, deallocate and (re-)distribute $A \in \mathbb{R}^{m \times n}$.

2 Designing a Reusable Software

The Krylov subspace methods and Block Arnoldi one 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. We first present the sequential design of LAKe, then we explain why polymorphism and dynamic binding are not sufficient to make this feature possible. 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.

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. Plain arrows stand for inheritance or the **is-a** relation [4, p. 811]. 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 client relation is *dynamic* if the relation is established at runtime and it is *static* if it may be established at compile time. Polymorphism

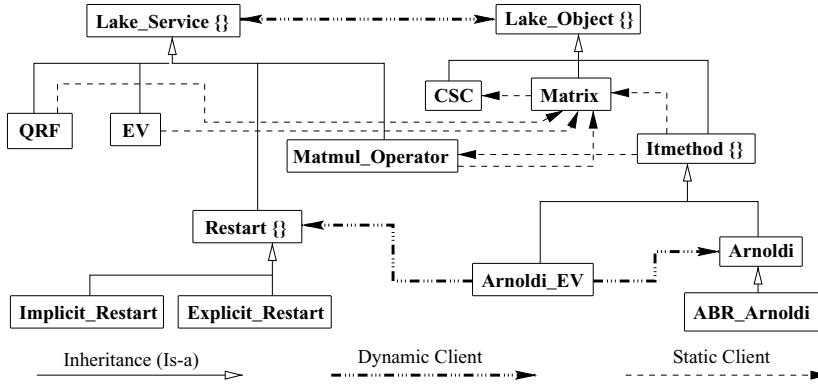

Fig. 1. The LAKe architecture

[4, p. 28] is the handling of different objects which share some parts of their interface. Polymorphism associated with dynamic binding [4, p. 29] enables the polymorphically handled object to act differently at runtime.

2.2 A Weakness of Polymorphism: Contravariance

Polymorphism seems an obvious way to parallelize iterative methods without touching the code. 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. 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. Contravariance comes out when trying to subtype functions and the correct rule for function subtyping is given in definition 2.2.

Definition 2.1 (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'.*

Definition 2.2 (Contravariance). *Let $TA \to TR$ 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: $TA' \to TR' \leq TA \to TR$ iff $TR' \leq TR$ and $TA \leq TA'$. We say that the outputs 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.*

The contravariance problem arises in the Block Arnoldi method when performing 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`. 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*)`. 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 proper method redefinition is `TB::tmatmul(TA*, TA*)`. Thus the type of the arguments must be checked dynamically in order to verify what they really are. 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 an old OO problem and has been solved in the past. It is generally not integrated in OO langages since it is costly. It was noticed and solved, *in the same linear algebra context* by F. Guidec in [3, pp. 96–99] for the Paladin linear algebra library. 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.

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* (inspired from the *Visitor Pattern* [2, p. 331]) represents a set of operations which register (or connect) their arguments one by one, a status determine which operation can be called. As an example the figure 2 shows the `DMatmul_Operator` service which performs the task $Y = A \cdot X$ on distributed matrix. The advantages of the *Service Pattern* are that the related operations

```
// register A as "Y", B as "A" and C as "X"
Matmul.connect(A,"Y");  Matmul.connect(B,"A");  Matmul.connect(C,"X");
Matmul.matmul();       // compute A = B * C
Matmul.disconnect();
```

Fig. 2. Examples of a LAKe Service

are grouped together in an object offering a complete service, and that the arguments are dispatched only when needed. 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. In the following `parm_name` stands for the name of the argument of the service, typically ``A'', ``X'', ``Y'' in fig. 2 which represent the role the arguments play for the service `Matmul_Operator`. 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`.
- 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.
- one redefinable method `disconnect(char* par_name)` which unregisters the specified parameter(s).
- one method `do_task()` for each computational task offered by the service.

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 these classes or functions. As an example the `Arnoldi` class must be able to allocate the matrices H, V and W. In a sequential context this is not a problem since the `Matrix` used by `Arnoldi` must have a method to create any rectangular matrix. In a parallel context, those matrices may be distributed and the `Arnoldi` class has no way to do this since it must not 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. 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 2.1 (Classes or Functions). It may seem a better choice to make the `Arnoldi` class a function as it is done in IML++ [1] or ITL [7]. We are convinced this is not when our goal is reusing the code implementing the Arnoldi algorithm. In fact, if Arnoldi were a function, every Arnoldi client would allocate the H, V and W matrices before using the function. In the end every related iterative method would become functions each of them requiring several preallocated matrices arguments including workspace like W. 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* 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 2.3 (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`. While writing the code of the methods of the generic class we implicitly assume that the formal generic parameter has some properties like having the $+$, and $*$ operators. If an actual type or class `AT` fullfills the constraints of the formal generic parameter `T` we say that `AT` conforms to `T`.

The solution to the distributed allocation problem is to parameterize the matrix class with an opaque type `TShape`. The formal generic parameter `TShape` encapsulates the information needed to create a matrix. The generic `Matrix<TShape>` class will have two methods: `Matrix<TShape>::create(TShape S)`, whose purpose is to create a matrix knowing its shape and `TShape Matrix<TShape>::shape()` which returns its actual shape. To have a complete solution to our problem we finally need to define a set of operations on shape: $*$, $+$, subrange, expansion... corresponding to the needed operations on matrices. Now, at compile time when we instantiate a shaped matrix we know the exact type of its shape. 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)`. Now if we want to create W whose shape is the product of matmul operator shape and x_0 shape, we write: `W.create((SMatmul.shape())*(x0.shape()))`. If we want to create V whose shape is the shape of x_0 expanded along the columns $m+1$ times we write: `V.create((x0.shape()).expand(1,max_it()+1))` Operations on shapes 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. Shapes unify guard conditions for matrix operations. When doing $Y = A \cdot X$ we should have `Y.shape() == A.shape()*X.shape()`.

For solving the contravariance problem genericity helps too. If we suppose `DMatrix` conforms to `TMatrix`, the instantiated class:
`Arnoldi_EV<Matmul_Operator<DMatrix>,DMatrix>` is a parallel iterative method which does not suffer from the contravariance problem. In fact the compiler is able to decide at *compile time* the right method it must call. The classes `Matrix` and `DMatrix` may even be unrelated (no inheritance relation) and this would work in the same way, since the conformance relation is weaker.

Remark 2.2 (Other generic libraries). IML++[1] and ITL[7] both define generic iterative methods. LAKe handles issues which are unresolved in those libraries:

1. they have not been used with distributed matrix classes.
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 fullfills 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 must note that the Service/Object hierarchy and the Service Pattern are still usefull for implementing the *dynamic client* relation. Moreover we can point out a methodology to choose between Service Pattern and generic approach: identify the dynamic and static client relation.

3 Numerical Experiments

We have implemented the LAKe library in C++ and used MPI through OOMPI [8] for the parallel classes. We used block Arnoldi method in order to find the 10 eigenvalues of largest modulus. Iterations were stopped whenever the residual associated with the ritz pair was less than $1e-6$. 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. Numerical experiments were done on the CRAY T3E of IDRIS[1].

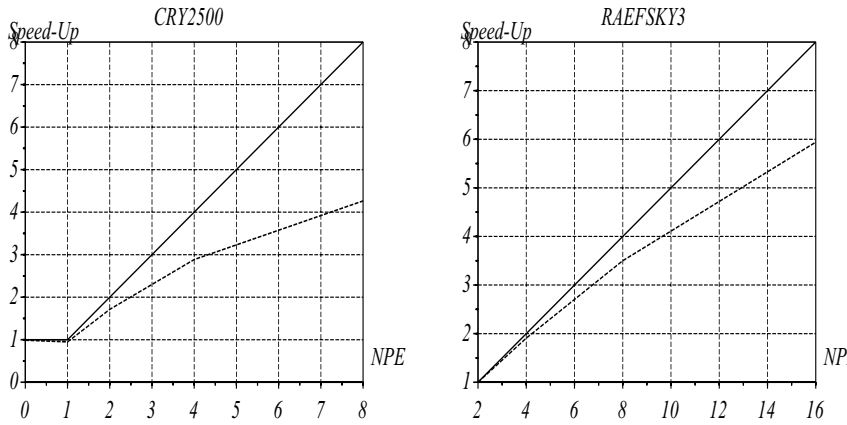

Fig. 3. Speed-up

Speed-up are shown in figure 3. The solid line curve corresponds to theoretical speed-up and the dashed curve to measured speed-up. The speed-up correspond-

[1] Institut du Developpement et des Ressources en Informatique Scientifique, CNRS, Orsay, France

ing to RAEFSKY3 begins with 2 processors since the code is unable to be run on one processor. For CRY2500 a number of processors $NPE = 0$ corresponds to the sequential code and $NPE \geq 1$ corresponds to the parallel one. The speed-up are good as soon 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 the F77 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 mecanism 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. A good perspective is a design methodology which explains how to choose between generic and polymorphic components in order to build reusable and extendable software for both sequential and parallel applications.

References

[1] Jack Dongarra, Andrew Lumsdaine, Roldan Pozo, and Karin. A. Remington. *Iterative Methods Library*, April 1996. Reference Guide.
[2] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, 1995.
[3] 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.
[4] Bertrand Meyer. *Object-Oriented Software Construction*. Prentice Hall, 1997.
[5] Yousef Saad. *Numerical Methods For Large Eigenvalue Problems*. Manchester University Press, 1991.
[6] Yousef Saad. *Iterative Methods for Sparse Linear Systems*. PWS Publishing Company, New York, 1996.
[7] 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.
[8] 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.