An Object-Oriented Framework for Supercomputing

Frédéric Guidec, Jean-Marc Jézéquel, Jean-Lin Pacherie

An Object Oriented Framework for Supercomputing

F. Guidec, J.-M. Jézéquel and J.-L. Pacherie

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. We propose a framework where the tricky parallel codes can be encapsulated in object oriented software components that can be reused, combined and customized in confidence by application programmers. We propose to use a kind of parallelism known as data-parallelism, encapsulated within classes of a purely sequential object-oriented language (Eiffel), using the SPMD (Single Program Multiple Data) programming model. We define a set of methodological rules to help a programmer design and exploit parallel software components within this framework. We illustrate our approach with the example of Paladin, an object-oriented library devoted to linear algebra computation on DCSs. Paladin relies on widely advertised object-oriented features to help a numerical programmer develop or parallelize application programs, following the guidelines of modern software engineering. Most notably, modularity and encapsulation are used to hide data distribution and code parallelization within classes presenting sequential interfaces, while inheritance allows a transparent reuse of basic parallel patterns in new applications. We discuss the implementation of a demonstrator of such a library as well as performance related aspects.

Keywords: Programming Environments for Distributed Computing Systems, Data-Parallelism and SPMD Model, Linear Algebra, Reusable Object-Oriented Libraries

1 Introduction

Although the physical world they model is inherently parallel, scientific programmers usually rely on sequential techniques and algorithms to solve their problems, because these algorithms often present a better computational complexity than possible direct solutions. Actually, their interest in concurrency mainly results from their desire to improve the performance of sequential algorithms when performing large-scale numerical computations [Pancake et al.90]. This interest is rising a lot now that powerful Distributed Computing Systems (DCSs) consisting of hundreds of processors are commercially available. However, these DCSs presently suffer from programming environments dating back to the days of punch cards and paper tape, and offering very little in terms of software engineering. Considering the tedious tasks of parallelization, distribution, process handling and communication management, which are not among the most attractive features of DCSs, scientific programmers are generally reluctant to cope with the manual porting of their applications on such machines.

In order to alleviate the porting of scientific applications on parallel computers, one of the main directions investigated so far consists in the design of semi-automatic parallelizing compilers. This approach has already produced research prototypes like SUPERB [Zima et al.88], PANDORE [Andre et al.90], or FORTRAN-D [Callahan et al.88], and industrial strength compilers for High Performance Fortran (HPF) [Hpf forum93] are expected in the near future. However, building compilers that generate efficient code for a language like HPF turns out to be more difficult than was originally expected and, so far, speedups achieved by automatic transformation alone are still disappointing. Although many optimization techniques are known (message vectorization, loop restriction, etc.), their automatic application requires a comprehensive analysis of the program being compiled. It is also unlikely that the first generation of semi-automatic parallelizing compilers could deal efficiently with application programs involving data redistribution or irregular computation patterns. Moreover, the semi-automatic parallelization approach offers no easy way to exploit the many hand-coded parallel algorithms that have been developed since parallelism became a mainstream research activity.

Propositions based on AI techniques have been made to deal with these problems [Fahringer et al.92]. Based on the observation that only a limited number of basic operations are needed in the time consuming inner loops of numeric programs, it is proposed in PARAMAT [Kessler94] to automatically identify these basic operations and substitute sequential expressions by equivalent optimized parallel counterparts. Although this approach is interesting for parallelizing pre-existing code, it is not the most straightforward way to develop new programs.

Actually, once the basic operations have been identified and stored in a library, it should be possible to reuse them directly as building blocks for new application programs. However, classical libraries built for FORTRAN programs are not versatile enough to deal with the complex problems bound to the management of distributed data structures that can be dynamically redistributed. We claim that object-oriented techniques are helpful to bring in such versatility and, in a larger extent, to ease the programming of DCSs.

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 the description of set of objects that share a common structure and/or a common behavior. An object is simply an instance of a class.

We show in this paper that beyond fostering the construction of reusable and extensible libraries of parallel software components, existing sequential object-oriented languages enable an efficient and transparent use of DCSs. In section 2 we present a programming model allowing the encapsulation of data distribution and parallelism in classes presenting a sequential interface, thus fostering a transparent use of the underlying DCS. We show in section 3 how this can be achieved in an efficient and scalable way, while preserving the user friendliness of the library.

All along this paper, we illustrate our approach with the example of PALADIN, an object-oriented library devoted to linear algebra computation on DCSs. PALADIN is based on EPEE, an Eiffel Parallel Execution Environment designed in our lab [Jezequel93]. EPEE can actually be seen as a kind of a toolbox: it mainly consists of a set of cross-compilation tools that allow the generation of executable code for —potentially— any DCS (to date, network of Unix workstations, Intel iPSC/2 and Paragon XP/S). It also includes a set of Eiffel classes that provide facilities for distributing data over a DCS and for handling data exchanges between the processors of a DCS in a portable way. Common data distribution patterns and computation schemes are factorized out and encapsulated in abstract parallel classes, which can be reused by means of multiple inheritance. These classes include parameterized general purpose algorithms for traversing and performing customizable actions on generic data structures.

Note that EPEE is based on Eiffel [Meyer88] because this language features all the concepts we need (*i.e.* strong encapsulation, static type checking, multiple inheritance, dynamic binding and genericity), and has clearly defined syntax and semantics. However any other statically typed OO language could have been used instead (*e.g.* Modula-3, Ada 95, or C++).

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 [Valiant90]. A computation that fits the BSP model can be seen as a succession of parallel phases separated by synchronizations and sequential phases.

In EPEE, the implementation of Valiant's model is 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 [Hamelin et al.94].

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 [America87] or ABCL/1 [Yonezawa et al.86], 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: One Abstraction, Several Implementations

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.

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.

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.

Object-oriented languages provide 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 methods 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 [Callahan et al.88], and described formally in [Andre et al.90].

The accessors of a distributed aggregate must be defined according to the *Refresh* and *Exec* mechanisms. The EPEE toolbox provides various facilities for doing so, as illustrated in the following sections with the implementation of distributed matrices.

2.3 Example: the Matrix Aggregate in Paladin

In PALADIN, the abstract specification of a matrix entity is encapsulated in a class MATRIX, that simply enumerates the attributes and the methods that are needed to handle a matrix object, together with their formal properties expressed as assertions (preconditions, postconditions, invariants, etc., see example 1).

For the sake of conciseness and clarity, the class MATRIX we consider in this paper is a simplified version of the real class implemented in PALADIN. The real class MATRIX is generic and thus allows the instantiation of real and double precision matrices, of complex matrices, etc. It also encapsulates the abstract or operational specification of many more operators, most of which are defined directly in MATRIX. 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 an 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 [Abelson et al.85, Cardelli et al.85]. A matrix is mainly characterized by its size, stored in attributes `nrow` and `ncolumn`. Methods can be classified in two categories, accessors and operators. Accessors such as `put` and `item` are methods that permit to access a matrix in read or write mode.

The implementation of accessors depends on the format chosen to represent a matrix object in memory. Consequently, in class MATRIX, both accessors `put` and `item` are left deferred, that is, they are declared but not defined (a *deferred* method in Eiffel is equivalent to a pure virtual function in C++). Their signature is specified

Example 1

```
deferred class MATRIX
feature -- Attributes
   nrow: INTEGER        -- Number of rows
   ncolumn: INTEGER     -- Number of columns

feature -- Accessors
   item (i, j: INTEGER): DOUBLE 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: DOUBLE; 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

feature -- Operators
   trace: DOUBLE is do ... end
   random (min, max: DOUBLE) is do ... end
   add (B: MATRIX) is do ... end
   mult (A, B: MATRIX) is do ... end
   LU is do ... end
   LDLt is do ... end
   Cholesky is do ... end
   ...
end -- class MATRIX
```

(number of arguments and types of these arguments), as well as their main properties (preconditions and postconditions).

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 methods that can generally be given an operational specification based on calls to accessors and other operators. Consequently, the implementation of an operator (*e.g.*, the `trace` of a matrix, which is the sum of its diagonal elements, see example 2) does not directly depend on the internal representation format of the aggregate considered, because this representation format is masked by the accessors.

Example 2

```
trace: DOUBLE is
    require
        is_square: (nrow = ncolumn)
    local
        i: INTEGER
    do
        from i := 1 until i > nrow loop
            Result := Result + item (i, i)
            i := i + 1
        end -- loop
    end -- trace
```

Besides the basic accessors `put` and `item`, the real class MATRIX additionally provides higher level accessors that allow the application programmer to handle a row, a column or a diagonal of a matrix object as a vector entity, and a rectangular section of this matrix as a plain matrix. These high level accessors actually provide a "view" on a section of a matrix. Modifying a view is equivalent to directly modifying the corresponding section. For example, the expression `M.row(4).random(-5.0, +5.0)` initializes the 4^{th} row of matrix `M` with values taken in the range $[-5.0, +5.0]$.

2.4 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 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.

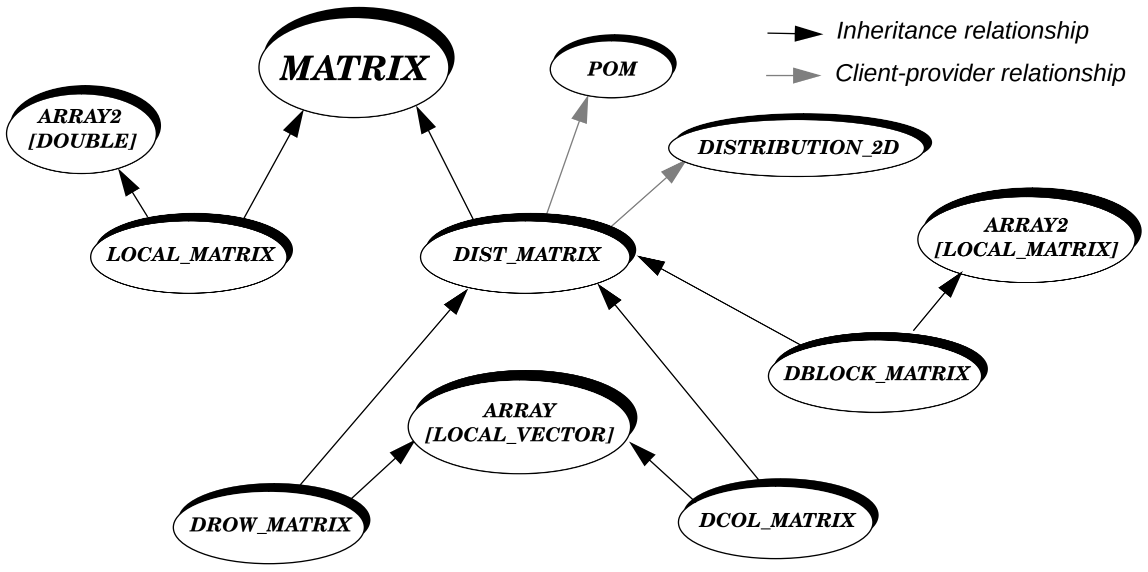


Figure 1: Inheritance structure for matrix aggregates (partial view)

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 standard class ARRAY2, one of the many classes of the Eiffel library (see also figure 1). 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 methods inherited from both ancestor classes are matched correctly (in example 3, the attributes `height` and `width` of class ARRAY2 are matched with the attributes `nrow` and `ncolumn` of class MATRIX).

Example 3

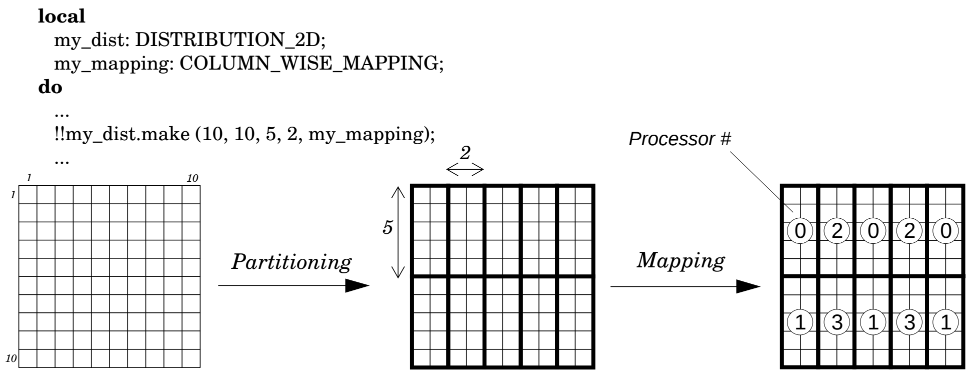
```
class LOCAL_MATRIX
inherit
    MATRIX
    ARRAY2 [DOUBLE]
        rename height as nrow, width as ncolumn end
creation
    make
end -- class LOCAL_MATRIX
```

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.

2.5 Distribution of Matrices in Paladin

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. Methods for doing such calculations have been encapsulated in a class DISTRIBUTION_2D, which allows the partition and distribution of 2-D data structures[1].

The parameterization of the creation method of class DISTRIBUTION_2D is inspired from the syntax used in the HPF directive DISTRIBUTE, and it has roughly the same expressiveness (as far as distributed matrices or 2-D arrays are concerned). 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.



```
local
  my_dist: DISTRIBUTION_2D;
  my_mapping: COLUMN_WISE_MAPPING;
do
  ...
  !!my_dist.make (10, 10, 5, 2, my_mapping);
  ...
```

Figure 2: Example of a distribution allowed by class DISTRIBUTION_2D.

Figure 2 shows the creation of an instance of DISTRIBUTION_2D. The creation method takes as parameters the size of the index domain considered, the size of

[1]The class DISTRIBUTION_2D was 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 building blocks for partitioning this domain, and a reference to an object whose type determines the kind of mapping required. The instance of DISTRIBUTION_2D created in figure 2 thus permits to manage the distribution of a 10×10 index domain partitioned into 5×2 blocks mapped column-wise on a set of processors. Figure 2 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.

2.6 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 2.2. This is achieved in a new class DIST_MATRIX that inherits from the abstract specification encapsulated in class MATRIX.

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 method `put`, the assignment is thus conditioned by a locality test using the distribution template of the matrix (see example 4).

Example 4

```
put (v: DOUBLE; i, j: INTEGER) is
    do
        if dist.item_is_local(i, j) then
            local_put (v, i, j)
        end -- if
    end -- put
```

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 method `item`, the processor that owns item (i, j) broadcasts its value so that all the other processors can receive it (see example 5). The invocation `M.item(i, j)` thus returns the same value on all the processors implied in the computation.

Example 5

```
item (i, j: INTEGER): DOUBLE is
   do
      if dist.item_is_local(i, j) then
         Result := local_item (i, j)
         POM.broadcast (Result)
      else
         Result := POM.receive_from (dist.owner_of_item (i, j))
      end -- if
   end -- item
```

The communication primitives are provided by the class Pom, which is part of the EPEE toolbox and constitutes an interface between the Eiffel world and the POM communication and observation library [Guidec et al.95]. This library was designed so as to provide a homogeneous interface upon the many communication kernels available on current parallel architectures. Moreover, it can be easily and efficiently implemented on most of these architectures.

The implementation of methods `put` and `item` shown in examples 4 and 5 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 methods `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.

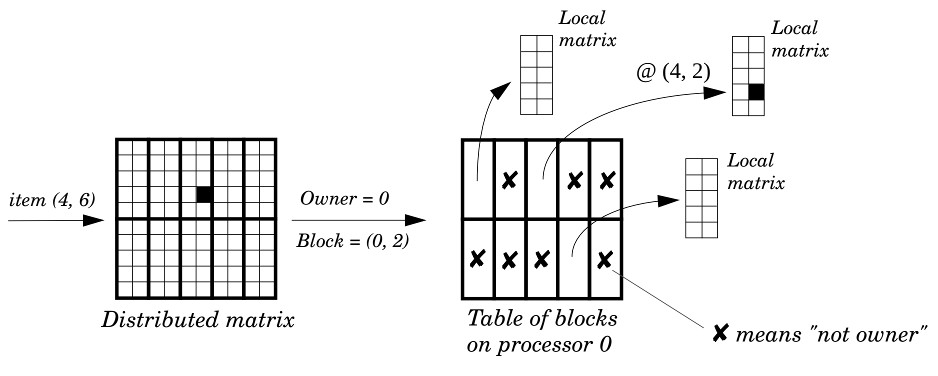


Figure 3: Internal representation of a matrix distributed by blocks

The class Dblock_Matrix (see figure 1) is one of the many possible descendants of Dist_Matrix. It inherits from Dist_Matrix as well as from Ar-

RAY2[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 3). 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.

The class hierarchy that results from our 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 matrices 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 OO 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.

3 Making Parallel Libraries Efficient

3.1 Optimization Techniques

The techniques presented in the previous section permit to achieve the transparent distribution of an aggregate: from the viewpoint of an application programmer, there is no fundamental difference between the services offered by a distributed aggregate and those offered by a local —sequential— one. It is only possible to handle larger aggregates due to the fact that the total memory offered by a DCS usually amounts to several times that of a mono-processor machine. However, the flow of control of all methods is still sequential, so that not much performance can be expected yet. To get better performances, it is necessary to redefine those operators of the distributed aggregate that can be considered as critical ones (that is, computation intensive ones). The method consists in using the facilities provided by the EPEE toolbox again, but this time the goal is to optimize the operators.

The optimization techniques involved at this level are rather classical: most of them result from the work led in the projects that aim at transforming sequential imperative languages like Fortran and C into parallel languages. These techniques are briefly enumerated below:

Discarding useless Refresh/Exec When redefining an operator, if it can be statically deduced from the distribution pattern of the aggregate considered that a data is locally present, this data can then be directly accessed. This optimization saves the useless tests and communications that are otherwise implied by the *Refresh* and *Exec* mechanisms. Likewise, when it can be locally decided that a remote data has not been updated between two remote readings, then a *Refresh* operation can be saved using a temporary variable.

Restricting iteration domains So far, each processor must mark every step of an iteration, although only those steps that have to do with local data should be considered. Operators can be redefined so as to mark in an iteration only those steps that have to do with local data. In the best case, the size of an iteration domain —and consequently the duration of its execution— can be divided by the number of processors implied in the computation.

Optimizing index transformations Accessing an element of a distributed aggregate is usually a costly operation, due to the many calculations that are needed to locate this element. Since the methods that compute index transformations can usually be encapsulated in a class (as shown in section 2.5 with the example of classes DISTRIBUTION_2D), it is possible to perform most of the index computations once and for all, and store the results in private tables. Index conversion tables are created and filled in when a new distributed aggregate is created. These tables make it possible to locate an item quickly. They do not need to be updated, unless the aggregate whose distribution pattern they describe is redistributed dynamically. With this optimization technique, the cost of an index conversion is only that of an indirection. Using index conversion tables thus reduces index conversion times, at the expense of the memory consumption.

Communication vectorization On most modern parallel architectures the cost of communications decreases every time a long message is preferred to many short ones. This observation led to a bunch of techniques for assembling several basic data elements in a single message (*e.g.*, direct communication, data coalescing, data aggregation, data vectorization). In EPEE, all these techniques can be used transparently, since data exchanges are encapsulated in classes and thus remain hidden to the user. We simply call *communication vectorization* the technique that consists in transmitting a large part of an aggregate instead of several smaller parts.

For example, the algorithm associated with method `trace` in class MATRIX (see example 2) is purely sequential and hence does not take into account the possible distribution of the matrix considered. We can redefine operator `trace` in class

DIST_MATRIX, providing the method with an algorithm that fits better the characteristics of a distributed matrix. The computation of the trace is now performed in two steps. In the first step, each processor computes locally a partial trace for the items it owns. The locality test reduces the iteration domain so that each processor only deals with local diagonal items. The second step of the algorithm is a classical SPMD reduction.

The parallelization of computations such as matrix multiply is more complex. The method `mult` takes two matrices as parameters and computes the product of these two matrices. It can thus be invoked to multiply two matrices `A` and `B` and assigning the result to a third matrix `C`, just by introducing the expression `C.mult(A, B)` in an application program. It was implemented quite simply in class MATRIX, based on three nested loops and accessors `put` and `item`. However, this default implementation is purely sequential, and therefore quite inefficient if at least one of `A`, `B` or `C` is distributed. We encapsulated in class DBLOCK_MATRIX another method `mult_dblock` that also computes the matrix product, but only if the three matrices implied in the computation are distributed by blocks.

Instead on relying on the basic scalar accessors `put` and `item`, the algorithm of `mult_dblock` is directly expressed in terms of block matrix operations. The iteration domain is reduced thanks to locality tests so that each processor only performs local calculations. Communications, when they cannot be avoided, are naturally vectorized due to the fact that the data exchanged are block matrices instead of scalar elements.

Actually, vectorizing data access is not profitable to communications only. Restructuring the algorithms encapsulated in class MATRIX and its descendants so that they perform block matrix operations in their inner loops globally enhances the performances of PALADIN, because it contributes to reduce memory swapping and cache defaults on most modern processors. Providing multi-level accessors to an aggregate therefore has an interesting consequence: it enhances the performances of operators thanks to a better exploitation of the memory architecture and thanks to a reduction of the cost of data exchanges between the processors of a DCS.

At this stage, it must be understood that this approach does *not* lead to an explosion of the number of methods in the library. Indeed, for each operation only a few number of distribution combinations can lead to an optimized parallel solver (*e.g.*, the matrix multiply above). If the object distributions do not fit them, one can either use the default (inherited) algorithm or (dynamically) *redistribute* one or several of the involved objects to select an efficient solver.

3.2 Preserving User Friendliness

With the optimization techniques discussed above, each operator declared in class MATRIX may be given several alternative implementations, each implementation being devoted to a particular representation or distribution pattern of the operands. It is obviously not desirable to have the application programmer specify explicitly which version of an operator is the most appropriate to mark every computation step in an application program.

The OO answer to this problem lies in the dynamic binding of methods to objects. Available in all OO languages, this mechanism ensures the transparency of method dispatching for parameterless methods such as method `trace`.

The problem is more complex for operators that admit several operands. Let us consider the method `mult` again. Remember that method `mult` was given a default sequential implementation in class MATRIX, but that DBLOCK_MATRIX encapsulates another method `mult_dblock` that is especially devoted to the computation of the product of two matrices distributed by blocks. It is actually most important to understand why we did not simply overwrite method `mult` in DBLOCK_MATRIX. Since it only permits to compute the product of matrices distributed by blocks, the algorithm of method `mult_dblock` cannot just replace that of method `mult`, which computes the product of any two matrices of compatible size, be they represented the same way or not. Because the service they offer to the application programmer is not exactly equivalent in both operators, `mult` and `mult_dblock` must remain available simultaneously in DBLOCK_MATRIX. Yet, this does not mean that the application programmer must be responsible for selecting which of these two operators is the most appropriate to compute the matrix product. Ideally, the selection of the most appropriate algorithm should be performed automatically at runtime based on the dynamic types of the operands. Such a mechanism of *multiple dispatching* is provided by some OO languages (*e.g.*, CLOS [Gabriel91] and Cecil [Chambers92]), but not by the Eiffel language neither by Modula-3, Ada 95 or C++ (whose *function overloading* mechanism is just syntactic sugar). Anyway, one can emulate multiple dispatching by performing explicit tests on the dynamic type of operands. The idea is to redefine method `mult` in class DBLOCK_MATRIX so as to test the dynamic type of the arguments `A` and `B` (see example 6). If the test shows that the dynamic types of `A` and `B` do not conform to type DBLOCK_MATRIX, then the default algorithm inherited from class MATRIX —and renamed as `mult_default` in the inheritance statement of class DBLOCK_MATRIX— is invoked. On the other hand, if the arguments are recognized as being matrices distributed by blocks, then the optimized parallel algorithm `mult_dblock` can be invoked. In both cases, the cost of the tests performed to select one or the other algorithm is negligible compared to the complexity of the algorithms that actually compute the matrix product. For the application programmer, however, the full transparency of the method dispatching is ensured.

There are actually other techniques to emulate multiple dispatching with a language such as Eiffel or C++ (*e.g.*, implementation of binary multi-methods based on a technique of reverse delegation), but as far as we know none of these techniques is fully satisfactory either. We are still looking forward to seeing an OO language

Example 6

```
class DBLOCK_MATRIX inherit
   DIST_MATRIX
      rename
        mult as mult_default  -- Keep the default sequential operator
      end
   ...
feature -- Optimized operators
   mult (A, B: MATRIX) is
      do
         if A.conforms_to ("DBLOCK_MATRIX")
            and B.conforms_to ("DBLOCK_MATRIX") then
            mult_dblock (A, B)
         else mult_default (A, B)
         end -- if
      end -- mult
   ...
end -- class DBLOCK_MATRIX
```

combining such features as static type checking[2], data encapsulation[3], and multiple dispatching.

3.3 Experimental Results

The OO approach leads naturally to the design of quite complex class hierarchies, and the question of their efficient implementation arises naturally. Efficiency is actually a crucial issue, since the main rationale for using DCSs lies in the exploitation of their ever growing computational power. If we were to end up with a parallel code running slower than the (best) equivalent sequential code, we would clearly have missed the point. Fortunately, it is not so.

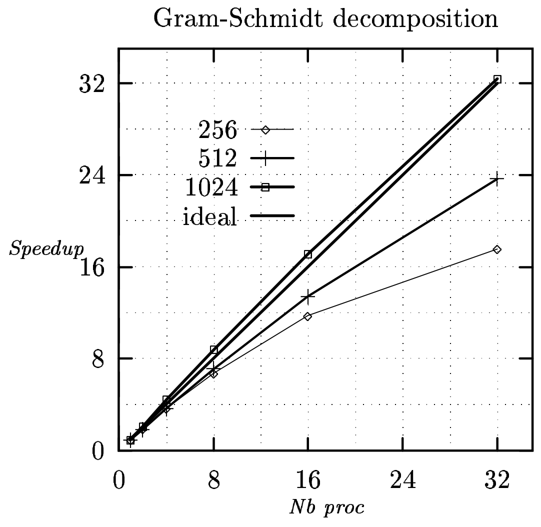
Figure 4 shows the speedups[4] obtained when performing parallel $Q.R$ decompositions of $N \times N$ dense square matrices distributed by columns, using a parallel implementation of the *Gram-Schmidt* algorithm on the Intel Paragon XP/S supercomputer.

The performances of PALADIN were significantly improved by interfacing some of its classes with the BLAS kernel [Lawson et al.89]. Thanks to this interfacing, BLAS routines are invoked whenever possible, that is, when the BLAS kernel is available on the target machine and when all the operands implied in an operation

[2]CLOS offers multiple dispatching but it is dynamically typed.

[3]Cecil implements multi-methods by partially breaking the encapsulation.

[4]We define the *speedup* as the execution time of a parallel Eiffel code on N nodes over the execution time of an equivalent sequential C code on one node.



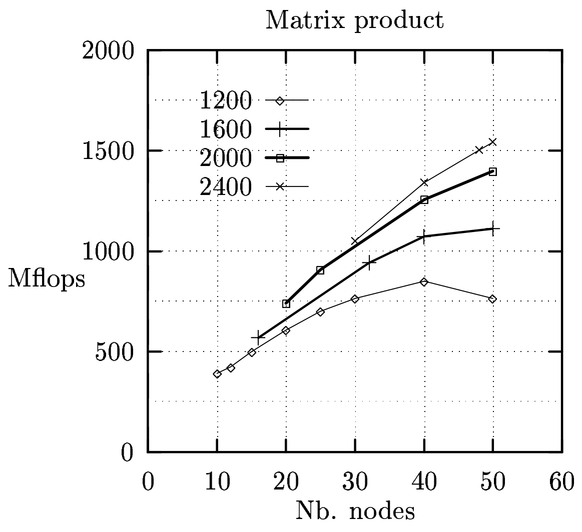
P	*time (sec)*	*speedup*	*efficiency*
	$N = 256$		
Ref.	84.17	—	—
1	87.53	.96	96 %
2	44.42	1.89	94 %
4	22.87	3.68	92 %
8	12.45	6.76	84 %
16	7.16	11.75	73 %
32	4.79	17.57	54 %
	$N = 512$		
Ref.	654.21	—	—
1	698.09	.93	93 %
2	357.12	1.83	91 %
4	178.27	3.66	91 %
8	91.95	7.11	88 %
16	48.78	13.41	83 %
32	27.61	23.69	74 %
	$N = 1024$		
Ref.	6358.98	—	—
1	6692.34	.95	95 %
2	2998.96	2.12	106 %
4	1426.17	4.45	111 %
8	722.79	8.79	109 %
16	371.80	17.10	106 %
32	196.50	32.36	101 %

Figure 4: Gram-Schmidt decomposition of NxN dense matrices distributed by columns

have a Fortran-compatible representation. For example, whenever the product of two instances of LOCAL_MATRIX must be computed, the BLAS subroutine `GEMM` (GEneral Matrix Multiply) is invoked transparently to perform the computation instead of the default Eiffel routine defined in class MATRIX. Figure 5 shows the performances observed when computing a double precision matrix-matrix product (with matrices distributed by blocks) on the Paragon XP/S. Despite the use of a high-level OO language — and thanks to the use of the BLAS kernel to perform the inner block matrix products —, we get the best performances out of the Paragon XP/S (around 1.5 Gflops on 50 nodes).

4 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



P	*time (sec)*	*Mflops*	*Mflops/node*
		$N = 1200$	
10	8.83	391.39	39.13
20	5.69	607.38	30.36
30	4.53	762.91	25.43
40	4.07	849.14	21.22
50	4.53	762.91	15.25
		$N = 1600$	
16	14.41	568.49	35.53
32	8.69	942.69	29.45
40	7.64	1072.25	26.80
50	7.36	1113.04	22.26
		$N = 2000$	
20	21.56	742.12	37.11
25	17.63	907.54	36.30
40	12.72	1257.86	31.44
50	11.45	1397.37	27.94
		$N = 2400$	
30	26.31	1050.85	35.02
40	20.61	1341.49	33.54
48	18.39	1503.42	31.32
50	17.92	1542.85	30.85

Figure 5: Matrix multiply of NxN dense matrices distributed by blocks

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. By interfacing the internals of PALADIN with the BLAS kernel, we also demonstrated that the OO approach makes it possible to benefit from highly optimized machine code libraries and provide the best performances without sacrificing conceptual high level of the user's point of view. 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.

Acknowledgments The work on the Paragon XP/S described in this paper is supported by Intel SSD under an External Research and Development Program (INRIA contract no. 193C21431318012).

References

[Abelson et al.85]	Abelson (H.), Jay Sussman (G.) and Sussman (J.). – *Structure and Interpretation of Computer Programs.* – MIT Press, Mac Graw Hill Book Company, 1985.
[America87]	America (P.). – Pool-T: A parallel object-oriented programming. *In: Object-Oriented Concurrent Programming*, éd. par Yonezawa (A.). pp. 199–220. – The MIT Press, 1987.
[Andre et al.90]	André (F.), Pazat (J.L.) and Thomas (H.). – Pandore: a system to manage data distribution. *In: ACM International Conference on Supercomputing.* – June 11-15 1990.
[Callahan et al.88]	Callahan (D.) and Kennedy (K.). – Compiling programs for distributed-memory multiprocessors. *The Journal of Supercomputing*, vol. 2, 1988, pp. 151–169.
[Cardelli et al.85]	Cardelli (L.) and Wegner (P.). – On understanding types, data abstraction, and polymorphism. *ACM Computing Surveys*, vol. 17 (4), 1985, pp. 211–221.
[Chambers92]	Chambers (C.). – Object-Oriented Multi-Methods in Cecil. *In: Proceedings of the European Conference on Object-Oriented Programming (ECOOP'92).* – 1992.
[Fahringer et al.92]	Fahringer (T.), Blasko (R.) and Zima (H. P.). – Automatic Performance Prediction to Support Parallelization of Fortran Programs for Massively Parallel Systems. *In: 6th ACM International Conference on Supercomputing*, pp. 347–356. – Washington, D.C., July 1992.
[Gabriel91]	Gabriel (R. et al.). – CLOS: Integrating Object-Oriented and Functional Programming. *Communications of the ACM*, vol. 34 (9), 1991.

[Guidec et al.95] Guidec (F.) and Mahéo (Y.). – POM: a Virtual Parallel Machine Featuring Observation Mechanisms. *In : Proc. of the International Conference on High Performance Computing, New Delhi, India.* – December, 27-30 1995.

[Hamelin et al.94] Hamelin (F.), Jézéquel (J.-M.) and Priol (T.). – A Multi-paradigm Object Oriented Parallel Environment. *In : Int. Parallel Processing Symposium IPPS'94 proceedings*, éd. par Siegel (H. J.). pp. 182–186. – IEEE Computer Society Press, April 1994.

[Hpf forum93] HPF-Forum. – *High Performance Fortran Language Specification.* – Technical Report Version 1.0, Rice University, May 1993.

[Jezequel93] Jézéquel (J.-M.). – EPEE: an Eiffel environment to program distributed memory parallel computers. *Journal of Object Oriented Programming*, vol. 6 (2), May 1993, pp. 48–54.

[Kessler94] Kessler (C.W.). – Symbolic Array Data Flow Analysis and Pattern Recognition in Dense Matrix Computations. *In : Proceedings of the Working Conference on Programming Environments for Massively Parallel Distributed Systems.* – IFIP WG 10.3, April 1994.

[Lawson et al.89] Lawson (C.), Hanson (R.), Kincaid (D.) and Krogh (F.). – Basic Linear Algebra Subprograms for Fortran. *ACM Transactions on Math. Software*, vol. 14, 1989, pp. 308–325.

[Meyer88] Meyer (B.). – *Object-Oriented Software Construction.* – Prentice-Hall, 1988.

[Pancake et al.90] Pancake (C.) and Bergmark (D.). – Do parallel languages respond to the needs of scientific programmers? *IEEE COMPUTER*, December 1990, pp. 13–23.

[Valiant90] Valiant (Leslie G.). – A bridging model for parallel computation. *CACM*, vol. 33 (8), Aug 1990.

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

[Zima et al.88] Zima (Hans P.), Bast (Heinz-J.) and Gerndt (Michael). – SUPERB: a Tool for Semi-Automatic MIMD/SIMD Parallelization. *Parallel Computing*, no6, 1988, pp. 1–18.

Jean-Marc JÉZÉQUEL received an engineering degree in Telecommunications from the ENSTB in 1986, and a Ph.D. degree in Computer Science from the University of Rennes, France, in 1989. He is a research manager in the Irisa Lab for the Centre National de la Recherche Scientifique. His research interests include software engineering and object oriented technology for telecommunications and distributed computers. He is the author of the book *Object Oriented Software Engineering with Eiffel*, Addison-Wesley, Feb. 1996.

Frédéric GUIDEC received his Ph.D. degree in Computer Science from the University of Rennes, France, in 1995. His research interests include the design of programming environments for distributed computing systems in a context of software engineering, using object-oriented techniques. He is now a Research Assistant at the Swiss Federal Institute of Technology (EPFL, Lausanne, Switzerland).

Jean-Lin PACHERIE received his DEA (Master) in Computer Science from the University of Grenoble, France, in 1994. He is now pursuing a PhD in Computer Science in IRISA (Rennes I, France). His research interests are focused on the design of reusable parallel software components for massively parallel architectures, with emphasis on data parallelism with a Distributed Shared Memory.

Figure 1: Inheritance structure for matrix aggregates (partial view)

```
local
   my_dist: DISTRIBUTION_2D;
   my_mapping: COLUMN_WISE_MAPPING;
do
   ...
   !!my_dist.make (10, 10, 5, 2, my_mapping);
   ...
```

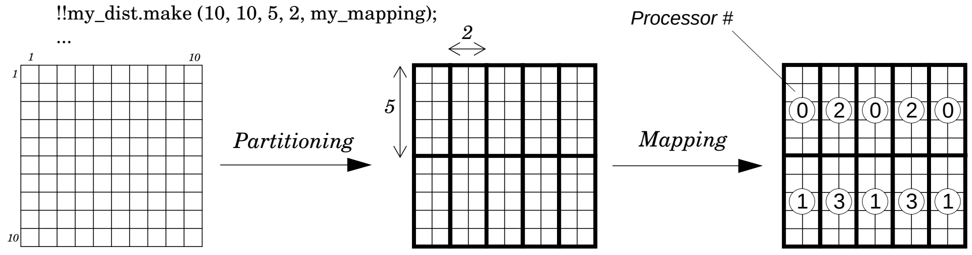


Figure 2: Example of a distribution allowed by class DISTRIBUTION_2D.

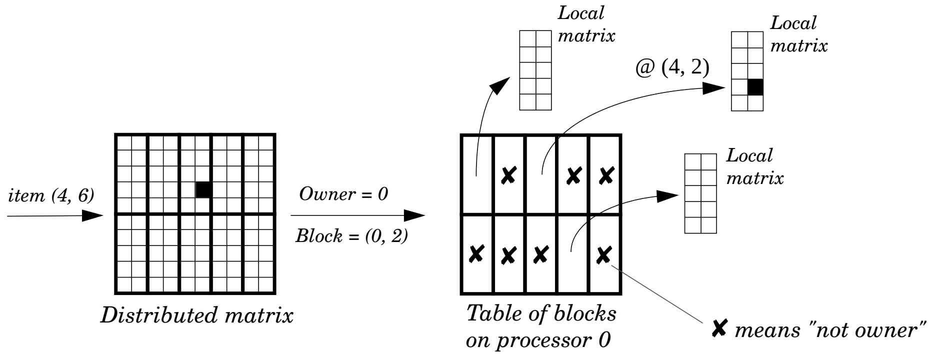
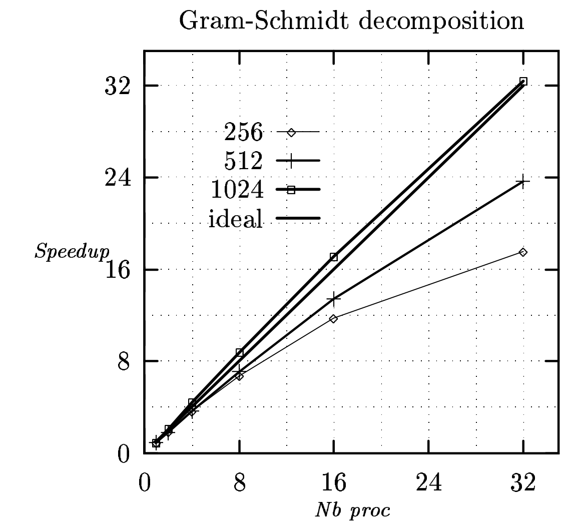
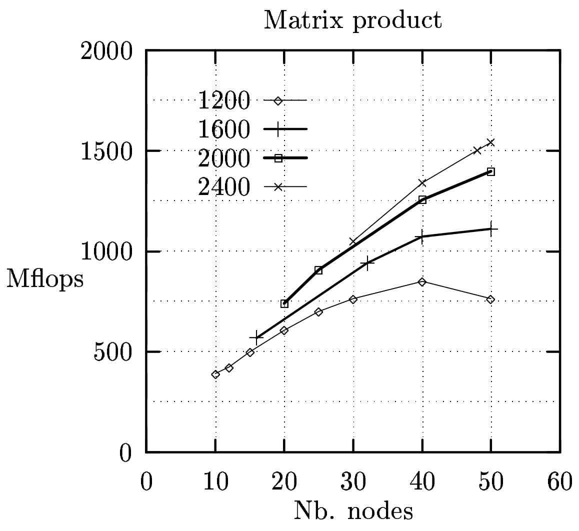


Figure 3: Internal representation of a matrix distributed by blocks



P	*time (sec)*	*speedup*	*efficiency*
	$N = 256$		
Ref.	84.17	—	—
1	87.53	.96	96 %
2	44.42	1.89	94 %
4	22.87	3.68	92 %
8	12.45	6.76	84 %
16	7.16	11.75	73 %
32	4.79	17.57	54 %
	$N = 512$		
Ref.	654.21	—	—
1	698.09	.93	93 %
2	357.12	1.83	91 %
4	178.27	3.66	91 %
8	91.95	7.11	88 %
16	48.78	13.41	83 %
32	27.61	23.69	74 %
	$N = 1024$		
Ref.	6358.98	—	—
1	6692.34	.95	95 %
2	2998.96	2.12	106 %
4	1426.17	4.45	111 %
8	722.79	8.79	109 %
16	371.80	17.10	106 %
32	196.50	32.36	101 %

Figure 4: Gram-Schmidt decomposition of NxN dense matrices distributed by columns



P	*time (sec)*	*Mflops*	*Mflops/node*
	$N = 1200$		
10	8.83	391.39	39.13
20	5.69	607.38	30.36
30	4.53	762.91	25.43
40	4.07	849.14	21.22
50	4.53	762.91	15.25
	$N = 1600$		
16	14.41	568.49	35.53
32	8.69	942.69	29.45
40	7.64	1072.25	26.80
50	7.36	1113.04	22.26
	$N = 2000$		
20	21.56	742.12	37.11
25	17.63	907.54	36.30
40	12.72	1257.86	31.44
50	11.45	1397.37	27.94
	$N = 2400$		
30	26.31	1050.85	35.02
40	20.61	1341.49	33.54
48	18.39	1503.42	31.32
50	17.92	1542.85	30.85

Figure 5: Matrix multiply of NxN dense matrices distributed by blocks