Polymorphic Matrices in Paladin

F. Guidec and J.-M. Jézéquel

I.R.I.S.A. Campus de Beaulieu
F-35042 Rennes Cedex, FRANCE
Tel: +33–99.84.71.92 — Fax: +33–99.84.71.71
E-mail: jezequel@irisa.fr

Abstract. Scientific programmers are eager to take advantage of the computational power offered by Distributed Computing Systems (DCSs), but are generally reluctant to undertake the porting of their application programs onto such machines. The DCS commercially available today are indeed widely believed to be difficult to use, which should not be a surprise since they are traditionally programmed with software tools dating back to the days of punch cards and paper tape. We claim that provided modern object oriented technologies are used, these computers can be programmed easily and efficiently. 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 Paladin, an object-oriented linear algebra library for DCSs. We show how dynamic binding and polymorphism can be used to solve the problems set by the dynamic aspects of the distribution of linear algebra objects such as matrices and vectors.

1 Introduction

Distributed computing systems (DCSs)—also called distributed memory parallel computers or *multiprocessors*—consist of hundreds or thousands of processors and are now commercially available. An example of this kind of DCS is the Intel Paragon supercomputer, a distributed-memory multicomputer with architecture that can accommodate more than a thousand heterogeneous nodes connected in a two-dimensional rectangular mesh (see Figure 1). Its computation nodes are based on Intel i860 processors, and communicate by passing messages over a high-speed internal interconnect network. These kinds of multiprocessors provide orders of magnitude more raw power than traditional supercomputers at lower costs. They enable the development of previously infeasible applications (called *grand challenges*) in various scientific domains, such as materials science (for the aerospace and automobile industries), molecular biology, high-energy physics (quantic chromo-dynamic), and global climate modeling.

Although the physical world they model is inherently parallel, scientific programmers used to rely on sequential techniques and algorithms to solve their problems, because these algorithms *e.g.,* the N-body problem) often present a better computational complexity than possible direct solutions. Their interest in concurrency only results from their desire to improve the performance of sequential algorithms applied to large-scale numerical computations [11]. Scientific programmers are generally reluctant

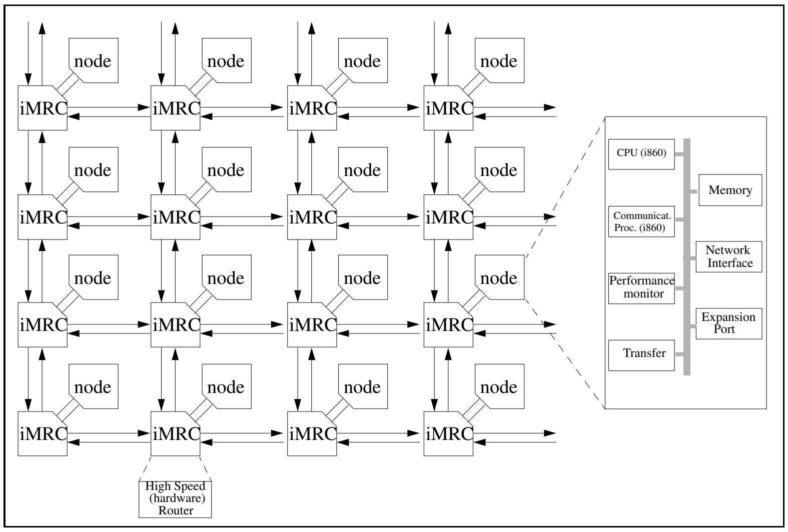

Fig. 1. The architecture of the Intel Paragon XP/S supercomputer

to cope with the manual porting of their applications on DCSs, because the average user will not move from an environment in which programming is relatively easy to one in which it is relatively hard unless the performance gains are truly remarkable and unachievable by any other method. They soon discovered how tedious it was to write parallel programs in a dialect that made the user responsible for creating and managing distribution and parallel computations and for explicit communication between the processors.

In this paper, we show how a sequential object oriented language such as Eiffel (featuring strong encapsulation, static type checking, multiple inheritance, dynamic binding and genericity) can be used to override these drawbacks. The idea is to build easy-to-use parallel object-oriented libraries permitting an efficient and transparent use of DCSs. We use the EPEE framework [10] to encapsulate the tricky parallel codes in object-oriented software components that can be reused, combined and customized in confidence by application programmers. Section 2 describes the principles underlining our method for designing an object-oriented linear algebra library for DCSs. We illustrate our approach with the example of PALADIN, an object-oriented library devoted to linear algebra computation on DCSs, whose design and implementation is outlined in Section 3. We then investigate the various aspects of dealing with multiple representations of linear algebra objects (Section 4). In the conclusion, we enumerate the advantages of our approach and make a few prospective remarks.

2 Encapsulating Parallelism and Distribution

2.1 A Simple Parallel Programming Model

The kind of parallelism we consider is inspired from Valiant's Block Synchronous Parallel (BSP) model [12]. A computation that fits the BSP model can be seen as a succession of parallel phases separated by synchronizations and sequential phases.

In EPEE, Valiant's model is implemented based on the Single Program Multiple Data (SPMD) programming model. Each process executes the same program, which corresponds to the initial user-defined sequential program. The SPMD model preserves the conceptual simplicity of the sequential instruction flow: a user can write an application program as a purely sequential one. At runtime, though, the distribution of data leads to a parallel execution.

When data parallelism is involved, only a subset of the data is considered on each processor: its own data partition. On the other hand, when control parallelism is involved, each processor runs a subset of the original execution flow (typically some parts of the iteration domain). In both cases, the user still views his program as a sequential one and the parallelism is derived from the data representation. Although EPEE allows the encapsulation of both kinds of parallelism in Eiffel classes, we mainly focused on the encapsulation of data parallelism so far. Yet, some work is now in progress to incorporate control parallelism in EPEE as well [8].

Our method for encapsulating parallelism within a class can be compared with the encapsulation of tricky pointer manipulations within a linked list class that provides the user with the abstraction of a list without any visible pointer handling. Opposite to concurrent OO languages along the line of POOL-T [2] or ABCL/1 [13], which were designed to tackle problems with explicit parallelism, our goal is to completely hide the parallelism to the application programmer.

A major consequence of this approach is that there exists two levels of programming with EPEE: the class user (or *client*) level and the class designer level. The aim is that, at client level, nothing but performance improvements appear when running an application program on a parallel computer. For a user of a library designed with EPEE, it must be possible to handle distributed objects just like local —*i.e.* non-distributed— ones.

The problem is thus for the designer of the library to implement distributed objects using the general data distribution and/or parallelization rules presented in this paper. While implementing these objects, the designer must notably ensure their portability and efficiency, and preserve a "sequential-like" interface for the sake of the user to whom distribution and parallelization issues must be masked.

2.2 Polymorphic Aggregates

The SPMD model is mostly appropriate for solving problems that are data-oriented and involve large amounts of data. This model thus fits well application domains that deal with large, homogeneous data structures. Such data structures are referred to as *aggregates* in the remaining of this paper. Typical aggregates are lists, sets, trees, graphs, arrays, matrices, vectors, etc.

A computation can be efficiently parallelized only if the cost of synchronization, communications and other processing paid for managing parallelism is compensated by the performance improvement brought by the parallelization.

Most aggregates admit several alternative representation layouts and must thus be considered as *polymorphic* entities, that is, objects that assume different forms and whose form can change dynamically. Consider the example of matrix aggregates. Although all matrices can share a common abstract specification, they do not necessarily require the same implementation layout. Obviously dense and sparse matrices deserve different internal representations. A dense matrix may be implemented quite simply as a bi-dimensional array, whereas a sparse matrix requires a smarter internal representation, based for example on lists or trees. Moreover, the choice of the most appropriate internal representation for a sparse matrix may depend on whether the sparsity of this matrix is likely to change during its lifetime. This choice may also be guided by considerations on the way the matrix is to be accessed (*e.g* regular vs irregular, non-predictable access), or by considerations on whether memory space or access time should be primarily saved.

The problem of choosing the most appropriate representation format of a matrix is even more crucial in the context of distributed computation, since matrix aggregates can be partitioned and distributed on multi-processor machines. Each distribution pattern for a matrix (distribution by rows, by columns, by blocks, etc.) can then be perceived as a particular implementation of this matrix.

When designing an application program that deals with matrices, the choice of the best representation layout for a given matrix is a crucial issue. PALADIN for example encapsulates several alternative representations for matrices (and for vectors as well, though this part of PALADIN is not discussed in this paper), and makes it possible for the application programmer to change the representation format of a matrix at any time during a computation. For example, after a few computation steps an application program may need to convert a sparse matrix into a dense one, because the sparsity of the matrix has decreased during the first part of the computation. Likewise, it may sometimes be necessary to change the distribution pattern of a distributed matrix at run-time in order to adapt its distribution to the requirements of the computation. PALADIN thus provides a facility to redistribute matrices dynamically, as well as a facility to transform dynamically the internal representation format of a matrix (see section 4).

2.3 One Abstraction, Several Implementations

To implement polymorphic aggregates —be they distributed or not— using the facilities of EPEE, we propose a method based on the dissociation of the abstract and operational specifications of an aggregate. The fundamental idea is to build a hierarchy of abstraction levels. Application programs are written in such a way that they operate on abstract data structures, whose concrete implementation is defined independently from the programs that use them.

Eiffel provides all the mechanisms we need to dissociate the abstract specification of an aggregate from the details relative to its implementation. The abstract specification can be easily encapsulated in a class whose interface determines precisely the way an application programmer will view this aggregate.

The distribution of an aggregate is usually achieved in two steps. The first step aims at providing transparency to the user. It consists in performing the actual distribution of the aggregate on the processors of a DCS, while ensuring that the resulting distributed aggregate can be handled in a SPMD program just like its local counterpart in a sequential program. The second step mostly addresses performance issues. It consists in parallelizing some of the features that operate on the distributed aggregate.

One or several distribution patterns must be chosen to spread the aggregate over a DCS. Since we opted for a data parallel approach, each processor will only own a part of the distributed aggregate. The first thing to do is thus to implement a mechanism ensuring a transparent remote access to non local data, while preserving the semantics of local accesses.

When implementing distributed aggregates with EPEE, a fundamental principle is a location rule known as the *Owner Write Rule*, which states that only the processor that owns a part of an aggregate is allowed to update this part. This mechanism is commonly referred to as the *Exec* mechanism in the community of data parallel computing. Similarly, the *Refresh* mechanism ensures that remote accesses are properly dealt with. Both mechanisms have been introduced in [4], and described formally in [3]. The EPEE toolbox provides various facilities for implementing these mechanisms, as illustrated in the following sections with the implementation of distributed matrices.

2.4 Matrices and Vectors in Paladin

Paladin is built around the specifications of the basic entities of linear algebra: matrices and vectors.

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. Representation-dependent features are simply declared, whereas other features are defined —*i.e.,* implemented— directly in Matrix and Vector, as shown below.

In the following we mainly focus on the content of class Matrix. Class Vector is designed in a very similar way. The class Matrix simply enumerates the features that are needed to handle a matrix object, together with their formal properties expressed as assertions (preconditions, postconditions, invariants, etc.), as illustrated in example 2.1.

For the sake of conciseness and clarity, the class Matrix we consider here is a simplified version of the real class implemented in Paladin. The class notably includes some of the most classical linear algebra operations (sum, difference, multiply, transpose, etc.) as well as more complex operations (e.g., LU, LDL^T and QR factorization, triangular system solvers, etc.). It also encapsulates the definition of infix operators that make it possible to write in application programs an expression such as $R := A + B$, where A, B and R refer to matrices.

The resulting class can be thought of as a close approximation of the abstract data type of a matrix entity [1, 5]. A matrix is mainly characterized by its size, stored in attributes *nrow* and *ncolumn*. Routines can be classified in two categories, accessors and operators.

Accessors Accessors are the features that permit to access a matrix in read or write mode. Paladin provides routines for accessing a matrix at different levels. Basic routines *put* and *item* give access to an item of the matrix. The implementation of accessors

Example 2.1 .

```
deferred class MATRIX [T->NUMERIC]
feature -- Attributes
   nrow: INTEGER        -- Number of rows
   ncolumn: INTEGER     -- Number of columns
feature -- Accessors
   item (i, j: INTEGER): T is
         -- Return current value of item(i, j)
      require
         valid_i:  (i > 0) and (i <= nrow)
         valid_j:  (j > 0) and (j <= ncolumn)
      deferred
      end -- item
   put (v: T; i, j: INTEGER) is
         -- Put value v into item(i, j)
      require
         valid_i:  (i > 0) and (i <= nrow)
         valid_j:  (j > 0) and (j <= ncolumn)
      deferred
      ensure
         item (i, j) = v
      end -- put
   row (i: INTEGER): VECTOR [T] is do ... end
   column (j: INTEGER): VECTOR [T] is do ... end
   diagonal (k: INTEGER): VECTOR [T] is do ... end
   submatrix (i, j, k, l: INTEGER): SUB_MATRIX [T] is do ... end
feature -- Operators
   trace: T is do ... end
   random (min, max: T) is do ... end
   add (B: MATRIX [T]) is do ... end
   mult (A, B: MATRIX [T]) is do ... end
   LU is do ... end
   LDLt is do ... end
   Cholesky is do ... end
   -- ...
end -- class MATRIX
```

depends on the format chosen to represent a matrix object in memory. Consequently, in class MATRIX, both accessors *put* and *item* are given a full specification (signature and preconditions and postconditions), but are left deferred.

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 (function *submatrix*). Assume that A is a newly created 5×5 integer matrix. The following code illustrates the use of accessor *submatrix* to fill a section of A with random values (originally all items are set to zero).

$$A = \begin{pmatrix} 0\,0\,0\,0\,0 \\ 0\,0\,0\,0\,0 \\ 0\,0\,0\,0\,0 \\ 0\,0\,0\,0\,0 \\ 0\,0\,0\,0\,0 \end{pmatrix} \xrightarrow{\text{A.submatrix(2,4,2,5).random}} \begin{pmatrix} 0\,0\,0\,0\,0 \\ 0\,6\,1\,9\,6 \\ 0\,2\,7\,2\,8 \\ 0\,7\,3\,5\,1 \\ 0\,0\,0\,0\,0 \end{pmatrix}$$

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.

The set of multilevel accessors actually provides the same abstractions as the syntactic short-cuts frequently used in books dealing with linear algebra, such as [6]. 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 Operators of class MATRIX 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 PALADIN 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.

Although the class MATRIX encapsulates the abstract specification of a matrix object, this does not imply that all features must be kept deferred in this class. Unlike accessors *put* and *item*, operators such as *trace*, *random*, *add*, etc. are features that can generally be given an operational specification based on calls to accessors and other operators. Consequently, the implementation of an operator does not directly depend on the internal representation format of the aggregate considered, because this representation format is masked by the accessors.

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.

3 Replicated and Distributed Matrices

3.1 Sequential Implementation of a Matrix

Once the abstract specification of an aggregate has been encapsulated in a class, it is possible to design one or several descendant classes (*i.e.,* classes that inherit from the abstract class), each descendant encapsulating an alternative implementation of the aggregate. This implementation can either consist in the description of a representation

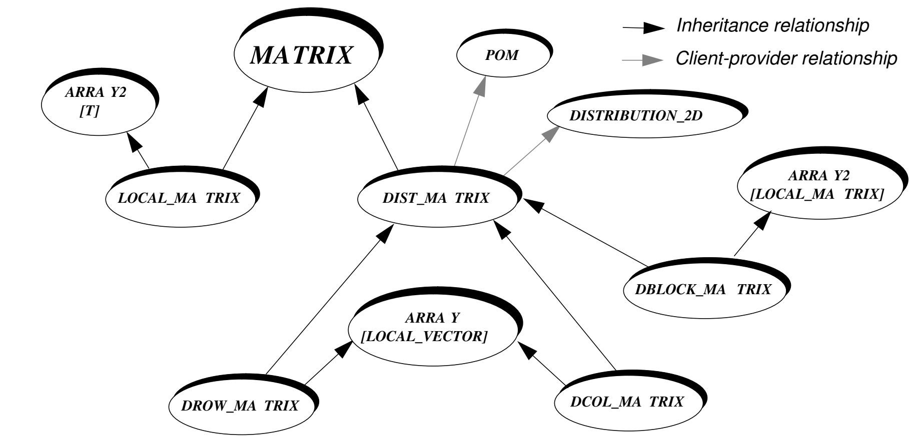

Fig. 2. Inheritance structure for matrix aggregates (partial view)

Example 3.1

```
class LOCAL_MATRIX [T->NUMERIC]
inherit
   MATRIX [T]
   ARRAY2 [T]
      rename height as nrow, width as ncolumn end
creation
   make
end -- class LOCAL_MATRIX
```

format to store the aggregate in the memory of a mono-processor machine, or it can be the description of a pattern to distribute the aggregate on a DCS.

In the following, we show how the mechanism of multiple inheritance helps designing classes that encapsulate fully operational specifications of matrix objects. We first illustrate the approach by describing the design of class LOCAL_MATRIX, which encapsulates a possible implementation for local —*i.e.,* non-distributed— matrix objects. In this class we specify that an object of type LOCAL_MATRIX must be stored in memory as a traditional bi-dimensional array.

The class LOCAL_MATRIX simply combines the abstract specification inherited from MATRIX together with the storage facilities provided by the class ARRAY2 available in most Eiffel libraries (see also figure 2). The text of LOCAL_MATRIX is readily written, thanks to the mechanism of multiple inheritance: the effort of design only comes down to combining the abstract specification of class MATRIX with the implementation facilities offered by ARRAY2, and ensuring that the names of the features inherited from both ancestor classes are matched correctly. In the example 3.1, the attributes *height* and *width* of class ARRAY2 are matched with the attributes *nrow* and *ncolumn* of class MATRIX through *renaming.*

A library designed along these lines 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 the designer of the library, providing a new representation variant for a matrix or a vector usually consists in assembling existing classes to produce a new one. Very often this process does not imply any development of new code.

Unlike the abstract class MATRIX, the class LOCAL_MATRIX is a concrete (or effective) class, which means that it can be instantiated (Assuming that no operator has been left deferred in class MATRIX). It is thus possible to create objects of type LOCAL_MATRIX in an application program, and to invoke on these objects some of the accessors and operators defined in MATRIX.

3.2 Distribution of Matrices in Paladin

The PALADIN approach to the distribution of matrices is quite similar to that of High Performance Fortran (HPF) [9]. The main difference is that HPF is based on weird extensions of the FORTRAN 90 syntax (distribution, alignment and mapping directives) whereas PALADIN only uses normal constructions of the Eiffel language.

Distributed matrices are decomposed into blocks, which are then mapped over the processors of the target DCS. Managing the distribution of a matrix implies a great amount of fairly simple but repetitive calculations, such as those that aim at determining the identity of the processor that owns the item (i, j) of a given matrix, and the local address of this item on this processor. The Features for doing such calculations have been encapsulated in a class DISTRIBUTION_2D, which allows the partition and distribution of 2-D data structures. The class DISTRIBUTION_2D is actually designed by inheriting two times from a more simple class DISTRIBUTION_1D. Hence, a class devoted to the distribution of 3-D data structures could be built just as easily.

The application programmer describes a distribution pattern by specifying the size of the index domain considered, the size of the basic building blocks in this domain, and how these blocks must be mapped on a set of processors. The definition of the mapping function has intentionally been left out of class DISTRIBUTION_2D and encapsulated in a small hierarchy of classes devoted to the mapping of 2-D structures on a set of processors (see class MAPPING_2D in example 3.2).

PALADIN includes two effective classes that permit to map the blocks of a distributed matrix either row-wise or column-wise on a set of processors. In the class ROW_WISE_MAPPING, for example, the feature *map_block* is implemented as shown in example 3.3.

The keyword `expanded` in the first line of this code implies that instances of class ROW_WISE_MAPPING are value objects. Any attribute declared as being of type ROW_WISE_MAPPING can be directly handled as an object of type ROW_WISE_MAPPING.

The implementation of COLUMN_WISE_MAPPING is of course very similar to that of ROW_WISE_MAPPING. Any user could easily propose alternative mapping policies (random mapping, diagonal-wise mapping, etc.): the only thing a user must do is design a new class that inherits from MAPPING_2D and that encapsulates an original implementation of the feature *map_block*.

Example 3.2 .

```
deferred class MAPPING_2D
feature
   map_block (bi, bj, bimax, bjmax, nproc:  INTEGER): INTEGER is
         -- Maps block(bi, bj) on a processor whose identifier
         -- must be in the range [0, nproc]
      require
         bi_valid:  (bi >= 0) and (bi <= bimax)
         bj_valid:  (bj >= 0) and (bj <= bjmax)
      deferred
      ensure
         (Result >= 0) and (Result < nproc)
      end -- map_block
end -- class MAPPING_2D
```

Example 3.3 .

```
expanded class ROW_WISE_MAPPING
inherit MAPPING_2D
feature
   map_block (bi, bj, bimax, bjmax, nproc: INTEGER): INTEGER is
      do
         Result := (bi * (bjmax + 1) + bj) \\ nproc
      end -- map_block
end -- class ROW_WISE_MAPPING
```

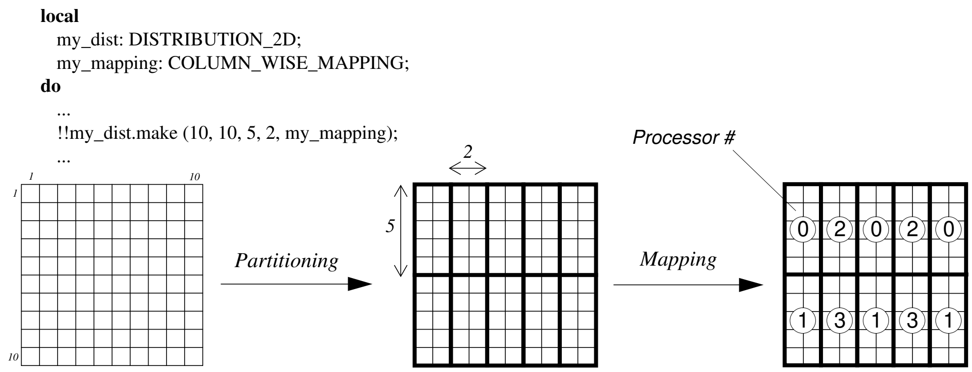

Fig. 3. Example of a distribution allowed by class DISTRIBUTION_2D

Figure 3 shows the creation of an instance of DISTRIBUTION_2D. The creation feature takes as parameters the size of the index domain considered, the size of the building blocks for partitioning this domain, and a reference to an object whose type conforms to —*i.e.*, is a descendant of— MAPPING_2D. The instance of DISTRIBUTION_2D created in figure 3 will thus permit to manage the distribution of a 10×10 index domain partitioned into 5×2 blocks mapped column-wise on a set of processors. Figure 3 also shows the resulting mapping on a parallel architecture providing 4 processors.

Each distributed matrix must be associated at creation time with an instance of DISTRIBUTION_2D, which plays the role of a distribution template for this matrix. The distribution pattern of a matrix can either be specified explicitly —in that case a new instance of DISTRIBUTION_2D is created for the matrix—, or implicitly by passing either a reference to an already existing distributed matrix or a reference to an existing distribution template as a parameter. Several distributed matrices can thus share a common distribution pattern by referencing the same distribution template.

3.3 Implementation of Distributed Matrices

The accessors *put* and *item* declared in class MATRIX must be implemented in accordance with the *Exec* and *Refresh* mechanisms introduced in section 2.3. This is achieved in a new class DIST_MATRIX that inherits from the abstract specification encapsulated in class MATRIX (see example 3.4).

The accessor *put* is defined so as to conform to the *Owner Write Rule*: when an SPMD application program contains an expression of the form `M.put(v, i, j)` —with `M` referring to a distributed matrix— the processor that owns item (i, j) is solely capable of performing the assignment. In the implementation of the feature *put*, the assignment is thus conditioned by a locality test using the distribution template (feature *dist*) of the matrix (see the lines 20–25 of the example 3.4).

The accessor *item* must be defined so that remote accesses are properly dealt with: when an SPMD application program contains an expression such as `v := M.item(i, j)`, the function *item* must return the same value on all the processors. Consequently, in the implementation of the feature *item*, the processor that owns item (i, j) broadcasts its value so that all the other processors can receive it (see the lines 16–19 of the example 3.4). The invocation `M.item(i, j)` thus returns the same value on all the processors implied in the computation (the communication primitives are provided by the class POM of the EPEE toolbox).

This implementation of the features *put* and *item* deals with the distribution of data, but it does not deal with the actual *access* to local data. This problem must be tackled in the local accessors *local_put* and *local_item*, whose implementation is closely dependent on the format chosen to represent a part of the distributed matrix on each processor. Since there may be numerous ways to store a distributed matrix in memory (*e.g.,* the distributed matrix may be dense or sparse), both features *local_put* and *local_item* are left deferred in class DIST_MATRIX. They must be defined in classes that descend from DIST_MATRIX and that encapsulate all the details relative to the internal representation of distributed matrices.

The class DBLOCK_MATRIX presented in example 3.5 is one of the many possible descendants of DIST_MATRIX (see figure 2). It inherits from DIST_MATRIX as well as from ARRAY2[LOCAL_MATRIX], and therefore implements a dense matrix distributed by blocks as a 2-D table of local matrices. Each entry in this table references a building block of the distributed matrix, stored in memory as an instance of LOCAL_MATRIX (see figure 4). A void entry in the table means that the local processor does not own the corresponding block matrix. In DBLOCK_MATRIX, the local accessors *local_put* and *local_item* are defined so as to take into account the indirection due to the table.

Example 3.4

```
indexing
   description: "Abstract matrix distributed along a template"

deferred class DIST_MATRIX [T->NUMERIC]
inherit
   MATRIX [T]    -- Abstract specification
feature -- Creation
   make (rows, cols, bfi, bfj: INTEGER; alignment: MAPPING_2D) is
      deferred end
   make_from (new_dist: DISTRIBUTION_2D) is  deferred end
   make_like (other: DIST_MATRIX) is deferred end
feature -- Distribution template
   dist: DISTRIBUTION_2D
feature -- Accessors
   item (i, j: INTEGER): T is
         -- element (i,j) of the Matrix, read using the Refresh mecanism
      do
         if dist.item_is_local(i, j) then
            Result := local_item (i, j)  -- I am the owner
            POM.broadcast (Result)  -- so I send the value to others
         else -- I'm not the owner, I wait for the value to be sent to me
            Result := POM.receive_from (dist.owner_of_item (i, j))
         end -- if
      end -- item
   put (v:  T; i, j:  INTEGER) is
         -- write the element (i,j) of the Matrix, the Owner Write Rule
      do
         if dist.item_is_local(i, j) then
            local_put (v, i, j) -- Only the owner writes the data
         end -- if
      end -- put

feature {DIST_MATRIX}  -- Communication features
   POM: POM

end -- DIST_MATRIX [T]
```

The class hierarchy that results from this approach is clearly organized as a layering of abstraction levels. At the highest level, the class MATRIX encapsulates the abstract specification of a matrix entity. The class DIST_MATRIX corresponds to an intermediate level, where the problem of the distribution of a matrix is solved, while the problem of the actual storage of the matrix in memory is deferred. At the lowest level, classes such as DBLOCK_MATRIX provide fully operational specifications for distributed matrices.

Besides DBLOCK_MATRIX, the class hierarchy of PALADIN includes two classes DCOL_MATRIX and DROW_MATRIX that encapsulate alternative implementations for row-wise and column-wise distributed matrices. In these classes, distributed matri-

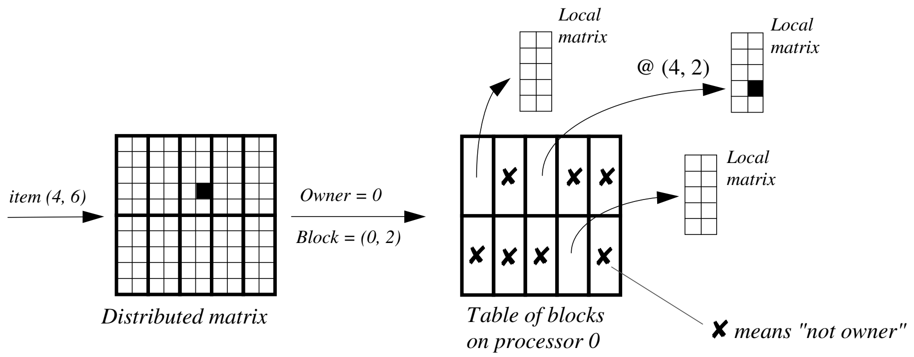

Fig. 4. Internal representation of a matrix distributed by blocks

Example 3.5

```
indexing
   description: "Matrix distributed by blocks"

class DBLOCK_MATRIX [T->NUMERIC]
inherit
   DIST_MATRIX [T]
   ARRAY2 [LOCAL_MATRIX[T]]
      rename
         make as make_table,
         put as put_block, item as local_block
      end
feature -- ...
end -- class DBLOCK_MATRIX [T->NUMERIC]
```

ces are implemented as tables of local vectors. This kind of implementation fits well application programs that perform many vector-vector operations. Other kinds of distribution patterns or other kinds of representation formats could be proposed. One could for example think of exotic distribution patterns based on a decomposition into heterogeneous blocks or on a random mapping policy. One could also decide to provide an implementation *ad hoc* for triangular or band distributed matrices. With the object-oriented approach, the extensibility of a class hierarchy such as that of Paladin has virtually no limit. It is always possible to incorporate new classes seamlessly in a pre-existing class hierarchy.

4 Dealing with multiple representations

4.1 Interoperability

One of the major advantages of this class organization is that it ensures the interoperability of all matrices and vectors. A feature declared —and possibly implemented—

in class MATRIX is inherited by all the descendants of this class. Hence a feature such as *cholesky*, which performs a Cholesky factorization, can operate on any matrix that satisfies the preconditions of the feature: the matrix must be square symmetric definite positive. This feature therefore operates on a local matrix as well as on a distributed one. In the library, a parallel version of the Cholesky algorithm is actually provided for distributed matrices, but this optimization remains absolutely transparent for the user who keeps using the feature the same way.

Interoperability also goes for algorithms that admit 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.

4.2 Dynamic Redistribution

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. A row-wise distributed matrix, for example, can be "transformed" dynamically into a column-wise distributed matrix, assuming that this new representation is likely to lead to better performances in some parts of an application program. The conversion mechanism therefore plays the role of a redistribution facility.

An algorithm that permits to redistribute a matrix can be obtained quite simply using the communication facilities provided by class POM and the distribution facilities provided by class DISTRIBUTION_2D. Such a redistribution facility was implemented as shown below in class DBLOCK_MATRIX.

In this code, a temporary instance of DBLOCK_MATRIX named *tmp_matrix* is created according to the desired distribution pattern. Block matrices are then transferred one after another from the current matrix to *tmp_matrix*. Once the transfer is over, the attribute *dist* of the current matrix is re-associated with the new distribution template. Its former distribution template can then be collected by the garbage collector of the runtime system, unless this template is still used by another distributed matrix. Likewise, the attribute *area*, which actually refers to the table of block matrices of the current matrix, is re-associated so as to refer to the table of *tmp_matrix*. The former block table can then be also collected by the garbage collector. When the feature *redistribute* returns, the current matrix is a matrix whose distribution to the pattern described by *new_dist* and its internal representation relies on the newly created table of block matrices.

Notice that this implementation of the feature *redistribute* can only redistribute a matrix if the source and the target distribution patterns have the same block size (see the precondition in the code of the feature *redistribute*). The code of the feature *redistribute* reproduced here is actually a simplified version of the code implemented in DBLOCK_MATRIX. The real code is more flexible (a matrix can be redistributed even if the size of blocks must change during the process), it does not rely on a temporary matrix but directly creates and handles a new table of block matrices. Moreover, the garbage collection is performed on the fly: on each processor the local blocks that are

Example 4.1

```
redistribute (new_dist: DISTRIBUTION_2D) is
   require
      new_dist_valid: (new_dist /= Void)
      compat_dist: (dist.bfi = new_dist.bfi) and (dist.bfj = new_dist.bfj)
   local
      bi, bj, source, target: INTEGER
      tmp_matrix:  DBLOCK_MATRIX [T]
   do
      !!tmp_matrix.make_from (new_dist)
      from bi := 0 until bi > dist.nbimax loop
         from bj := 0 until bj > dist.nbjmax loop
            source := dist.owner_of_block (bi, bj)
            target := tmp_matrix.dist.owner_of_block (bi, bj)
            if (source = POM.my_node) then
               -- Send block matrix to target
               local_block (bi, bj).send (target)
            end -- if
            if (target = POM.my_node) then
               -- Receive block matrix from source
               tmp_matrix.local_block (bi, bj).recv_from (source)
            end -- if
            bj := bj + 1
         end -- loop
         bi := bi + 1
      end -- loop
      dist := tmp_matrix.dist
      area := tmp_matrix.area
   end -- redistribute
```

sent to another processor are collected by the garbage collector immediately after they have been sent. Data exchanges are also performed more efficiently: the sequencing constraints imposed by the *Refresh/Exec* model in the former code are relaxed so that the resulting implementation of *redistribute* allows more concurrency. The real code encapsulated in the feature *redistribute* is thus more efficient than the code reproduced above, but it is also longer and more complex. This is the reason why we preferred to reproduce a simple implementation of *redistribute* here.

Anyway, whatever the actual complexity of the algorithm encapsulated in the feature *redistribute*, it does not shows through the interface of class DBLOCK_MATRIX. From the viewpoint of the application programmer, an instance of DBLOCK_MATRIX can thus be redistributed quite simply. Consider the small SPMD application program of the example 4.2.

Imagine that in this application the requirements of the computation impose that matrices A and B be distributed differently in the first part of the concurrent execution. On the other hand, the second part of the computation requires that A and B have the same distribution. Then the redistribution facility encapsulated in class DBLOCK_MATRIX can be used to achieve the redistribution.

Example 4.2

```
local
   A, B: DBLOCK_MATRIX [DOUBLE]
do
   !!A.make (100, 100, 5, 2, ROW_WISE_MAPPING)
   !!B.make (100, 100, 7, 3, COLUMN_WISE_MAPPING)
     ...(1)...
   B.redistribute (A.dist)
     ...(2)...
end
```

Other classes of Paladin (*e.g.,* Dist_Matrix, Dcol_Matrix, Drow_Matrix) also encapsulate a version of the feature *redistribute*, whose implementation fits the characteristics of their distribution pattern.

4.3 Matrix type conversion

Eiffel, like most statically typed object-oriented languages, does not allow for objects to change their internal structure at runtime: once an object has been created, its internal organization is in a way "frozen". Thus, in Paladin, there is for example no way one can transform an object of type Local_Matrix in an object of type Dblock_Matrix. However, we can go round this constraint and propose a close approximation of "polymorphic" matrices, using the only really polymorphic entities available in Eiffel: references.

Whenever we need to change the internal representation of a matrix aggregate, the conversion must be performed in three steps. At first, a new matrix aggregate must be created, whose dynamic type conforms to the desired internal representation. Next, data must be "transferred" from the original aggregate into the new one. Finally, the reference associated with the original aggregate must be re-associated with the newly created one. This conversion procedure is illustrated below.

Assume that in an application program a local matrix is created and associated with reference `M` (example 4.3).

After some computation (part 1) it becomes necessary to transform this local matrix into a distributed one. An instance of type Dblock_Matrix is created and associated with a temporary reference `N`. The information encapsulated in the original local matrix is copied in the distributed one using routine *convert*. Once the copy is complete, attribute M is re-associated with the newly created matrix thanks to a polymorphic assignment, so that the programmer can still refer to the matrix using attribute M in the remaining of the application program. The computation goes on using the distributed matrix (part 2). The conversion process is illustrated in figure 5.

Conceptually, the feature *convert* simply performs a copy from the source matrix into the target one. It simply requires that both matrices have the same size. In class Matrix, the feature *convert* can be given a very simple implementation, based on two nested loops and calls to accessors *put* and *item*. However, this implementation, which does not depend on the internal representation formats of the source and target

Example 4.3

```
local
   M, N: MATRIX [DOUBLE]
do
   !LOCAL_MATRIX[DOUBLE]!M.make (...)
   ...(1)...
   !DBLOCK_MATRIX[DOUBLE]!N.make (...)
   N.convert (M)
   M := N
   ...(2)...
end
```

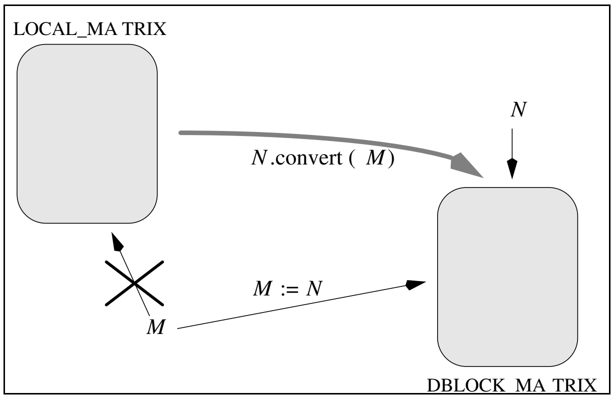

Fig. 5. Example of matrix conversion

matrices, should only be considered as a default implementation. Better implementations of *convert* can be encapsulated in descendants of class MATRIX, using some of the optimization techniques discussed in [7].

Notice that this method to change the type of an aggregate actually requires that a new object be created. This is acceptable, since the Eiffel garbage collector ensures that the object corresponding to the "obsolete" representation of the aggregate will be collected after the conversion is over. Actually, the main problem with this conversion mechanism lies in the lack of transparency for the application programmer, who must explicitly declare a temporary reference, create a new aggregate of the desired dynamic type, invoke the feature *convert* on this object and eventually reassign the reference bound to the original object so that it now refers to the new object.

Another problem concerns *aliases*. An application program may reference the same matrix through many variables. The type conversion method presented above does not deal with this aliasing problem. A number of approaches have been proposed to solve this kind of problem. For example, we can maintain a table referencing all the objects that may be subject to type conversions. All subsequent accesses to these objects then go through this table. Another method consist in keeping a list of client objects in each polymorphic object. Such an object can then inform its clients upon type conversion.

Example 4.4

```
class POLY_MATRIX

feature {NONE} -- Reference to a matrix container
   container: MATRIX;

feature -- Basic Accessors
   item (i, j: INTEGER): DOUBLE is do Result := container.item (i, j); end;
   put (v: like item; i, j: INTEGER) is do container.put (v, i, j); end;
   ...
feature -- Operators
   mult (A, B: like Current) is
      do
         container.mult (A.container, B.container);
      end;
   ...
end -- POLY_MATRIX
```

Because they are costly to implement, none of these approaches is fully satisfactory. A better solution would be to encapsulate the type conversions.

4.4 Towards Full Polymorphic Matrices

The only feasible solution to provide full polymorphic matrices in the context of a language such as Eiffel is to introduce a level of indirection. This boils down to introducing a distinction between the data structure containing the matrix data and the concept of a polymorphic matrix. A polymorphic matrix is just a client of the matrix class defined previously, and is thus able to dynamically change its representation (example 4.4).

A POLY_MATRIX has the same interface as the class MATRIX, but privately uses a MATRIX for its implementation (this is the meaning of the clause `{NONE}` in example 4.3). Its basic accessors *put* and *item* are defined so as to access the data stored in the container, which can be any subtype of the class MATRIX. In the same way, each operator of the class POLY_MATRIX is defined as to call the corresponding operator in the container.

A new set of routine is also available in the POLY_MATRIX class for it to be able to dynamically change its internal representation, that is to polymorph itself. For example, a POLY_MATRIX can acquire a LOCAL_MATRIX representation with the procedure *become_local* presented in example 4.5).

The performance overhead of the extra indirection is negligible with respect to the algorithmic complexity of the operations on the large matrices considered in PALADIN.

4.5 Using Polymorphic Matrices

In PALADIN, the powerful abstraction of polymorphic matrices, together with features such as *become_xxx*, *redistribute* or *convert* are made available to the application programmer. Yet, they could also be invoked automatically within the library whenever

Example 4.5

```
class POLY_MATRIX
   ...
feature -- Internal representation conversion
   become_local is
      local
         new_container: like container;
      do
         -- Create a new matrix container with type as required
         !LOCAL_MATRIX!new_container.make (nrow, ncolumn);
         -- Transfer data from old matrix container to new one
         new_container.convert (container);
         -- Adopt new matrix container and discard old one
         container := new_container;
      end; -- become_local
   ...
end -- class POLY_MATRIX
```

an operator requires a particular distribution pattern of its operands. For example, any operator dealing with a distributed matrix could be implemented so as to systematically redistribute this matrix according to its needs prior to beginning the actual computation. If all operators in PALADIN were implemented that way, the application programmer would not have to care about distribution patterns anymore, all matrices being redistributed transparently as and when needed. Yet, redistributing a matrix — or changing its type— is a costly operation, so that this approach would probably lead to concurrent executions in which most of the activity would consist in redistributing matrices or vectors. The best approach is probably an intermediate between manual and automatic redistribution.

5 Conclusion

An OO library is built around the specifications of the basic data structures it deals with. The principle of dissociating the abstract specification of a data structure (somewhat its abstract data type) from any kind of implementation detail enables the construction of reusable and extensible libraries of parallel software components. Using this approach, we have shown in this paper that existing sequential OO languages are versatile enough to enable an efficient and easy use of DCSs. Thanks to the distributed data structures of a parallel library such as PALADIN, any programmer can write an application program that runs concurrently on a DCS. The parallelization can actually proceed in a seamless way: the programmer first designs a simple application using only local aggregates. The resulting sequential application can then be transformed into a SPMD application, just by changing the type of the aggregates implied in the computation. For large computations, we have shown that the overhead brought about by the higher level of OO languages remains negligible. Using the same framework, we are in the process of extending PALADIN to deal with sparse computations and control parallelism.

Although our approach hides a lot of the tedious parallelism management, the application programmer still remains responsible for deciding which representation format is the most appropriate for a given aggregate. Hence, when transforming a sequential program into an SPMD one, the programmer must decide which aggregate shall be distributed and how it shall be distributed. This may not always be an easy choice. Finding a "good" distribution may be quite difficult for complex application programs, especially since a distribution pattern that may seem appropriate for a given computation step of an application may not be appropriate anymore for the following computation step. Dynamically redistributing aggregates as and where needed (as is possible in PALADIN) might be a way to go round this problem. The redistribution could be controlled by the user, or even encapsulated with the methods needing special distributions to perform an operation efficiently. On this topic the OO approach has an important edge over HPF compilers that can only bind methods to objects statically, thus producing very inefficient code if the dynamic redistribution pattern is not trivial.

References

1. H. Abelson, G. Jay Sussman, and J. Sussman. – *Structure and Interpretation of Computer Programs.* – MIT Press, Mac Graw Hill Book Company, 1985.
2. P. America. – Pool-T: A parallel object-oriented programming. – In A. Yonezawa, editor, *Object-Oriented Concurrent Programming*, pages 199–220. The MIT Press, 1987.
3. F. André, J.L. Pazat, and H. Thomas. – Pandore: a system to manage data distribution. – In *ACM International Conference on Supercomputing*, June 11-15 1990.
4. D. Callahan and K. Kennedy. – Compiling programs for distributed-memory multiprocessors. – *The Journal of Supercomputing*, 2:151–169, 1988.
5. L. Cardelli and P. Wegner. – On understanding types, data abstraction, and polymorphism. – *ACM Computing Surveys*, 17(4):211–221, 1985.
6. G.H. Golub and C.F. Van Loan. – *Matrix Computations.* – The Johns Hopkins University Press, 1991.
7. 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.
8. F. Hamelin, J.-M. Jézéquel, and T. Priol. – A Multi-paradigm Object Oriented Parallel Environment. – In H. J. Siegel, editor, *Int. Parallel Processing Symposium IPPS'94 proceedings*, pages 182–186. IEEE Computer Society Press, April 1994.
9. HPF-Forum. – High Performance Fortran Language Specification. – Technical Report Version 1.0, Rice University, May 1993.
10. J.-M. Jézéquel. – EPEE: an Eiffel environment to program distributed memory parallel computers. – *Journal of Object Oriented Programming*, 6(2):48–54, May 1993.
11. C. Pancake and D. Bergmark. – Do parallel languages respond to the needs of scientific programmers? – *IEEE COMPUTER*, pages 13–23, December 1990.
12. Leslie G. Valiant. – A bridging model for parallel computation. – *CACM*, 33(8), Aug 1990.
13. Akinori Yonezawa, Jean-Pierre Briot, and Etsuya Shibayama. – Object-oriented concurrent programming in ABCL/1. – In *OOPSLA'86 Proceedings*, September 1986.