EPEE: an Eiffel Environment to Program Distributed Memory Parallel Computers

J.-M. JÉZÉQUEL

I.R.I.S.A. Campus de Beaulieu
F-35042 RENNES CEDEX, FRANCE
E-mail: jezequel@irisa.fr
Tel: +33 99 84 71 92 ; Fax: +33 99 38 38 32

Abstract. Most parallel object oriented languages (OOL) are currently using a general parallelism model based on communicating sequential processes. This approach makes it difficult to program massively parallel systems in an easy and efficient way. So we propose to use another form of parallelism, known as data parallelism. We describe how a pure sequential OOL can embed data parallelism in a clean and elegant fashion to exploit the potential power of massively parallel systems. Then we present EPEE (an Eiffel Parallel Execution Environment) at work with a well known parallel paradigm (matrix computations), along with experimental performance results. We draw some conclusions on the generality of this approach.

Keywords: Massively Parallel Architectures, Data Parallelism, Parallel Object Oriented Programming, Eiffel

1 Introduction

It is now widely agreed that the road leading to TeraFlops machines (*i.e.* machines performing 10^{12} floating point operations per second) must go by massively parallel architectures. Only modular and distributed memory multi-processor architectures can be scaled up —theoretically— at will. Precursors of these architectures already exist today, as illustrated for example by the Intel iPSC hypercubes or the transputer based machines. More powerful ones are about to come as a result of research projects led in Europe (GP-MIMD machines), in America (Touchstone project) or in Japan.

But these Distributed Memory Parallel Computers (DMPC) presently suffer from a lack of programming environments: the user must have a wide experience in parallel programming and a good understanding on his machine architecture and operating system. This is probably the main factor hampering their wider diffusion. So, it is our strong feeling that the Object Oriented powerful concepts should be made available to program DMPC.

Parallelism is often introduced in OOL through the idea that objects can be made active, *i.e.* can be seen like concurrent activities communicating by sending messages: the object oriented message passing paradigm is mapped onto the communication structure of a parallel system. In that case, object creation can involve a process creation. Such notion of parallelism is implemented in POOL-T [1], where an object may have a "body" (kind of background activity). Communications between objects

is based on the Remote Procedure Call paradigm, and the POOL-T user has to control explicitly the concurrency structure.

Another way to introduce parallelism is by mean of asynchronous operation calls: an object calling a method of another object may continue its activities in parallel with the object executing the called operation. This is implemented for instance in ABCL/1 [15], ELLIE [2], ConcurrentSmalltalk [14].

Both ideas of active object and asynchronous operation calls are used to make OOL parallel either by integrating this parallelism directly into the design of a new concurrent programming language, or simply by extending existing sequential languages to handle parallelism. The former approach leads to clear and unified support of conceptual models. ELLIE, POOL-T, ABCL/1, etc. are OOL where a significant number of structures are devoted to the deal with parallelism. Using the latter approach, we can find for example DistributedSmalltalk [4] which extends Smalltalk or COOL [6] which extends C++.

So, near all parallel object oriented languages are based on a MIMD programming model, which seems to match rather well some kind of problems (operating systems, industrial process control...). This model of parallelism is typically dedicated to handle functional parallelism: a given function is divided in some sub-functions that can be processed by new computational threads. So the definition of processes is determined by the subtask decomposition, and as this is application dependent, it necessitates a strong involvement from the user. Furthermore, the processes are of an heterogeneous nature, leading to difficult load balancing problems.

One of the most difficult obstacle to be worked around with this approach is to ensure that well known problems in the protocol engineering community —such as deadlocks, livelocks, unspecified receptions and so on— are properly dealt with. Because the cooperation between processes is explicitly coded, it is usually difficult to manage the overall communication structure.

When programming DMPC that way, we must face another major issue: efficiency. We have to remember that getting scalable performances is the main reason for using DMPC with large number of processors. However functional parallelism usually lacks of compile-time defined process and communication structures. Few regular structures may be derived from the original program at compile time, and most of the work must be done at runtime. But the dynamic nature of this process definition implies the use of general and costly mechanisms such as objects naming servers, object migration, and general purpose operating system functions to implement load balancing, scheduling, etc... Above all, this kind of parallelism suffers from a lack of scalability: the subtasks decomposition does not allow an efficient handling of the very high number of processors available on future DMPC.

To override these problems we propose to use a different model of parallelism, known as data parallelism. This approach seems to be rather natural in an OO context, as object oriented programming usually focuses on data rather than on functions. Some principles to build an OOL aimed at data parallelism have already been introduced in [10], and basic ideas to extend C++ in the same way are presented in [7]. With EPEE we go one step beyond: data distribution and parallelism is totally embedded in an OOL using only already existing language constructions. EPEE is based on Eiffel [11] because it features all the concepts that we need, using a clear syntax and semantic. However, our approach is not strongly dependent on Eiffel; it

could be implemented in any OOL featuring strong encapsulation (and static type checking), multiple inheritance, dynamic binding and some form of genericity.

We show in section 2 that these object oriented concepts are powerful enough to embed data parallelism in an OOL in a clean and elegant fashion, without modifying the OOL syntax and semantic. We present in section 3 our data distribution and parallelization rules through a well known example, from specification and analysis to implementation and experimental results. In the last part, we draw some conclusions on our approach and present some perspectives.

2 Embedding data parallelism in an OOL

2.1 A programming model to exploit data parallelism

Programming models other than the message passing MIMD model have been investigated for programming DMPC. For instance the SIMD (Single Instruction Multiple Data) model is sometimes a natural way to describe algorithms; and special SIMD computers (like the Connection Machine) have been built to implement it efficiently.

A lot of work is also carried on full automatic parallelizing compilers: here the programming model is the well know SISD (Single Instruction Single Data). However, this approach does not seems to work well for DMPC, as it is well known that speed-up above 10 are rarely seen for parallelization of dusty-deck FORTRAN programs.

The SPMD (Single Program, Multiple Data) is a kind of a compromise between both approaches: it brings the conceptual simplicity of the SISD model and the parallelism of the SIMD one. This model exploits the fact that most of the problems running on DMPC involve large amount of data. The set of data may be split into partitions, to which processes are associated statically. Each process executes the same program (corresponding to the initial user-defined sequential program) on its own data partition. So the user view of a program remains a sequential one: a sequence of actions is applied to a set of data. The parallelism is automatically derived from the data decomposition, leading to a regular and scalable form of parallelism.

Data parallelism and SPMD basic principles are described in [5], and SUPERB [16] or PANDORE [3] are examples of compilers aiming at translating classical imperative languages like FORTRAN and C towards parallel distributed code for DMPC. However the user is still required to define (at least) its data partitions, thus changing the original program either interactively or through the use of compiler directives. Furthermore, this kind of compiler must still have a wide expertise on algorithmic parallelization rules for the distributed data structures (actually only arrays) on which it works.

We propose to conceptually *remove* this expertise from the compiler core to encapsulate it with the data structure on which it is defined. We show in the following that an OOL makes it possible to achieve this encapsulation to leave the illusion of a sequential programming to the user (*i.e.* the application programmer).

2.2 Encapsulating Parallelism within Classes

Various research projects have attempted at using object oriented programming encapsulation and information hiding features to encapsulate parallelism, for example

extending C++ with so called *path expressions* [13]. However, we want to go beyond that, actually completely hiding the parallelism to the user. Thus we no longer want to map the object oriented message passing paradigm onto actual interprocess communications. Our approach at encapsulating parallelism is much more like encapsulating tricky pointer manipulations within a linked list class, thus providing the abstraction *list* without annoying the user with pointer related notions.

As it is explained above, we are interested in parallelizing classes aggregating large amounts of data. For such a class C, we want to provided a parallel implementation that we call *Distributed_C*, having exactly the same (sequential) interface as C. In the object oriented framework, this can be done using the inheritance concept: one can say that *Distributed_C* inherit from C, as it is a specialization of C.

As our distributed classes will share a lot of features, we factor out and materialize this sharing through an ***had hoc*** class that we call *Distributed_Aggregate*. Then we can make *Distributed_C* inherit from both C and *Distributed_Aggregate*.

The EPEE *Distributed_Aggregate* is not unrelated to the notion proposed in [8]: it is an aggregate of generic data that is spread across a DMPC, together with a set of methods to access its data transparently, to redistribute it, to perform a method on each of its elements, and to compute any associative function on the aggregate. The way these methods are actually implemented depends on the underlying architecture: to preserve full portability at the OOL level, we introduce a system dependent interface module between a Massive Parallel Architecture (MPA) and the *Distributed_Aggregate* class, as illustrated in figure 1.

For common DMPC (set of processors communicating only by FIFO message passing), here is an outline of a possible implementation of this concept. As a given node only owns a subset of the distributed aggregate data, we have first to use a mechanism allowing the transparent remote access to non local data, while preserving the semantic of local accesses. This mechanism is based on the Refresh/Exec notion (described in [3]): a data is *Refresh*ed before any reading attempt, and writing is based on a local write principle *(Exec)*, *i.e.* a data is updated only on the processor owning it. On such DMPC, a possible implementation of the Refresh is:

```
If Owner(V) = this_node
  then broadcast (V) and return (Value of V)
  else receive_from (Owner(V),V) and return (Value of V)
```

To implement redistribution and circulation of data, we can use the underlying FIFO message passing mechanism. For "global" methods, we can either implement on the DMPC well known relevant algorithms or use already existing global operation facilities when they are built-in, as in the iPSC computers.

2.3 A Methodology to use EPEE

We distinguish two levels of programming in EPEE: the class user (or *client*) level and the parallelized class designer level. Our aim is that at the client level, nothing but performance improvements appears when running an application program on a DMPC. We would like this performance improvement to be proportional to the number of processors of the DMPC (linear speed-up), which would guarantee scalability.

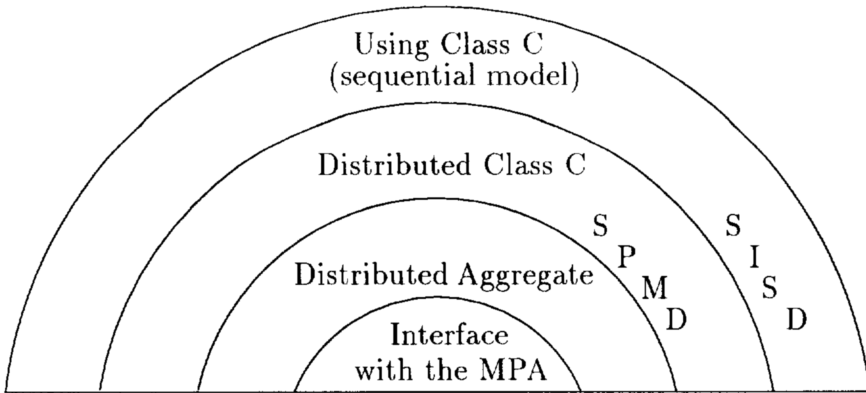

Fig. 1. EPEE: programming a DMPC at different levels

The designer of a parallelized class has the responsibility to implement general enough data distribution and parallelization rules, ensuring portability, efficiency and above all scalability, while preserving a "sequential-like" interface for the user. If a class already has a specification —and/or a sequential implementation— the parallel implementation would have the same semantic: each parallelized method would leave an object in the same abstract state that the corresponding sequential one.

To implement that, a designer would select interesting classes to be data parallelized, *i.e.* classes aggregating large amounts of data, as for example classes built on *arrays, sets, trees, lists*... Then, for each such class, one or more distribution policies are to be chosen and data access methods redefined accordingly, using the abstractions provided in the EPEE distributed aggregate class. Finally, critical methods (in terms of efficiency) can be redefined, using the EPEE SPMD programming model to take advantage of a given data distribution. Here again, the general methods defined in the distributed aggregate class are to be used to hide the underlying system (see figure 1).

In the next section, we apply this methodology to a well known example: linear algebra matrix computations.

3 Using EPEE for matrix computations

3.1 Problem specification and analysis

Parallelization of linear algebra computations is a well known and well understood parallel programming paradigm. We present EPEE at work on this example to let the reader concentrate himself more on the approach than on the peculiarities of the example. However, the EPEE interest lies in that it is not limited to full matrix computations.

Let the Matrix class be a class encapsulating the abstract data type matrix of real, with such operations as reading, addition, multiplication and Inversion (see an

```
class interface MATRIX exported features
        read, infix "+", infix "*", inversion, nb_rows, nb_columns, item, put
feature specification
        Create (nb_rows, nb_columns: INTEGER)
                require
                        not_flat: nb_rows > 0; not_thin: nb_columns > 0
        Read            -- Read Matrix elements
        infix "+" (m: like Current): like Current
                require
                        m_must_exist: not m.Void ;
                        compatible: nb_rows=m.nb_rows
                                      and nb_columns=m.nb_columns
        infix "*" (m: like Current): like Current
                require
                        m_must_exist: not m.Void ;
                        compatible: nb_columns=m.nb_rows
        Inversion       -- compute Result such that Result*Current=Identity
                require
                        square: nb_rows=nb_columns
end interface -- class MATRIX
```

Fig. 2. A Matrix class in Eiffel

Eiffel specification on figure 2). These operations have a sequential implementation, and a client class could use the Matrix class as follows:

```
Class client
feature Create is
    local M, A, B, C: Matrix
    do
        A.Create (512, 512); B.Create (512, 320); C.Create (320, 512);
        A.read; B.read; C.read;
        M := A + B * C;
        M.inversion;
        M.print;
    end;
end;
```

According to the the methodology described above, let us start by analyzing this kind of program in some details to find out means to improve its performances when run on a DMPC.

The first line contains three *creation* instructions, each one of 0(1) complexity (wrt. the size n of the matrices). Clearly, these three instructions are fully independent. Thus they could be run in parallel. This kind of parallelism is exploited in various parallelization tools, but in EPEE we are not interested in it for two kind of reasons. First, nothing proves that the execution in parallel of these three instructions on a real DMPC is faster than a sequential one, because of communication, synchronization and sub-task creation overheads. Then, this kind of parallelism is

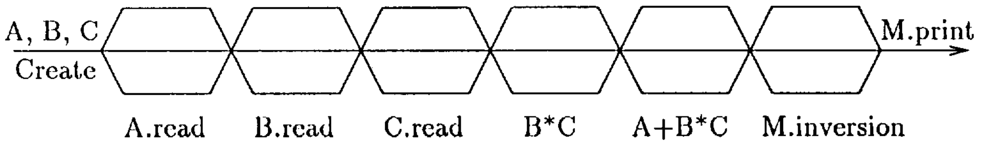

Fig. 3. SPMD execution on 3 processors of our Class Client example

not scalable, neither in terms of the matrix size nor in terms of the processor number of the DMPC.

The same arguments hold for the second program line (matrix reading). However here, each instruction has a $0(n^2)$ complexity. So it would be interesting to parallelize them just by making each processor read its part of the data only. If p is the processor number of the DMPC, then a speed-up of p is theoretically achievable —the DMPC must however feature concurrent I/O: this is the case when a CFS module is available for an Intel iPSC hypercube for example. Clearly, this parallelism is scalable.

The third line of the program contains two operations (matrix addition and multiplication). It is well known that parallel algorithms bringing a theoretical speed-up of p exist for both, so these two operations can be parallelized in a scalable way. One the following line, matrix inversion can be also parallelized in a scalable way. Finally, writing the result on a sequential device (printer, file, screen, etc.) can be parallelized only if special hardware is available.

Not every program can be parallelized in a scalable fashion, *i.e.* its parallelization does not bring significant performance improvement. To formalize that, Valiant proposed in [12] the BSP model (Block Synchronous Parallel). A computation fits the BSP model if it can be seen as a succession of parallel phases separated by synchronization barriers and sequential phases. In figure 3, we represented the potential data parallelism existing in our Class Client example, for three processors: our example perfectly fits this scalable BSP model.

In this model, a computation can be efficiently parallelized only if the cost for synchronizations, communications and other processing paid for managing parallelism is compensated by the performance improvement brought by the parallelization. So we will get speed-up above 1 only if the size of the matrices is large enough, and our speed-up will increase with the ratio computation time vs. communication time.

Having described the kind of parallelism we are interested in, we now show how to implement it on a massive parallel system with satisfying performances.

3.2 Data distribution in the Matrix class

From our example Matrix class we want to build a distributed Matrix class having both the Matrix class interface and a distributed implementation. The multiple inheritance feature found in Eiffel makes it possible to express that a distributed Matrix is both a Matrix and a distributed aggregate:

```
Class Distributed Matrix
    export repeat Matrix -- Same interface as Matrix
    inherit Distributed_Aggregate; Matrix -- inherit from both ancestors
```

Then the Distributed Matrix constructor (Create feature in Eiffel) can be defined as to split the Matrix on the various nodes of the DMPC, using the Distributed Aggregate features. As we work in a SPMD model, each processor executes the creation instruction, but actually creates only its own part of the Matrix.

Because of Eiffel strict encapsulation feature, the only way Matrix element are accessed is through the methods:

- $item(i, j)$, which returns the value of the Matrix element found at line i and column j
- $put(value, i, j)$, which stores a value at line i and column j.

Thus we only have to redefine these two methods to make them work with our data distribution. Using the local write principle, we redefine *put* as to store the required value only if the node executing it owns the relevant Matrix cell. The refresh instruction allows the remote reading of a value, so we redefine *item* to make use of it, along with various other features inherited from Distributed Aggregate, as for example index transformation features. Thus, our Distributed Matrix class becomes:

```
Class Distributed Matrix
    export repeat Matrix -- Same interface as Matrix
    inherit Distributed_Aggregate; Matrix -- inherit from both ancestors
            rename item as local_item, put as local_put;
            redefine item, put
feature
    item (row, column: integer): real is
        do Result := Refresh (row, column) end;
    put (v: like item; row, column: integer) is
        do if Exec (row, column)
             then local_put (v, absolute_to_local_index(row),column) end;
        end
end;
```

In so far, we have a Distributed Matrix class having the same semantic as the Matrix class, *i.e.* they have the same interface and the same behavior. When for example a Distributed Matrix is multiplied to another one, it is the Matrix class multiplication method that is invoked, but with the redefined versions of *item* and *put* methods.

The only behavior differences come from the environment. First, as a Distributed Matrix is split across the DMPC, each node owns only a fragment of it: this makes it possible to process larger matrices. However, as the access mechanism defined for the Distributed Matrix class is more expansive (even without considering dynamic binding, the Refresh/Exec mechanism involves extra tests and message exchanges) than the basic one, the overall computation takes more time.

To get better results, we can carrying on redefining relevant methods of the Distributed Matrix class. We are still using the Distributed Aggregate features, but now for optimization purposes.

Some of these optimizations are rather classical: there are based on the work led in the PANDORE [3] project. The originality of our approach is to present them in an object oriented framework, which eventually allows the prototyping of new ideas.

3.3 Optimizing the basic SPMD model

Discarding useless Refresh/Exec When redefining a method, if we can statically know (*i.e* from the data distribution rules) that an aggregate element is locally present, this method can directly access this element through the *put* and *item* methods of the original Matrix class. These methods are still available, being renamed *local-put* and *local-item*. This optimization saves the Refresh/Exec extra tests and useless extra communications. In the same way, when it is statically known that a data does not change between two remote readings, a Refresh operation can be saved using a temporary variable.

Restricting iteration domains Until now, each node does every step of an iteration, whereas only computations modifying local data are useful. So we can redefine methods to perform only relevant parts of iterations, *i.e* only those steps that modify local data. In the best case, we can divide an iteration size (and its execution time) by the DMPC number of nodes. Using these two first optimization ideas, we can for example redefine our distributed Matrix addition method as shown in figure 4.

```
redefine infix "+";
infix "+" (m: like Current): like Current
    require
        m_must_exist: not m.Void ;
        compatible: nb_rows=m.nb_rows and nb_columns=m.nb_columns
    local
        i,j : INTEGER;
    do
        result.Create(nb_rows,nb_columns);
        from i := 1 until i > local_nb_rows --Restricting iteration domain
        loop
            from j := 1 until j > nb_columns
            loop
                result.local_put(local_item(i,j) + m.local_item(i,j),i,j);
                j := j+1
            end;
            i := i+1
        end;
end; -- infix "+"
```

Fig. 4. Adding two distributed matrices in Eiffel

Optimizing index transformations Accessing a distributed aggregate element is a costly operation, which depends on the actual distribution policy. However, as index transformation methods (local-to-absolute-index and converse, owner-of-index and so on) are encapsulated features of the Distributed Aggregate class, we can redefine that in Distributed Matrix without consequences. We can for example perform only once the index computations and store the results in private tables, at the expense of some memory consumption. These tables should be updated at Distributed Matrix creation time and whenever the actual distribution changes. With this optimization, the index transformation cost is only that of an indirection.

Dynamic redistribution As our Distributed Aggregate class has a built-in method (called "circulate") to make its data pieces locally available one after another on each node, we can redefine Distributed Matrix methods that need access to all the data using this efficient feature. This optimization can be use for example to implement the Distributed Matrix multiplication method: one matrix is left motionless while the other one is "circulated" on a virtual ring of processors. So its rows are made locally available one after another on each node, thus allowing the computation of relevant pieces of the result matrix (see figure 5).

3.4 EPEE implementation

EPEE (Eiffel Parallel Execution Environment) is actually made of three main parts:

- the Distributed Aggregate Class, which is a normal Eiffel class making heavy usage of external *C* functions calls. This class must be inherited by each Eiffel class willing to take advantage of parallelism and distribution features within EPEE.
- a set of interface modules (written in *C*), built on top of the ECHIDNA experimentation environment [9] to provide an homogeneous and instrumented interface to the Distributed Aggregate Class. At present, modules for Intel iPSC/2, iPSC/860 and Sun networks are available. Other are currently being developed for Transputer based architectures.
- a set of tools to facilitate cross-compilation and distributed experimentation of an Eiffel program in EPEE.

We experimented our ideas through prototype Matrix and Distributed Matrix class implementations, using the ISE Eiffel compiler (V.2.3) which allows the production of portable *C* packages.

It is worth insisting that with EPEE, a pure Eiffel program is compiled, with the full sequential semantic of Eiffel. However, we currently have a limitation: we do not handle non-fatal exceptions properly, so the Eiffel *rescue* mechanism cannot be used. The only other visible difference when executed on a DMPC is an increasing of performances.

Differing from most MIMD OOL, our objet migrations are fully encapsulated in relevant classes: an objet cannot really "move", it can just be redistributed. Thus we can get rid of a global naming system, which is usually very costly both in communications and synchronizations. Furthermore, as every object exists on every

```
redefine infix "*";
infix "*" (m: like Current): like Current
    require
        m_must_exist: not m.Void ;
        compatible: nb_columns=m.nb_rows
    local
        p,i,j,k : INTEGER;
    do
        Result.Create(nb_rows,m.nb_columns);

        from p := 1 until p > number_of_nodes loop
           from k := 1 until k > m.local_nb_rows loop
              from i := 1 until i > Result.local_nb_rows loop
                 from j := 1 until j > Result.nb_columns loop
                    Result.local_put(Result.local_item(i,j)
                      + local_item(i,m.local_to_absolute_index(mynode,r,k))
                      * m.local_item(k,j),
                     i,j);
                    j := j+1
                 end;
                 i := i+1
              end;
              k := k +1
           end;
           m.circulate(p); -- shift m data already processed to neighbor
           p := p + 1
        end;
end; -- infix "*"
```

Fig. 5. Multiplying two distributed matrices in Eiffel

node (be it duplicated, this is the default, or split across the DMPC if it inherits from Distributed Aggregate), the standard Eiffel garbage collector performs correctly, thus avoiding the need for a costly distributed garbage collector. By the way, we can also make use of the *co-routine* aspect of the ISE Eiffel garbage collector to make it run whenever a node is blocked awaiting a reception.

One of the most difficult problem to be solved in a distributed environment concerns distributed debugging. Only low-level debugging features are usually available on DMPC, and the user must spend great efforts trying to figure out what is going on within his DMPC, as even inserting some trace commands may change drastically his program behavior. With EPEE, one can use the standard Eiffel environment within various windows —each one representing a node— on a single workstation to incrementally develop and debug the SPMD classes (in our example Distributed Matrix). Our parallelism model being fully scalable and encapsulated within the Distributed Aggregate class, we know our applications run the very same way and without any modification whatever the underlying DMPC. This has been confirmed by our experiments.

3.5 Experimental Results

We led some performance measurements of our implementation on an Intel hypercube iPSC/2 with 32 processors. For various cube sizes, we measured execution times of +, × and *Inversion* methods of class Distributed Matrix, along with the total execution time of the client program described in the paragraph 3.1. We compared these results to their best sequential counterparts (i.e from class Matrix) run on one iPSC/2 node, thus allowing a speed-up evaluation (see figures 6, 7, 8 and 9).

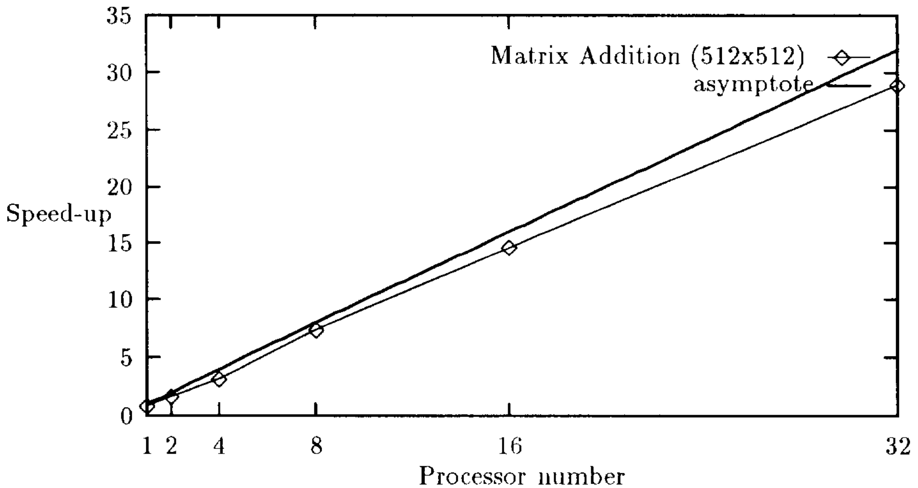

Fig. 6. Matrix Addition: speed-up

In each case, speed-ups are near proportional to the number of nodes used, thus confirming the interest of our approach for programming massively parallel systems. The performance loss between the execution of class Matrix methods and the corresponding Distributed Matrix methods run on only one node can be explained by the overhead of the dynamic binding of *put* and *item* methods. Actually, when a feature is never redefined —as *item* and *put* in the Matrix Class— the ISE Compiler discards the general dynamic binding mechanism and produces a normal procedure call. As we redefine these features in Distributed Matrix, we have to pay the price for dynamic binding. In a lesser extent, the overhead of the index manipulation (double indirection) is also costly. We think that a *cache* mechanism —as implemented in

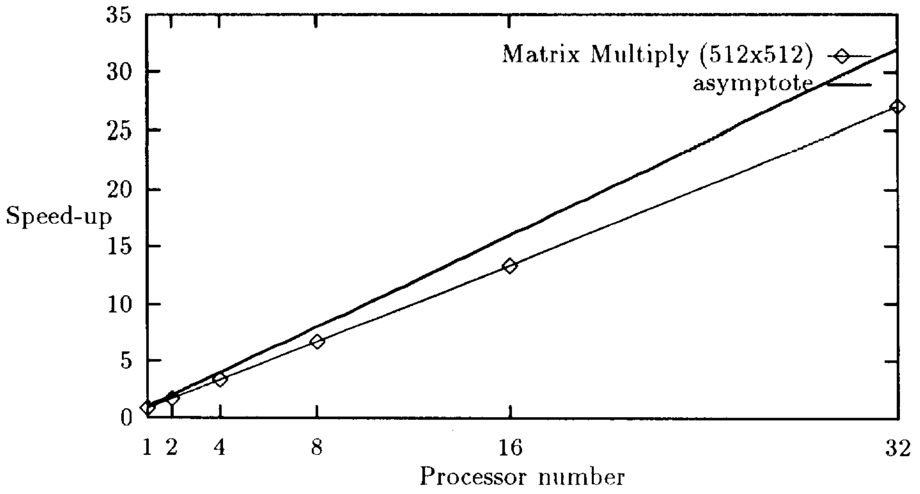

Fig. 7. Matrix Multiply: speed-up

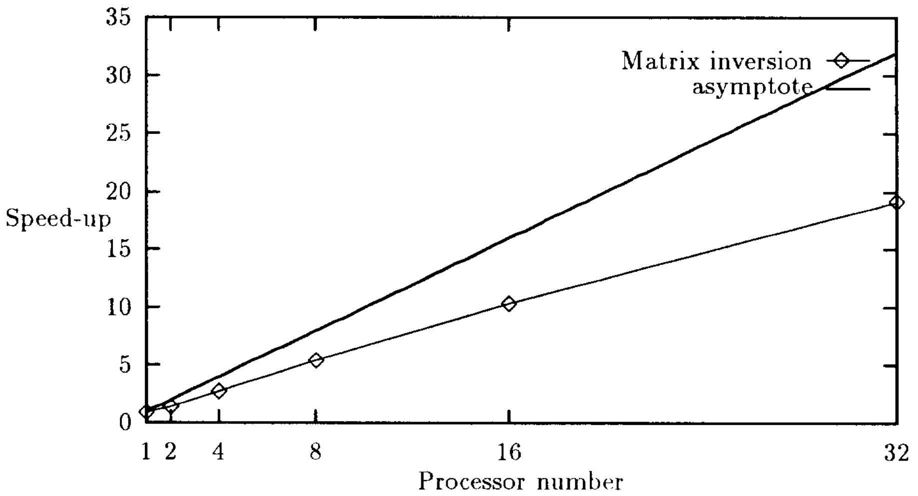

Fig. 8. Matrix Inversion: speed-up

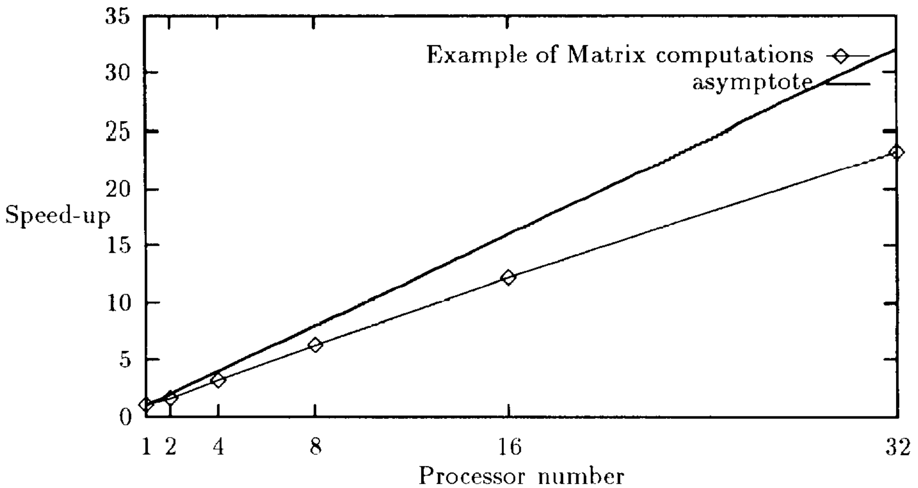

Fig. 9. Full Client program speed-up

an Eiffel dialect called Sather— would lower this dynamic binding overhead to an acceptable cost.

The actual speed of our experimental implementation (in terms of meaningful floating point operations per second) is around 0.6 MFLOPS on 32 processors, whereas a direct C implementation hits 3 MFLOPS. This is more or less the same ratio as for sequential benchmarks, so it must only be dependent on the current Eiffel implementation, which is to evolve soon.

4 Conclusion

Through the presentation of EPEE, we have proposed a new method allowing the programming of massively parallel architectures in a scalable way, using powerful object oriented concepts. EPEE facilitates MPA programming at both user and class designer levels: while providing a SPMD programming model to the designer of distributed classes, EPEE presents a sequential model to the user, so the DMPC is only seen as a powerful processor whose architecture details are ignored. Furthermore, EPEE makes it possible to reuse already existing Eiffel sequential classes in a parallel context. Of course there is no performance improvement, but neither performance loss. This allows an incremental porting of already existing applications to DMPC: we have just to select one after another interesting classes to be data parallelized, then make them inherit from Distributed Aggregate, and finally redefine access methods and optimize critical ones.

From a methodological point of view, we have shown that basic features of OOL are sufficient to express the proposed model. The class encapsulation mechanism allows us to hide the parallelism in classes describing the low level accesses to data structures without altering their interface. The modularity encourages the construction of methods by refinement. First, a simple implementation using the pure SPMD model could be realized. Optimizations may then be added for each method separately. Multiple inheritance is a property facilitating code reusability that we exploited in our use of the Distributed Aggregate class concept.

Although very promising, EPEE presently suffers from some limitations. First, our performance results are not yet totally satisfying: this is primarily due to the use of dynamic binding in OOL. However, this could be enhanced when better compiler technology is available. The limitation in performances is also due to the use of the SPMD model when all optimizations are not applied. Another limitation is relative to the exception handling: at present time we don't have defined a method to handle non-fatal exceptions so that the effect of the distributed execution is the very same as the sequential one.

EPEE offers many opportunities for future works. Indeed, the object oriented approach is a good mean —if not the only one— to tackle the problem of parallelization of operations on complex data structures like set, tree, graph, etc. The ease of use of the object oriented model allows us to achieve a lot of experiments in the domain of parallelization techniques. So we are currently experimenting EPEE with both sparse matrices and reachability graphs: this can contribute to make our Distributed Aggregate Class more general.

In EPEE, the optimization of the distributed class methods still require the distributed class designer to have a wide expertise on both the relevant application domain and on parallelization techniques. However, as the optimization principles described above seem rather general and systematic, some kind of automation can be foreseen in the form of an interactive tool to help the designer in the optimization steps for example.

Finally, it is worth noting that the overall approach of encapsulating architecture details in *had hoc* classes that can be inherited by critical classes is not limited to the DMPC context: it could be used with vectorial or systolic co-processors, thus providing a workable alternative to special purpose customized compilers.

References

1. Pierre America. *POOL-T: a Parallel Object-Oriented Language*, pages 200–220. MIT Press, 1987.
2. Birger Andersen. Ellie language definition report. *Sigplan Notices*, 1990.
3. Françoise André, Jean-Louis Pazat, and Henry Thomas. Pandore : A system to manage Data Distribution. In *International Conference on Supercomputing*, ACM, June 11-15 1990.
4. John K. Bennett. The design and implementation of DistributedSmalltalk. In *OOPSLA '87 Proceedings*, 1987.
5. David Callahan and Ken Kennedy. Compiling programs for distributed-memory multiprocessors. *The Journal of Supercomputing*, 2:151–169, 1988.

6. Rohit Chandra, Anoop Gupta, and John L Hennessy. *COOL: a Language for Parallel Programming*, chapter 8. Gelernter, D. et al., 1990.
7. C. M. Chase, A. L. Cheung, A. P. Reeves, and M. R. Smith. Paragon: a parallel programming environment for scientific appllications using communication structures. In *1991 International Conference on Parallel Processing*, 1991.
8. A. A. Chien and W. J. Dally. Concurrent aggregates (ca). In *Proc. of the Second ACM Sigplan Symposium on Principles and Practice of Parallel Programming*, 1991.
9. C. Jard and J.-M. Jézéquel. A multi-processor Estelle to C compiler to experiment distributed algorithms on parallel machines. In *Proc. of the* 9th *IFIP International Workshop on Protocol Specification, Testing and Verification, University of Twente, The Netherlands*, North Holland, 1989.
10. Michael F. Kilian. Object-oriented programming for massively parallel machines. In *1991 International Conference on Parallel Processing*, 1991.
11. Bertrand Meyer. *Object-Oriented Software Construction*. Prentice-Hall, 1988.
12. Leslie G. Valiant. A bridging model for parallel computation. *CACM*, 33(8), Aug 1990.
13. Youfeng Wu. Parallelism encapsulation in C++. In *1990 International Conference on Parallel Processing*, 1990.
14. Yasuhiko Yokote and Mario Tokoro. The design and implementation of ConcurrentSmalltalk. In *OOPSLA '86 Proceedings*, 1986.
15. Akinori Yonezawa, Jean-Pierre Briot, and Etsuya Shibayama. Object-oriented concurrent programming in ABCL/1. In *OOPSLA '86 Proceedings*, 1986.
16. Hans P. Zima, Heinz-J. Bast, and Michael Gerndt. SUPERB: a tool for semi-automatic MIMD /SIMD parallelization. *Parallel Computing*, (6):1–18, 1988.

This article was processed using the LaTeX macro package with LLNCS style