Design of a Parallel Object-Oriented Linear Algebra Library

Frédéric Guidec, Jean-Marc Jézéquel

Design of a Parallel Object-Oriented Linear Algebra Library

F. Guidec and J.-M. Jézéquel

I.R.I.S.A. Campus de Beaulieu
F-35042 Rennes Cedex, FRANCE
Tel: +33-99.84.71.91 — Fax: +33-99.38.38.32
E-mail: guidec@irisa.fr, jezequel@irisa.fr

Abstract

Scientific programmers are eager to exploit the computational power offered by Distributed Memory Parallel Computers (DMPCs), but are generally reluctant to undertake the manual porting of their application programs onto such machines. A possible solution to this problem consists in using libraries that hide the details of parallel computations. We show how to go one step beyond, using the full power of modern object-oriented languages to build generic, extensible object-oriented libraries that permit an efficient and transparent use of DMPCs. In EPEE, our Eiffel Parallel Execution Environment, we propose to use a kind of parallelism known as data-parallelism, encapsulated within classes of the Eiffel sequential object-oriented language, using the SPMD (Single Program Multiple Data) programming model. We describe our method for designing with this environment a truly object-oriented linear algebra library for DMPCs. In the conclusion, we enumerate the advantages of our approach and make a few prospective remarks.

Keywords: Programming Environments for Massively Parallel Architectures, Data-Parallelism and SPMD model, Object-Oriented Libraries

1 Introduction

Although scientific programmers long for the computational power of Distributed Memory Parallel Computers (DMPCs), they are generally reluctant to cope with the manual porting of their applications onto such machines. The tedious tasks of parallelization, distribution, process handling and communication management are not among the most attractive features of DMPCs.

It is our conviction that object-oriented techniques should be used to ease the programming of DMPCs. In object-oriented programming, basic entities are *objects* and *classes*. Major design mechanisms are encapsulation (for information hiding) and inheritance (for code sharing and reuse). A class can be roughly defined as a set of objects that share a common structure and a common behavior. An object is simply an instance of a class.

The idea that object-oriented programming is a key for handling DMPCs has already been explored with libraries Lapack.h++ and ScaLAPACK. These libraries are built after time honoured libraries such as LINPACK and EISPACK, which constitute a kind of a standard in the FORTRAN community. LAPACK [Aa90] is a modern compilation of both LINPACK and EISPACK. Lapack.h++ [Ver93] is an encapsulation of LAPACK to be used with C++. ScaLAPACK [Da93] is a distributed memory version of the LAPACK software package, devoted to dense and banded matrix problems. Key design features are the use of distributed versions of the Level 3 BLAS routines as algorithmic building blocks, and an object-based interface to the library. Although C++ can be used as an object-oriented language, both Lapack.h++ and ScaLAPACK scarcely use object-oriented concepts. They exhibit a very shallow class hierarchy, almost no multiple inheritance and dynamic binding of methods is most of the time impossible, due to the fact that static binding is the default in C++. One can hardly use subclassing to specialize an existing class of the library. It is thus very difficult for an application programmer to integrate new features in the library. In this respect Lapack.h++ and ScaLAPACK remain as closed as are the old FORTRAN libraries. Furthermore genericity is seldom used: for instantiating symmetric matrices, for example, one can find such classes as *FloatSymMat, DoubleSymMat, FComplexSymMat, DComplexSymMat* instead of a single generic class *SymMat[T]*.

In this paper, we show how to go one step beyond, using the full power of modern object-oriented languages to build generic and extensible —that is, open— object-oriented libraries, while permitting an efficient and transparent use of DMPCs. We propose to use a kind of parallelism known as data-parallelism, encapsulated within classes of a purely sequential object-oriented language, using the SPMD (Single Program Multiple Data) programming model. In section 2 we present EPEE, our Eiffel Parallel Execution Environment. Section 3 describes the organization of our object-oriented linear algebra library. The implementation of local and distributed matrices and vectors is discussed in section 4. Section 5 deals with interoperability issues. In section 6 we show how the parallelization of the library can be performed incrementally. In the conclusion, we enumerate the advantages of our approach and make a few prospective remarks.

2 EPEE: an Eiffel Parallel Execution Environment

Opposite to object-oriented languages such as POOL-T [Ame87] and ABCL/1 [YBS86], where objects can be made active and invocations of methods result in actual message passing, we focus on a data-parallelism model associated with a SPMD (Single Program Multiple Data) mode of execution. Our goal is to completely hide the parallelism to the user, that is, the application programmer.

The SPMD mode of execution appears as an attractive one because it offers the conceptual simplicity of the sequential instruction flow, while exploiting the fact that most of the problems submitted to DMPCs involve large amounts of data in order to generate useful parallelism. Each process executes the same program, corresponding to the initial user-defined sequential program, on its own data partition. The application programmer still views his program as purely sequential and the parallelism is derived from the data

decomposition, leading to a regular and scalable parallelism.

We implemented these ideas in EPEE (Eiffel Parallel Execution Environment [Jez92]): data distribution and parallelism are totally embedded in traditional sequential classes, using nothing but already existing syntactical constructions. EPEE is based on Eiffel [Mey88] because this language features strong encapsulation with static type checking, multiple inheritance, dynamic binding and genericity.

We distinguish two levels of programming in EPEE: the user (or *client*) level and the designer (or *provider*) level. Our aim is that at client level, nothing but performance improvements appear when running an application program on a DMPC. Moreover, we would like these performance improvements to be proportional to the number of processors available in the DMPC. This would result in a linear speedup and guarantee the scalability of the application.

The designer of a parallelized class is responsible for implementing general data distribution patterns and code parallelization rules, thus ensuring its portability, efficiency and scalability, while preserving a "sequential-like" interface for the user. The designer selects a candidate to data parallelization. Interesting classes are those that describe data structures that aggregate large amounts of data, such as classes based on arrays, sets, trees, lists... For each class, one or several distribution policies are to be chosen and data access methods —also called *accessors*— redefined accordingly, using the abstractions and facilities provided by the EPEE library. At first, a simple implementation based on the single SPMD model is realized, reusing sequential classes in a parallel context. Methods are then optimized —*i.e.* parallelized— as and where needed. This progression permits an incremental porting of already existing applications from sequential target machines to DMPCs. Parallelization can be performed either manually or automatically, using the reusable parallel software components which are a part of the EPEE library: common data distribution and computation schemes are factorized out and encapsulated in abstract parallel classes, which are to be reused by means of multiple inheritance. Reusable parallel components include parameterized general purpose algorithms for traversing and performing actions on generic data structures. They have been described in [Jez93b], along with a prototype implementation and application examples on various data types.

3 Design of "Abstract" Specifications for Matrices and Vectors

Our library is built around the specifications of the basic entities of linear algebra: matrices and vectors. A fundamental principle applied when designing this library is that the abstract specification of an entity is dissociated from any kind of implementation detail. Although all matrices, for example, share a common abstract specification, they do not necessarily require the same implementation layout. Obviously dense and sparse matrices deserve different internal representations. The same remark goes for vector entities.

The abstract specifications of matrices and vectors are encapsulated in classes MATRIX and VECTOR. Both classes are generic and can thus be used to instantiate integer matrices and vectors, real matrices and vectors, complex matrices and vectors, etc.

Classes MATRIX and VECTOR are deferred classes; they provide no details about the way matrices and vectors shall be represented in memory. The specification of their internal representation is thus left to descendant classes. This does not imply that all features are kept deferred (a *deferred* method in Eiffel is equivalent to a pure virtual function in C++). Representation-dependent features are simply declared, whereas other features are defined —*i.e.* implemented— directly in MATRIX and VECTOR.

Class MATRIX enumerates the attributes and methods for handling a matrix entity, together with their formal properties expressed as assertions (preconditions and postconditions, etc.). A matrix is featured by the type of its generic parameter and by its size. Methods are divided in two categories: accessors and operators.

Accessors are procedures and functions used for accessing the matrix in read or write mode. We provide routines for accessing a matrix at different levels. Basic routines *put* and *item* give access to an item of the matrix. Higher level accessors allow the user to handle a row, a column or a diagonal of the matrix as a vector entity, and a rectangular section of the matrix as a plain matrix. An important feature about accessors is that most of the time they imply no copy of data. They simply provide a "view" of a section of a matrix. Thus modifying this view is equivalent to modifying the corresponding section. Views necessitate special implementations, which are encapsulated in classes ROW, COLUMN, DIAGONAL, SUBMATRIX, and SUBVECTOR (figure 1).

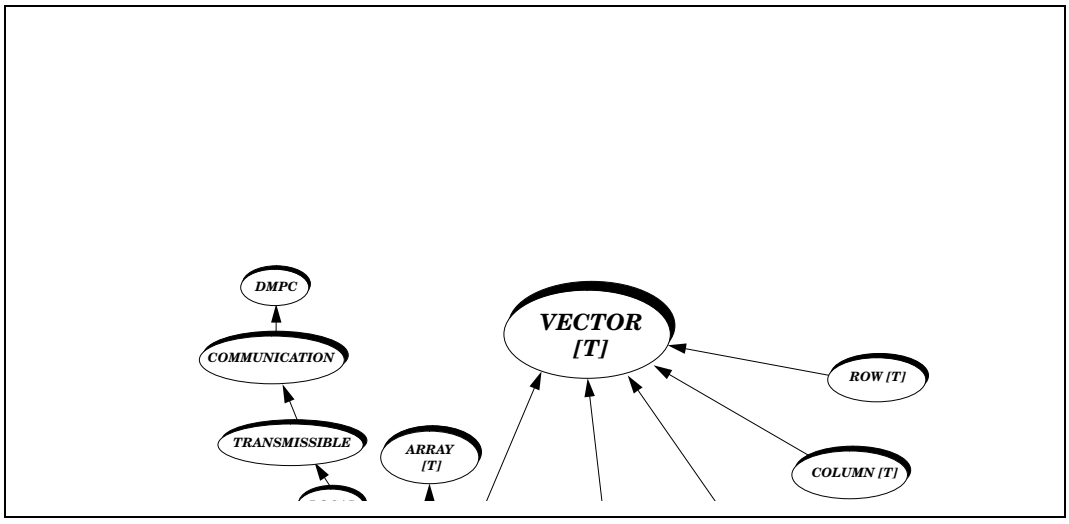

Figure 1: Class hierarchy for vector entities

The set of multi-level accessors actually provides the same abstractions as the syntactic short-cuts frequently used in books dealing with linear algebra, such as [GV91]. Assuming that A is a $n \times m$ matrix, the expression *A.submatrix (i, j, k, l)* is equivalent to the notation $A(i : j, k : l)$. Likewise, *A.row(i)* and *A.column(j)* are equivalent to $A(i, :)$ and $A(:, j)$ respectively.

Operators are high level routines used for performing computations implying a matrix as a whole and possibly other arguments (*i.e.* other matrices or vectors). Typical operators

include routines that perform scalar-matrix, vector-matrix and matrix-matrix operations. The class also contains more complicated routines for performing such computations as the Cholesky, LDL^T and LU factorizations, for solving triangular systems, etc. Since we provide accessors at different levels (item, vector, submatrix), defining new operators is not a difficult task. Any algorithm presented in a book can be readily reproduced in the library.

The organization of class VECTOR is quite similar to that of MATRIX. In addition to the basic features (attribute *length*, accessors *put* and *item*, etc.), this class contains routines that perform scalar-vector, vector-vector (*saxpy*) and matrix-vector (*gaxpy*) operations.

The set of linear algebra algorithms available in our library is thus distributed among classes MATRIX and VECTOR. This distribution is a consequence of the fact that, like many other object-oriented languages, Eiffel is a *single-dispatching language* [Cha92]. The first argument of a method is implicitly the *current* object, whose type determines which implementation of the method is to be invoked.

The reader familiar with object-oriented programming might suggest that instead of being distributed among classes MATRIX and VECTOR, algorithms may as well be encapsulated in a third class that would constitute a collection of routines for linear algebra computation. This approach would clearly favor the separation of algorithms from the descriptions of matrices and vectors. But it would also make it more difficult to parallelize the operators of the library, since our parallelization technique (see section 6) depends on the availability of a single-dispatching mechanism in Eiffel.

4 Towards Multiple Concrete Descendants

Details relative to the internal representation of matrix and vector objects are encapsulated in classes that descend from MATRIX and VECTOR. To date our demonstrator library only permits the creation of local and distributed dense matrices. However, it may easily be augmented with new classes describing other kinds of entities such as sparse matrices and vectors, or symmetric, lower triangular and upper triangular matrices, etc. Adding new representation variants for matrices and vectors simply comes down to adding new classes in the library. Moreover, each new class is not built from scratch, but inherits from already existing classes. For example, a distributed symmetric matrix may combine features inherited from such classes as MATRIX, SYMMETRIC and DISTRIBUTED (see figure 2). For the designer of the library, providing a new representation variant for a matrix or a vector mostly consists in assembling existing classes to produce a new one. Most of the time, this process does not imply any development of new code.

A local matrix is stored in memory as a bi-dimensional array. Class LOCAL_MATRIX (see figure 2) simply combines the specification inherited from MATRIX together with the storage facilities provided by class ARRAY2, one of the numerous standard classes of the Eiffel library. This is a typical example of the so-called "marriage of convenience" [Mey88]: the abstract specification of a matrix is combined by multiple inheritance with the imple-

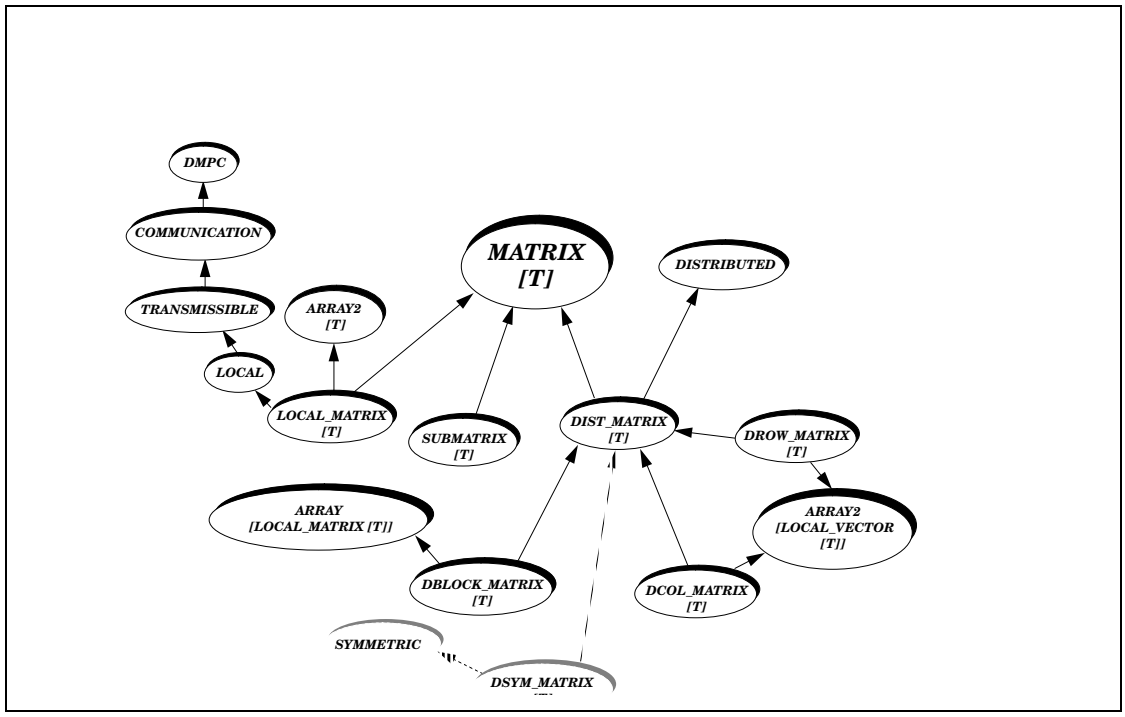

Figure 2: Class hierarchy for matrix entities

mentation facilities offered by one or several classes taken from a library. The implementation of local vectors is based on the same method: class LOCAL_VECTOR combines the features of both classes VECTOR and ARRAY.

The specifications of distributed matrices are encapsulated in descendants of class DIST_MATRIX. This class provides the designer with routines for handling the distribution of a matrix, that is, for partitioning the matrix and mapping the partitions over the processors of the target machine.

Routines for transmitting data over the network of processors are taken from the toolbox of EPEE. Class *Communication* (see figure 1), which is part of this toolbox, provides routines for performing data transmissions in one-to-one or one-to-many mode. Class DMPC provides informations about the target machine (number of processors, identity of local processor, etc.).

Our approach to the distribution of matrices is quite similar to that of HPF [For93]. The main difference is that HPF is based on weird extensions of the FORTRAN 90 syntax (distribution, alignment and mapping directives) whereas our approach only uses normal constructions of the Eiffel language. Matrices are decomposed into blocks, which are mapped over the processors of the target DMPC. The parameterization of the distribution is inspired from the syntax used in the HPF directive DISTRIBUTE, but it has been simplified in order to fit our particular needs. In spite of these simplifications we consider that, as far as the distribution is concerned, the expressiveness permitted by our

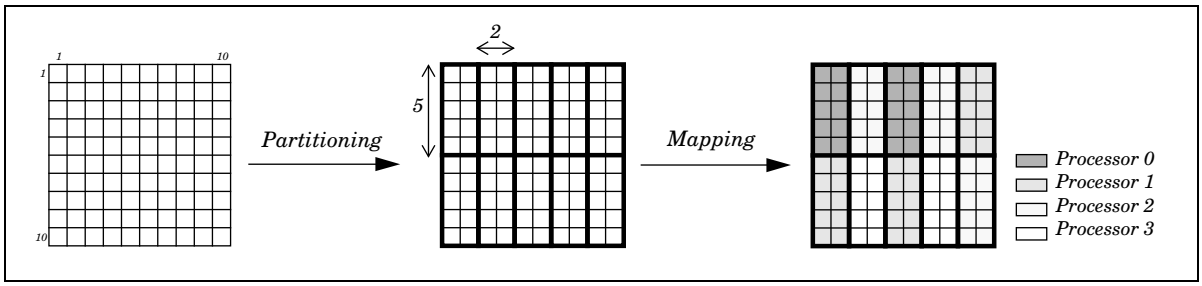

Figure 3: Partitioning of a 10×10 matrix into 5×2 blocks and mapping over 4 processors

approach is slightly equivalent to that of HPF. The user must describe at creation time the distribution pattern of a distributed matrix. He simply needs to specify the size of the matrix, the size of blocks, and whether the interleaving of blocks over the processors must be performed in a row-wise or column-wise fashion. Since all blocks have the same size, the decomposition is regular. Figure 3 shows the decomposition of a 10×10 real matrix M into 5×2 blocks, which are then mapped over a set of 4 processors in a column-wise fashion.

Class DBLOCK_MATRIX describes the implementation of a matrix distributed by blocks as a bi-dimensional table of local matrices. Each entry in this table is a local matrix. A void entry means that the local processor is not the owner of the specified block. Accessors are defined so as to take into account the indirection due to the table. When dealing with a matrix distributed by blocks, accessing block matrices is more efficient than accessing items, rows or columns of the matrix. This is an interesting consequence of providing multi-level accessors: it helps parallelizing algorithms. The idea is not new: LAPACK enhances the performances of LINPACK and EISPACK by restructuring algorithms to perform block matrix operations in their inner loops, using BLAS 3 primitives wherever possible. In LAPACK, block operations are optimized to exploit the memory architecture of shared memory machines in a machine-dependent way. ScaLAPACK uses the same method, but this time block matrix operations aim at reducing the cost of transmissions between the processors of a DMPC.

For the sake of efficiency, we also designed two classes DROW_MATRIX and DCOL_MATRIX that encapsulate alternative implementations for row-wise and column-wise distributed matrices. These distributions actually come down to particular block distribution patterns, where either the height or the width of blocks is equal to the height or width of the distributed matrix. However, when a user asks for a matrix to be distributed by rows, he actually asks for operations that imply row vectors of the matrix to be performed efficiently. Consequently, in class DROW_MATRIX, access to row vectors is favored as much as possible: a row-wise distributed matrix is implemented as a table of local vectors. Likewise, a column-wise distributed matrix is also implemented as a table of local vectors. The internal representations of row-wise and column-wise distributed matrices allow algorithms based on vectorial accessors to run faster.

The library may easily be augmented with new classes corresponding to more "exotic" distribution patterns, such as diagonal-wise distributions, or even distributions with irregular patterns (blocks of various sizes or various shapes, etc.).

5 Interoperability of Representation Variants

One of the major advantages of our library is that we ensure the interoperability of all matrices and vectors. As far as the user is concerned, a distributed matrix can be handled exactly like any other matrix. The only difference is that all matrices are not created the same way. However, after a matrix has been created, its use does not depend on whether it is distributed or local, sparse or dense, symmetric or triangular.

A method such as procedure *cholesky*, which performs a Cholesky factorization, can operate on any matrix that satisfies its preconditions: the matrix must be square symmetric definite positive. This method therefore operates on a local matrix as well as on a distributed one. In the library, an parallelized version of the Cholesky algorithm is actually provided for distributed matrices, but this parallelization remains absolutely transparent for the user who keeps calling the method the same way.

Interoperability also goes for algorithms that require several arguments. For example class MATRIX provides an infix operator that computes the sum of two matrices A and B and returns the resulting matrix R. The user may write an expression such as $R := A + B$ while matrix R is duplicated on all processors, A is distributed by rows and B is distributed by columns. Interoperability ensures that all internal representations can be combined transparently.

Complementary to the interoperability of representation variants, a conversion mechanism is available for adapting the representation of a matrix or vector to the requirements of the computation. This mechanism also plays the role of a redistribution facility. A local matrix, for example, can thus be converted dynamically into a distributed matrix, if this new representation is likely to lead to better performances. The conversion mechanism has been presented in a paper submitted to the ICSE'94 conference [Gui94].

6 Incremental Parallelization of the Library

By redefining accessors in class DIST_MATRIX and its descendants so that they allow matrices to be distributed, we simply preserve the global semantics of matrix objects. However, these redefinitions do not provide any concurrency. Executions are still sequential, since all processors execute the very same sequence of operations on the same data. Concurrency is introduced in some of the operators of the library by redefining these operators, so that during the computation each processor only deals with a part of the data. The technique consists in defining several variants of an operator and automatically selecting the most appropriate variant at runtime.

Consider the example of the matrix multiply procedure, that performs the computation $R = A \times B$. A "default" implementation of this procedure is provided in class MATRIX, inspired from the *gaxpy* based matrix multiply algorithm taken from [GV91].

This procedure operates on all kinds of matrices, but does not take into account the possible distribution of its arguments. A highly optimized version of this procedure can be provided that performs concurrently when, say, B and R are of type DCOL_MATRIX while A is of type LOCAL_MATRIX. In this algorithm, conditional statements are used for restricting the iteration domain: each processor only computes its part of the resulting matrix. Moreover, data transfers are reduced to a minimum.

Since we do not want the application programmer to be responsible for the choice of the most appropriate variant whenever he invokes a method of the library, we need a mechanism that performs this choice automatically, based on the dynamic types of the arguments. For methods with no arguments —*i.e.* methods such as *cholesky*, whose unique implicit argument is the current object— the single-dispatching of Eiffel is sufficient. As for methods with several arguments, such as the matrix multiply routine, we need a mechanism that dispatches on all the arguments. Unfortunately, unlike multiple-dispatching languages CLOS [Gab91] and Cecil [Cha92], Eiffel does not permit the implementation of multi-methods. The library is thus parallelized using an approximation of multi-methods: a set of functions is available for testing the dynamic type of an object, based on a reverse assignment attempt [Mey88]. With such functions, multiple-dispatching can be performed *manually* by the designer of the library. This mechanism works quite satisfactorily, but we must nevertheless advocate for an extension of Eiffel with a real multiple-dispatching mechanism. This would make things much easier for parallelizing the library, especially if numerous optimized versions of each operator are to be provided.

Fortunately, the parallelization of the library does not require for all accessors and operators to be parallelized at once. In a first step, it is sufficient to select and parallelize "basic" methods considered as critical for the performance of the library. Most of these methods correspond to the routines offered in the BLAS kernel (scalar-vector, vector-vector and matrix-vector operations). Actually, an interesting approach to provide the library with efficient basic methods would be to encapsulate the BLAS kernel in the library. Eiffel allows external routines to be called from any class, and we could easily modify the internal representations of local matrices and vectors so that they conform to the FORTRAN (or C) representations of arrays. It is thus possible to combine the advantages of object-oriented programming together with the performances of a highly optimized machine-dependent kernel such as BLAS, in order to build an efficient, extensible object-oriented library for massively parallel architectures.

Once the most critical methods of the library have been parallelized and optimized, the remaining of the parallelization can carry on incrementally, with the redefinition of higher level algorithms as and when they come to be needed in application programs.

7 Conclusion

EPEE aims at easing the programming of DMPCs, using powerful object-oriented techniques. Encapsulation allows data distribution and code parallelization to be hidden within classes whose accessors are redefined according to the distribution pattern, without any alteration of the class interface. An interesting point is that EPEE makes it possible to

reuse already existing Eiffel sequential classes in a parallel context. It therefore permits an incremental porting of already existing sequential applications onto DMPCs: the designer selects interesting classes to be data-parallelized, and provides these classes with a distributed implementation. Operators are then parallelized incrementally as and when needed.

In EPEE, the design of a parallelized class still requires that the designer be an expert in the application domain as well as in parallelization techniques. However, since the techniques described in this paper exhibit a rather general and systematic pattern, some kind of automation can be foreseen in the form of an interactive tool to assist the designer in the process of designing parallelized classes. This automation would be an important step towards the design of libraries providing efficient and reusable software components for DMPCs, using the widely acknowledged benefits of the object-oriented technology.

Our library is in the process of being updated from Eiffel 2 to Eiffel 3. Consequently we cannot exhibit benchmarks of the tests performed with Eiffel 3 yet. Interested readers can nevertheless refer to [GJ93] that presents some of the results we obtained with Eiffel 2.

In this paper we have described our method for designing a parallel linear algebra library, whose performances are likely to be as efficient as those obtained from a hand-written FORTRAN source code. We focused on the distribution of matrices over a DMPC. We shall soon use the same approach to distribute multi-dimensional arrays, thus providing an object-oriented library exhibiting the same features as HPF.

References

[Aa90] E. Anderson and al. LAPACK: a portable linear algebra library for high-performance computers. In *Proceedings of Supercomputing'90*, 1990.

[Ame87] P. America. Pool-T: a parallel object-oriented programming. In A. Yonezawa, editor, *Object-Oriented Concurrent Programming*, pages 199–220, The MIT Press, 1987.

[APT90] Françoise André, Jean-Louis Pazat, and Henry Thomas. Pandore: a system to manage data distribution. In *ACM International Conference on Supercomputing*, June 11-15 1990.

[Cha92] C. Chambers. Object-oriented multi-methods in Cecil. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP'92)*, 1992.

[Da93] J. Dongarra and al. An object-oriented design for high-performance linear algebra on distributed memory architectures. In *Proceedings of the Object-Oriented Numerics Conference (OON-SKI'93)*, 1993.

[For93] High Performance Fortran Forum. *High Performance Fortran Language Specification.* Technical Report Version 1.0, Rice University, May 1993.

[Gab91] R. et al. Gabriel. CLOS: integrating object-oriented and functional programming. *Communications of the ACM*, 34(9), 1991.

[GJ93] F. Guidec and J.-M. Jézéquel. Numeric parallel programming with sequential object oriented languages. In *Proceedings of the First Annual Object-Oriented Numerics Conference (OONSKI'93)*, pages 55–69, April 1993.

[Gui94] F. Guidec. Towards an approximation of polymorphic objects in Eiffel. Submitted to the 16th International Conference on Software Engineering (ICSE'94), Sorrento, Italy, May 1994.

[GV91] G.H. Golub and C.F. Van Loan. *Matrix Computations*. The Johns Hopkins University Press, 1991.

[JAB92] J.-M. Jézéquel, F. André, and F. Bergheul. A parallel execution environment for a sequential object oriented language. In *ICS'92 proceedings*, ACM, July 1992.

[Jez92] J.-M. Jézéquel. EPEE: an Eiffel environment to program distributed memory parallel computers. In *ECOOP'92 proceedings*, Lecture Notes in Computer Science, Springer Verlag, (also to appear in the Journal of Object Oriented Programming, 1993), July 1992.

[Jez93a] J.-M. Jézéquel. EPEE: an Eiffel environment to program distributed memory parallel computers. *Journal of Object Oriented Programming*, 6(2):48–54, May 1993.

[Jez93b] J.-M. Jézéquel. Transparent parallelisation through reuse: between a compiler and a library approach. In O. M. Nierstrasz (Ed.), editor, *ECOOP'93 proceedings*, pages 384–405, Lecture Notes in Computer Science, Springer Verlag, July 1993.

[Mey88] Bertrand Meyer. *Object-Oriented Software Construction*. Prentice-Hall, 1988.

[PB90] C. Pancake and D. Bergmark. Do parallel languages respond to the needs of scientific programmers? *IEEE COMPUTER*, 13–23, December 1990.

[Val90] Leslie G. Valiant. A bridging model for parallel computation. *CACM*, 33(8), Aug 1990.

[Ver93] A. Vermeulen. Eigenvalues in Lapack.h++. In *Proceedings of the Object-Oriented Numerics Conference (OON-SKI'93)*, 1993.

[YBS86] Akinori Yonezawa, Jean-Pierre Briot, and Etsuya Shibayama. Object-oriented concurrent programming in ABCL/1. In *OOPSLA'86 Proceedings*, September 1986.