Transparent parallelisation through reuse: between a compiler and a library approach

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. Software environments for commercially available Distributed Memory Parallel Computers (DMPCs) mainly consist of libraries of routines to handle communications between processes written in sequential languages such as C or Fortran. This approach makes it difficult to program massively parallel systems in both an easy and efficient way. Another approach relies on (semi-)automatic parallelizing compilers but it has its own drawbacks. We propose to tackle this problem at an intermediate level (i.e. between high level parallelizing compilers and raw libraries), using Object Oriented (OO) technologies. We show that existing OO techniques based on the reuse of carefully designed software components can be applied with satisfactory results to the large scale scientific computation field. We propose to use a form of parallelism, known as data parallelism, and to embed it in a pure sequential OOL (Eiffel). We illustrate on several examples how sequential components and frameworks can be modified for parallel execution on DMPCs to allow for transparent parallelisation of classes using these components and frameworks.

Keywords: Distribution, Data Parallelism, Reuse, Components and Frameworks

1 Introduction

The large scale scientific computation field is looking for ever growing performances that only Distributed Memory Parallel Computers (DMPCs) could provide. The spreading of DMPCs in this user community is hampered by the fact that writing or porting application programs to these architectures is a difficult, time-consuming and error-prone task. Nowadays software environments for commercially available DMPCs mainly consist of libraries of routines to handle communications between processes resulting from the execution of programs written in sequential languages such as C and Fortran.

For instance, since its introduction in the late 50's, Fortran has been widely used for programming sequential numerical algorithms of engineering and science. Naturally, Fortran users wished to program DMPCs that way. But as if it weren't complex enough to program large (and even very large) applications with such an ill designed language, application programmers also had to deal with the whole task of parallelisation, distribution, process creation, communication management, and eventually very long debugging sessions.

In order to abstract Fortran programs toward parallel and massively parallel architectures, leading industries and academics launched the so-called High Performance Fortran Forum bound to deliver a new release of Fortran called High Performance Fortran (HPF). It is based on the so-called data-parallelism model where the set of data involved in a computation is split into partitions, to which processes are associated: this makes it possible to benefit from one of the fundamental aspects of scientific applications which is the use of repetitive computations on a large data space. HPF extends Fortran 90 (which contains itself Fortran 77 as a proper subset) by means of syntactically distinguished directives. Most representative among them are directives to specify data alignment and distribution. When coupled with array operations of Fortran 90, they would enable compilers endowed with sophisticated dependence analysis and MIMD parallelisation techniques to produce message passing code efficiently utilizing DMPCs. But there is a long way to go before such semi-automatic tools are made available. Presently available commercial tools such as MIMDizer [17] only help the user to decompose, distribute and parallelize his program interactively. Anyway, even future tools emerging from either US big companies or European Esprit projects (like the PREPARE Project) will still require the user to define (at least) his data partitions, thus altering the original sequential program either interactively or through the use of compiler directives.

Furthermore, this kind of compiler must still have a wide know-how about algorithmic parallelisation rules for the distributed data structures (actually only arrays) on which it works, and thus will be able to generate efficient code for a limited set of well known problems only: if the programmer faces a new problem, the only solution for him/her would be to get back to low level message passing Fortran, with all its drawbacks.

Another important thing to be highlighted is that data distribution compiler directives change the program semantics in such a way that it is not easily manageable outside of the compiler itself. Linking a program with HPF object code in libraries poses several problems, because some data distribution information must exist at runtime but cannot be encapsulated (in a compiler independant fashion) with subroutines performing operations on the distributed data structure, as it could be if modular or object oriented languages were used. Hence new runtime format standards or external tools such as databases holding this information will be needed, thus bringing compatibility and/or coherence problems; and adding complexity to already overwhelming complex environments.

We claim that no library level approach can solve easily the problem of code reuse on a DMPC. In this paper, we propose to tackle all these problems at an intermediate level (i.e. between high level parallelizing compilers and raw libraries), using OO technologies. We show that existing OO techniques based on the reuse of carefully designed software components can be applied to this field with good results.

We are not the first at thinking that the reuse of software components could help to manage the complexity of concurrent programming. For example in [6] it is proposed to derive parallel programs from sequential ones just by introducing asynchronous (inter-object) message passing along with the *wait by necessity* mechanism. Then *had-hoc* synchronization features encapsulated in classes allow for a customizable and versatile reuse of sequential code for safe concurrent execution. In a similar way, it is shown in [9] how synchronization constraints can be encapsulated in classes and reused (and customized) through inheritance.

We consider these approaches as very valuable and promising, but we think they lack the scalability that would make them efficient in the context of DMPCs with hundreds of processors. Actually this kind of parallelism is of a functional nature and thus it is not scalable: the definition of processes is determined by the sub-task decomposition and does not allow an efficient mapping onto the very high number of processors that may be available in a DMPC.

In a rather orthogonal way, our approach aims at embedding the scalable data parallelism programming model in an OOL, Eiffel in our case. This is presented in the next section. In the third section, we describe main reuse techniques, both within and outside of the object oriented context. Then we show how they can be applied in a distributed framework: this leads us to define reusable parallel abstractions that we illustrate on a toy example. In the fourth section, we describe how these reusable parallel abstractions can be applied to the large scale scientific computation field, and we study their performance overhead. Some implementation related remarks are made before we conclude on our experiment.

2 Programming Massively Parallel Architectures with Sequential Object Oriented Languages

2.1 Embedding data parallelism in an OOL

In our opinion a programming language should be kept as small as possible, and most notably should leave data structures and their access procedures and functions outside. So logically, we propose to conceptually remove the parallelisation know-how existing for example in a parallelizing FORTRAN compiler, and to encapsulate this know-how with the data structure to which it applies. Our approach at encapsulating parallelism can be compared to the encapsulation of tricky pointer manipulations within a linked list class, thus providing the abstraction *list* without annoying the user with pointer related notions.

Opposite to OOL where objects can be made active and methods invocations can result in actual message passing communications (sometimes referred as functional parallelism as implemented in POOL-T [1], ELLIE [2], ABCL/1 [20], Emerald [5], COOL [7] or PRESTO [4] for example), we focus on the data parallelism model associated with a SPMD (Single Program Multiple Data) mode of execution, we no longer map the object oriented message passing paradigm onto actual interprocess communications, because our goal is to completely hide the parallelism to the user (*i.e.* the application programmer).

This approach seems to be rather natural in an OO context, since object oriented programming usually focuses on data rather than on functions. Furthermore, the SPMD mode of execution appears as an attractive one because it offers the conceptual simplicity of the sequential instruction flow, while exploiting the fact that most of the problems running on DMPCs involve large amounts of data (in order to generate usefull parallelism). Each process executes the same program, corresponding to the initial user-defined sequential program, on its own data partition. The application programmer view of his program is still a sequential one and the parallelism is automatically derived from the data decomposition, leading to a regular and scalable kind of parallelism.

In [12], we have described how a sequential Object Oriented Language (OOL) can embed data parallelism in a clean and elegant way —without any language extensions— to exploit the potential power of massively parallel systems. The encapsulation of all the methods of a given object allow us to precisely know the structure of the data accesses and to define appropriate parallel techniques accordingly: parallelism is thus hidden in classes describing the low level accesses to data structures without altering their interface. These "distributed" classes are also compilation units (like any normal class), thus there are no more problems to link separately compiled modules: this is a major advantage with respect to FORTRAN-like appoaches. The modularity encourages the construction of methods by refinement. At first, a simple implementation using the pure SPMD model is realized, reusing sequential classes in this parallel context. Optimizations may then be added for each method separately: this allows an incremental porting of already existing applications to DMPCs.

2.2 The Eiffel Parallel Execution Environment

We implemented these ideas in EPEE (Eiffel Parallel Execution Environment): data distribution and parallelism are totally embedded in standard language structures (classes) using nothing but already existing language constructions. EPEE is based on Eiffel because Eiffel offers all the concepts we need, using a clearly defined syntax and semantics. 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 kind of genericity.

An EPEE prototype is available for Intel iPSC computers (iPSC/2 or iPSC/860) and networks of workstations above TCP/IP. We validated our approach through an experimentation with an implementation of distributed matrix using EPEE, and got interesting results [11].

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 appear when running an application program on a DMPC. We would like these performance improvements to be proportional to the number of processors of the DMPC (linear speed-up), which would guarantee scalability.

The designer of a parallelized class is responsible for implementing general data distribution and parallelisation rules, thus ensuring portability, efficiency and 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 should have the same semantics: each parallelized method should leave an object in the same abstract state as the corresponding sequential one.

To implement that, a designer selects interesting classes to be data parallelized, *i.e.* classes aggregating large amounts of data, such as classes based 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 (referred as DISTAGG in the following). Our distributed aggregate concept is not unrelated to the notion proposed in [8]: it is an abstract 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.

3 Towards reusing software components for parallelism

3.1 Classical approach of reuse

Reuse is not a new idea in software engineering (see for instance [13]). However, attempts to go beyond the reuse of source code and the reuse of personnel (*i.e.* reuse of the know-how of a given software engineer) are plagued by the "Not Invented Here" complex and/or lack of flexibility of existing software components.

Still, building and using libraries of (sub)routines is a classical technique that is quite successful in the scientific computation field. According to [14], this is mainly because every instance of each problem can be identified with a small set of parameters and is quite independent from other problems, and also because few complex data structures are actually involved beyond arrays. However, in the context of DMPC programming, these assumptions no longer hold —for the reasons explained in the introduction. One must rely on more modern and flexible reuse techniques.

One of the key issue in reuse is the aptitude to parameterize general purpose algorithms of data structures traversing with the "actions" to be performed at each step. In ML (actually CAML, an ML dialect [19]) we can find for example the function `map`:

```
map : (('a -> 'b) -> 'a list -> 'b list)
```

which applies its first argument (the function `('a -> 'b)` taking as input an object of type `a` and returning an object of type `b`) to a list of type `a` objects and returns a list of type `b` objects. If one wants to build a list of squares from a list of integers, one should just call:

```
# let square x = x * x;;
Value square is <fun> : int -> int
# map square [1; 2; 3; 4; 5];;
[1; 4; 9; 16; 25] : int list
```

In a language such as ANSI-C, this could be emulated through the use of function pointers (see figure 1).

However, C has not the secured flexibility of something like ML: if one wants to introduce some kind of parameterization (genericity), one must use the principle of *type coercion* which is widely known as a very unsafe feature (it can lead to runtime type errors that crash the program).

Anyway, ML and C (along with other languages like Smalltalk) are languages where routines are *first-class objects*, *i.e.* can be handled at runtime with dedicated language constructs. Whereas first-class objects provide a great deal of flexibility

```
#include <stdio.h>
#define MAXT 5
void print_square (int *n) {  printf("%d\n",*n**n); }

void apply (void (*f) (int *n), int *t)
{
  int i;
  for (i=0;i<MAXT;i++) { (*f)(&t[i]);}
}

main(void)
{
  int tab[MAXT]={1,2,3,4,5};
  apply(print_square,tab);
}
```

Fig. 1. Applying a function in *C*

(which does not always preclude type checking security: cf. CAML), this feature generally requires costly run time support: in high level languages (i.e. not *C* for that matter) the run-time data structure representing a routine is substantially more complicated than the sole address of an entry point in the code. In this context, it seems that we have to trade efficiency (a *C* like approach) for type checking security (CAML). One way to cumulate efficiency and type checking security is to make use of the feature characterizing OOL vs. modular languages like Modula or Ada, that is to say, *inheritance*.

3.2 A key towards reuse in OOL: inheritance

A large amount of numeric computations involves algorithms performing data structure walking (think of matrix operations for example). These are the computations we are actually interested in parallelizing on DMPCs, because here lies the main kind of scalable parallelism. Actually, not every program can be parallelized in a scalable way, *i.e.* its parallelisation does not necessarily bring significant performance improvement when adding more processors. To formalize that, Valiant proposed in [18] 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 this model, a computation can be efficiently parallelized only if the cost of synchronization, communications and other processing paid for managing parallelism is compensated by the performance improvement brought by the parallelisation. We will get speed-up greater than one only if the size of the data structure is large enough, and our speed-up will increase with the ratio computation time vs. communication time.

In the following, we focus on this kind of algorithms (*i.e.* those performing walking across data structure big enough); and we show how well-designed sequential

```
deferred Class ENUMERABLE [E]
  -- E is a formal generic parameter
  -- identifying the type of the ENUMERABLE elements
feature
  start is deferred end;          -- Move to some arbitrary first element
  forth is deferred end;          -- Advance to a not yet enumerated element
  off : boolean is deferred end;  -- Is there not a current item?

  item : E is
        -- Current item of the enumerable data structure.
      require not_off: not off
      deferred
      end;
  put ( new : E ) is
        -- Change the current item of the enumerable data structure with new
      require not_off: not off
      deferred
      ensure item = new
      end;
end; -- class ENUMERABLE [E]
```

Fig. 2. The generic deferred ENUMERABLE class

frameworks can be reused for parallel execution on DMPCs in a transparent way. Basically, there are two kinds of such algorithms:

- those applying a given action to each (or some of the) elements of the data structure
- and those computing an associative function based on each (or some of the) element of the data structure (much like the APL *reduce* operator).

The first step is to specify the data structures where walking algorithms can be defined at the most abstract level. We could think of abstract data types allowing some kind of finite *enumeration*, *i.e.* let us take an element, then another one, etc. until we reach the last one. It is convenient to make this abstract data type (called ENUMERABLE in the following) hold the abstract notion of *cursor*, which could be a kind of window focusing on at most one element of the ENUMERABLE object. Then an ENUMERABLE would offer the following operations:

item to look at the element under the cursor
put to change the value of the element under the cursor
start to move the cursor to an arbitrary first element
forth to advance to a not yet enumerated element
off to tell whether there is an element under the cursor

Using the Eiffel syntax, this ENUMERABLE abstract data type could be defined as described in figure 2.

This class ENUMERABLE is declared as *deferred* because it declares features without giving any implementation (deferred features): these features will be given actual definitions (*effected*) in classes inheriting from ENUMERABLE. Eiffel also allows the specification of pre-conditions (keyword **require**) and postconditions (keyword **ensure**) in the abstract data type classical way.

Our class ENUMERABLE can be seen as a generalization of the ISE Eiffel V2.3 library class called TRAVERSABLE, and will be the basis of our construction. Actually, if we need a kind of class ENUMERABLE where we can *apply* a given *action* to each element, we can define a class APPLIABLE as displayed in figure 3.

```
deferred Class APPLIABLE [E]
  inherit ENUMERABLE [E]
feature
  apply is
    do
      from start until off
        loop action; forth end
    end;
  action is do end;
end; -- APPLIABLE [E]
```

Fig. 3. The generic deferred APPLIABLE class

By mean of multiple inheritance, this class APPLIABLE could be used in a class LIST of INTEGER in the following way:

```
Class LISTINT
export squarelist, repeat FIXED_LIST       -- specify the class interface
inherit
  FIXED_LIST [INTEGER];
  APPLIABLE [INTEGER]
    rename action as square,               -- Give a more significant name
            apply as squarelist            --   to action and apply
    define start, forth, off, item, put    -- Merge APPLIABLE features with
                                           -- corresponding ones in FIXED_LIST
    redefine square;                       -- Give a new definition for square
feature
  square is do put (item * item) end;
end;
```

```
deferred Class REDUCIBLE [E,F]
inherit
  ENUMERABLE [E]
feature
  reduce : F is
    do
      Result := initialisation;
      from start until off
        loop Result := function (Result); forth end;
    end;
  function (accumulator : F) : F is do end;
  initialisation : F is do end;
end; -- REDUCIBLE [E,F]
```

Fig. 4. The generic deferred REDUCIBLE class

A client would just call `MyIntList.squarelist` to apply the square function to each element of its list.

For the reader not fluent in Eiffel, the main rules driving multiple inheritance are recalled below, (in agreement with [16]):

- If various features are inherited under the same final name *f* in class C, then they are said to be *shared*, i.e. C has only one feature named *f*. At most one instance of the features named *f* can be effected (at least all but one must be deferred) or else there is a name conflict (detected by the compiler) that must be remove through renaming.
- If two features are inherited under different final names in class C, then they are said to be *duplicated*, i.e. C has two distinct features

In LISTINT, the features **`start, forth, off, item, put`** exist in a single instance, and their implementation is the one found in FIXED_LIST.

In a similar way, if our client wish to call something like `MyIntList.maxelem` to know the higher element of the list (thus performing a *reduce* operation on the list), the class LISTINT should be modified as follow:

```
Class LISTINT
export maxelem ...
inherit
...
  REDUCIBLE [INTEGER,INTEGER]
    rename function as sup,                  -- Give a more significant name
           initialisation as neg_infinite, --  to function, initialisation
            reduce as maxelem              --   and reduce
    define start, forth, off, item, put     -- Merge REDUCIBLE features with
                                            -- corresponding ones in FIXED_LIS
    redefine sup, neg_infinite;             -- Give a new definition for these
...
feature
...
  sup (max : INTEGER) : INTEGER is
    do if item>max then Result := item  else Result := max end; end;
  neg_infinite : INTEGER is -2147483648;
...
```

where the class REDUCIBLE is as displayed in figure 4.

But now, what if we want to have another reduce-like function (say `total`, which computes the sum of the elements) on this list? We simply have to inherit again from REDUCIBLE, directly using the *repeated inheritance* mechanism of Eiffel: the inheritance graph for class LISTINT is displayed in figure 5, where deferred features are marked with an asterisk*. Again, features with the same final name will be merged, whereas features with differing final names will be duplicated (for example, `total` and `maxelem` are both renamed instances of the `reduce` feature). Here is the final text of our LISTINT class:

```
Class LISTINT
export squarelist, maxelem, total, repeat FIXED_LIST
inherit
  FIXED_LIST [INTEGER]
    rename Create as fixed_list_Create;     -- Hold fixed_list constructor
  APPLIABLE [INTEGER]
    rename action as square,                -- Give a more significant name
            apply as squarelist             --   to action and apply
    define start, forth, off, item, put     -- Merge APPLIABLE features with
                                            -- corresponding ones in FIXED_LIST
    redefine square;                        -- Give a new definition for square
  REDUCIBLE [INTEGER,INTEGER]
    rename function as sup,                 -- Give a more significant name
            initialisation as neg_infinite, --  to function, initialisation
            reduce as maxelem               --   and reduce
    define start, forth, off, item, put     -- Merge REDUCIBLE features with
                                            -- corresponding ones in FIXED_LIST
    redefine sup, neg_infinite;             -- Give a new definition for these
  REDUCIBLE [INTEGER,INTEGER]
    rename function as plus,                -- Give a more significant name
            reduce as total                 --  to function and reduce
    define start, forth, off, item, put     -- Merge REDUCIBLE features with
                                            -- corresponding ones in FIXED_LIST
    redefine plus;                          -- Give a new definition for plus
feature
  Create (n: INTEGER) is do fixed_list_Create(n) end;
  square is do put (item * item) end;
  sup (max : INTEGER) : INTEGER is
    do if item>max then Result := item  else Result := max end; end;
  neg_infinite : INTEGER is -2147483648;
  plus (acc : INTEGER) : INTEGER is do Result := acc + item end;
  positive (acc : BOOLEAN) : BOOLEAN is do Result := item>0 or else acc end;
end; -- LISTINT
```

It may look complicated, but one has to keep in mind that a client of this class only has to look at the class interface (directly provided by the *short* command), whereas a descendant can get a flat view of the inheritance graph through the *flat* command.

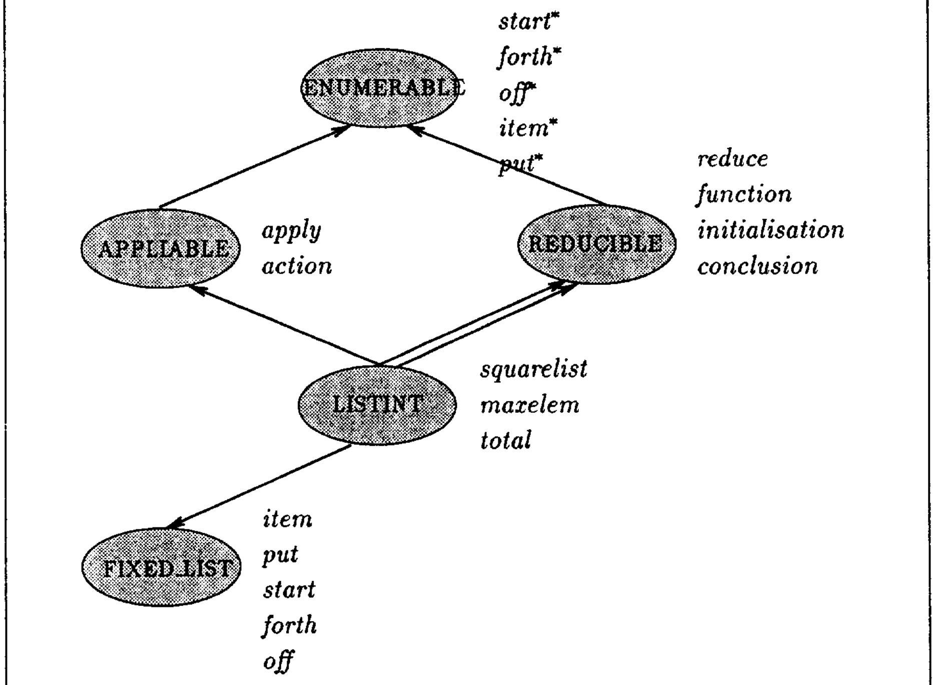

Fig. 5. The inheritance graph for the class LISTINT

We can see that the frame of the `reduce` operation has been actually reused twice (in `total` and `maxelem`) and could be still reused several times, thus achieving sequential reuse. If we can give a distributed version of this kind of reusable operation, we can achieve the transparent reuse of this distributed version by already defined features (like `total` and `maxelem`). This is the aim of the next section.

3.3 Reusing sequential code for a parallel execution

Distributed Enumerable Objects We want to reuse our class LISTINT for execution on a DMPC. The basic idea is to change the meaning of our ENUMERABLE, REDUCIBLE and APPLIABLE abstractions to deal with ENUMERABLE objects distributed across a DMPC. These distributed objects are split across the DMPC (each processor has only a piece of the object), and operations are implemented using the SPMD programming model along with the *owner-compute rule* principle. This principle assesses that an assignment is only run by the process on which the left hand side variable is located. To deal with remote accesses, a data belonging to a distributed object is *Refresh*ed before any reading attempt, i.e. the owner of this data broadcasts it to the other processors (see [3] for more details).

By definition, the class APPLIABLE makes use of local assignments only (an *action* can only be applied on the item under the cursor position), so we do not need to change it. On the contrary, the reduce function of the class REDUCIBLE is meant to access each item of the ENUMERABLE object. But if we would use the version of reduce directly as described above, each reduce function would only compute on each node a local result on the locally available items. To compute the global result from these local ones, we have to append a *conclusion* to this reduce function (this is why the computed function has to be associative: the order of evaluation may be different between a sequential execution and the distributed one). A simple minded implementation of this *conclusion* would be to have each processor send its local result to a master processor, which would then compute the global result and broadcast it. But as the number of processors of the DMPC grows, this simple minded implementation shows its performance limitations. We could choose more sophisticated algorithms, using for example minimal spanning tree algorithms where each processor computes its result with the values received from its childs on the tree, sends it to its father, and so on until the root is reached. The global result can then be sent back along the tree or simply broadcasted. We could also use a built-in system level function of the DMPC when available (on the Intel iPSC/2 for example).

A Distributed List example Our idea is to realize one (careful, robust, efficient) distributed implementation of some general purpose data structure walking algorithms that distributed implementations of sequential classes can reuse in a transparent way.

To achieve that on our example, we create a new class, DLISTINT, which has the very same interface as LISTINT but with such a distributed implementation. A client of the class LISTINT wishing to take advantage of a distributed implementation would just use DLISTINT instead of LISTINT, without altering its code in any other way. The only difference we could see would be the improvement of performances *when running on a DMPC.*

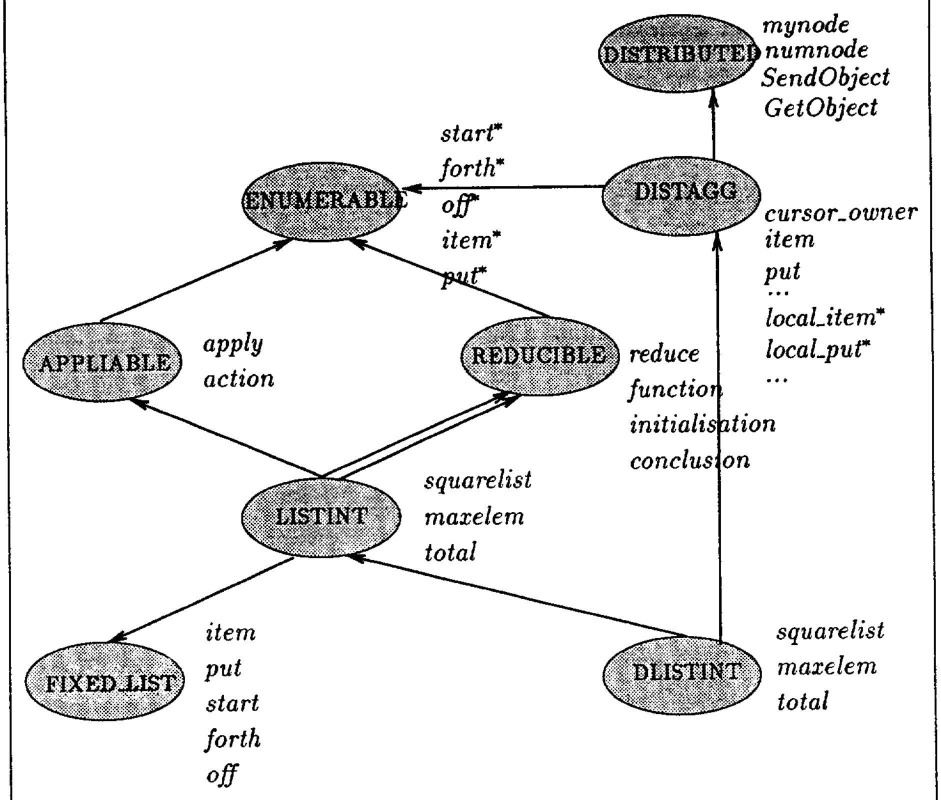

Fig. 6. The inheritance graph for the DLISTINT class

The multiple inheritance feature makes it possible to express that this distributed DLISTINT is both a LISTINT and a distributed aggregate of integers as described in DISTAGG [INTEGER]: see on figure 6 how we update the inheritance graph.

In DLISTINT, the only way to assign a value to an element of the list is to call the feature *put*. To implement the *owner-compute rule* principle for this class, we need to give a new definition of this feature, so to make it store the required value only if the processor trying to execute the assignment owns the relevant part of the list, i.e. if it owns the cursor. Symetrically, the feature *item* is the only way to access an element of the list. We give a new definition of *item* to implement the *Refresh* operation as defined above. As this is general for every distributed aggregate, these features are actually defined in the class DISTAGG in the following way:

```
deferred Class DISTAGG [E->ANY]
  -- E is a formal generic parameter
  -- identifying the type of the DISTAGG elements
inherit
  DISTRIBUTED;   -- Imports low level features like mynode, numnode...
  ENUMERABLE [E] -- DISTAGG is a special case of ENUMERABLE made of the
                 -- union accross the DMPC of "local" ENUMERABLE pieces
    define start, forth, item, put;
                 -- Give a global SPMD meaning for those
                 -- while letting off deferred
feature
  owner_of_cursor : INTEGER is deferred end;
  cursor_owner : BOOLEAN is do Result := owner_of_cursor = mynode end;
  item : E is
    do
      if cursor_owner
        then SendObject(local_item,-1); Result := local_item
        else Result ?= GetExpandedObject (owner_of_cursor)
      end
    end;
  put (v: like item) is do if cursor_owner then local_put(v) end end;
  start is
    do cursor_start; if cursor_owner then local_start end end;
  forth is
    local previous_owner : INTEGER;
    do
      previous_owner := owner_of_cursor;
      if cursor_owner then local_forth end;
      cursor_forth;
      if cursor_owner and then previous_owner /= owner_of_cursor
        then local_start end
    end;

  cursor_start is deferred end;
  cursor_forth is deferred end;
  local_start is deferred end; -- Move to arbitrary local first element
  local_forth is deferred end; -- Go to a not yet enumerated local element
  local_off : boolean is deferred end; -- Is there not a current local_item?
  local_item : like item is deferred end;
  local_put (v: like item) is deferred end;
end; -- DISTAGG
```

We can see that features dealing with the cursor (start, forth, off) are also given new meanings, according to their global semantics in our SPMD programming model. Actually we want every exported (public) feature of DLISTINT to have exactly the same semantics as in LISTINT, so that a client can use DLISTINT instead of

LISTINT transparently. The global version implementation can be given for **start** and **forth** using the local (deferred) definitions. Thus an actual heir of DISTAGG is left to *effect* the feature **off**, the local versions of cursor features (**local_start**, **local_forth**, **local_off**), along with those dealing with distribution policies (for instance **owner_of_cursor**).

To implement DLISTINT we choose a very simple distribution scheme: we cut the list in parts of more or less the same size, and allocate them to consecutive processors (as shown below in the Create feature), so that it is easy to compute the owner of a list item (see the **owner_of_cursor** feature below). To implement local versions of cursor features, we use the LISTINT cursor features: we have just to rename them as **local_start**, **local_forth**, and **local_off**. Finally, the global cursor abstraction is implemented by mean of an INTEGER (cursor), and deferred cursor related features are given a definition accordingly. The class DLISTINT eventually looks like:

```
Class DLISTINT
    export repeat LISTINT
           -- Same interface than LISTINT
    inherit DISTAGG [INTEGER]
                define local_item, local_put, local_start,
                        local_forth, local_off   -- to be merged with
                                                 --   LISTINT features
            LISTINT
                rename  Create as Listint_Create,
                        item as local_item,      -- Merge with
                        put as local_put         --   DISTAGG features
                        start as local_start,
                        forth as local_forth,
                        off as local_off;

    feature
      cursor : INTEGER;          -- implements the global cursor abstraction
      cursor_start is do cursor := 1 end;
      cursor_forth is do cursor := cursor + 1 end;
      cardinal : INTEGER;        -- The total number of items in the list
      owner_of_cursor : INTEGER -- A simple distribution scheme
        is do Result:= (cursor*numnode-1) div cardinal end;
      off : BOOLEAN is
        do Result := (cursor = 0) or else (cursor > cardinal) end;
      Create (n: INTEGER) is     -- Creates just the local part of the list
        do
          cardinal := n;
          if cardinal mod numnode > mynode
            then Listint_Create(cardinal div numnode + 1)
            else Listint_Create(cardinal div numnode + 1)
          end
    end;
```

When a client is calling `MyDListInt.total` for example, it is actually the distributed version of `reduce` which is called: transparent parallelisation is achieved through reuse.

4 A realistic example

4.1 Generality of the method

These ideas have also been applied to more realistic heirs of ENUMERABLE, like the class MATRIX encapsulating the abstract data type matrix of real, with such operations as reading, addition, multiplication, inversion, and trace (sum of the elements on the diagonal). These features have been implemented by means of the APPLIABLE and REDUCIBLE ones. However, whereas the class LISTINT had the very notion of a cursor already available through the features `start`, `forth`, and `off` (inherited from the library class LIST), other ENUMERABLE heirs may not have it. We must provide it when it is not available. Since a MATRIX element is usually accessed by means of a pair of indexes (i,j), we can encapsulate this in a class called INDEXABLE2D and say a MATRIX is an INDEXABLE2D (i.e. inherit from it). Figure 7 shows a possible definition of INDEXABLE2D.

```
deferred Class INDEXABLE2D [E]
  inherit ENUMERABLE [E] -- An enumerable with 2D indexes
  feature
    i, j: INTEGER;
    start_i is do i := 1 end;
    start_j is do j := 1 end;
    start is do start_i; start_j end;

    forth_i is do i := i + 1 end;
    forth_j is do j := j + 1 end;
    forth is do forth_j; if off_j then start_j; forth_i end end;

    off_i : BOOLEAN is do Result := i > bsup_i end;
    off_j : BOOLEAN is do Result := j > bsup_j end;
    off   : BOOLEAN is do Result :=  off_i end;

    bsup_i : INTEGER is deferred end;
    bsup_j : INTEGER is deferred end;
  end;
```

Fig. 7. The generic deferred INDEXABLE2D class

We also added new features to deal with the potentially different signatures of `apply` and `action`, as for example when only one parameter is needed:

```
deferred Class APPLIABLE [E]
...
feature
...
  apply1 (other : like Current) is
    do
      from start until off
        loop action1(other.item); forth; other.forth  end
    end;
...
```

Then the addition operation `M1.add(M2)` can be implemented as follow:

```
  inherit
    ARRAY2[REAL]
      rename Create as Array2_Create,
             item as item2d,    -- Avoid name clashes between default ARRAY2
             put as put2d;      -- features and ENUMERABLE ones
    INDEXABLE2D                 -- implementing the cursor (i,j)
      rename bsup_i as height, -- merge the upper limits of iterations
             bsup_j as width    -- with the numbers of lines and columns
      define height, width;     -- of ARRAY2
    APPLIABLE [REAL]
      rename action1 as add_item,
             apply1 as add
      define start, forth, off, item, put
      redefine add_item;
...
  feature
    item : REAL is do Result := item2d(i,j) end;
    put (v: like item) is do put (v,i,j) end;
    add_item(other_item: REAL) is do put(item+other_item) end;
...
```

Then this feature **add** may be reused "as it is" in a Class DistributedMatrix implemented the same way the DLISTINT was.

4.2 Customized walks

One point of interest appears when we need to perform a customized walk through the data structure, like for the implementation of the **trace** operation. Instead of using the common implementation for cursor moves (start, forth, off), we rename and redefine these to implement the customized walk we need:

```
...
  inherit
    REDUCIBLE [REAL]
      rename reduce as trace,
             function as plus
             forth as diagonal_forth, -- customizes REDUCIBLE with the new
                                      -- meaning for forth, as defined below
      define start, diagonal_forth, off, item, put
      redefine plus;
...
    plus (acc : REAL) : REAL is do Result := acc + item end;
    diagonal_forth is do forth_i ; forth_j end;
...
```

When the feature **trace** is called, it is actually **reduce** that is invoked, but a version of **reduce** modified so that it calls **diag_forth** instead of the default **forth**.

4.3 Reusing the class Matrix to build a class DistributedMatrix

We proceed the same way as for building a class DLISTINT from the LISTINT one. First we express that a distributed Matrix is both a Matrix and a distributed aggregate:

```
Class DMATRIX
    export repeat MATRIX
           -- Same interface than MATRIX
           -- inherit from both ancestors
    inherit DISTAGG [REAL];
            MATRIX
```

The Distributed Matrix constructor (Create feature in Eiffel) can be defined so that it splits the Matrix onto 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 creates only its own part of the matrix. Then as the only way MATRIX elements are accessed is through the features **item** and **put**, these features are renamed and given a new definition exactly the same way as for corresponding features in DLISTINT, i.e. using the global versions available in DISTAGG.

4.4 A comment on the efficiency of this approach

Since we are using rather advanced features of Eiffel (repeated inheritance, subtle use of the rename clause to manage feature duplication or merging), the question of their efficient implementation arises. It is even crucial for us, as our main rationale to use DMPCs lies in their potential computing power. If we would end up with a parallelized code running slower than the (best) sequential one, we would have completely missed the point. Fortunately, it is not so.

The first thing to be highlighted is that most of the feature name resolutions can usually be done at compile time. If a matrix M is declared of type DMATRIX (and if DMATRIX has no descendant), *every* feature call on M can be identified statically: the general dynamic binding mechanism can be discarded and replaced with a mere procedure call. For instance, the compiler can realize that the addition operation (as in `M1.add(M2)`) invoked in some client is in fact the renamed form of the `apply1` feature of the APPLIABLE class, with `action` (re-)defined in MATRIX, and `start`, `forth`, and `off` features defined in INDEXABLE2D: the direct procedure calls can be generated accordingly.

Then it is possible to avoid the overhead of procedure calls through inline expansions: the ISE Eiffel compiler can do it automatically when it finds it interesting. Finally, with all this OO stuff removed from the intermediate code, state-of-the-art compiling techniques can be used to implement loop merging, common subexpression eliminations, etc. and to generate code as efficient as a hand-written equivalent in FORTRAN.

Since the features taking advantage of the parallelism of the DMPC can rely on the reuse of the implementations of features such as `reduce` and `apply`, the final code running on the DMPC could be as efficient as the best hand-coded one (i.e. at the message passing level). As an example, we can say that our feature `add` (from class DMATRIX) is optimal because it involves absolutely no (machine level) message exchange: the better FORTRAN hand-written version would have exactly the same behavior and performance.

4.5 Implementation comments and results

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

- the DISTAGG generic class, encapsulating the distributed aggregate abstraction and the DISTRIBUTED 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 [10] 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.
- a set of tools to facilitate cross-compilation and distributed experimentation of an Eiffel program in EPEE.

We experimented our ideas through the implementation of prototype classes (Matrix and Distributed Matrix), 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 semantics of Eiffel. However, we currently have a limitation: we do not handle non-fatal exceptions properly, so the Eiffel *rescue* mechanism cannot always be used safely. The only other visible difference when executed on a DMPC is an increase of performances.

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 various methods of the class DMATRIX. We compared these results to their best sequential counterparts (i.e from the class MATRIX) run on one iPSC/2 node, thus allowing a speed-up evaluation. We got nearly linear speed-up (presented in [11]), that is to say that when problems are large enough, we can half the computing time by doubling the number of processors.

4.6 Problems and limitations

We found two kinds of problem while implementing our ideas. First, conceptual problems that limit the expressiveness and ease of use of our approach:

- Not all operation can be expressed in terms of the few reusable parallel abstractions (apply, reduce, etc.) we provide: a real implementation should be much more complete.
- Full automatic parallelism is obtained only for the features making use of our reusable parallel abstractions: other feature will also work in parallel, but no performance improvement will be obtained. However, critical methods (in terms of efficiency) can be redefined, using the SPMD programming model to take advantage of a specific data distribution. The general methods defined in the classes DISTRIBUTED and DISTAGG are to be used to hide the underlying system.
- features such as `apply` must be provided with at least three signatures: `apply` without parameter, with one, and with a list of parameters. This can be tedious, as the code is essentially the same. Furthermore, each APPLIABLE descendant class would have these three features, even if only one is needed. A possible solution would be to have three classes (APPLIABLE, APPLIABLE1, and APPLIABLEn) with only one apply feature in each of them.

We also found problems linked to the version 2.3 of the Eiffel language and its compiler:

- The Eiffel concept of genericity is very powerful, and works very well in general. However, it is not totally orthogonal to the rest of the language: we got problems when trying to mix genericity and expanded types (mostly for I/O and interprocessor communications).
- In case of complex occurrences of repeated inheritance and renaming, the compiler does not always give the expected results, and tools like *flat* can be fooled. Furthermore, the current syntactic mechanism driving the merging or the duplication of features does not have all the expressive power that we would have liked, so in a few cases, we had to hack the design of our reusable classes to make them pass through the compilation process.

On these two points, the next version of Eiffel (as described in the book *Eiffel: The Language* [15]) will bring significant improvements: expanded types should be much better integrated in the language, and a more expressive syntax will be available to drive renaming, merging, selection and even undefinition of features.

5 Conclusion

We have proposed a method based on the reuse of carefully designed software components to allow the programming of DMPCs in an easy and efficient way. A prototype of a software environment (EPEE) has been developed, implemented and experimented on real DMPCs with satisfying performances.

EPEE facilitates DMPC 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 hidden. Furthermore, EPEE makes it possible to reuse already existing Eiffel sequential classes in a parallel context, with a transparent gain in performances for features using our data structure walking abstractions.

However our prototype is just a first step demonstrating the interest of programming DMPCs at the right level, using the versatile features of OOL: we have shown there is an interesting intermediate level between full (semi-)automatic parallelizing compilers and raw message passing libraries. We currently try to extend it through experimentations with other domains, like sparse matrix operations, with promising results. But much more work is still necessary before we have some really operational and efficient object oriented environment available for the programming of DMPCs.

References

1. P. America. Pool-t: a parallel object-oriented programming. In A. Yonezawa, editor, *Object-Oriented Concurrent Programming*, pages 199–220, The MITI Press, Yonezawa A.Tokoro M., "Object-Oriented Concurrent Programming", Cambridge, MA, 1987.
2. Birger Andersen. Ellie language definition report. *Sigplan Notices*, 25(11):45–64, November 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. Briand N. Bershad, Edward D. Lazowska, and Henry M. Levy. Presto: a system for object-oriented parallel programming. In *Software-Practice and Experience*, February 1988.
5. Andrew P. et al. Black. Emerald: a general-purpose programming language. *Software-Practice and Experience*, 21(1):91–118, January 1991.
6. D. Caromel. Concurrency and reusability: from sequential to parallel. *Journal of Object-Oriented Programming*, 3(3):34–42, September 1990.
7. Rohit Chandra, Anoop Gupta, and John L Hennessy. *COOL: a Language for Parallel Programming*, chapter 8. MIT Press, 1990.
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*, March 1991.
9. J.-F. Colin and J.-M. Geib. Eiffel classes for concurrent programming. In J. Bezivin et al. (eds.), editor, *TOOLS 4*, pages 23–34, Prentice Hall, 1991.
10. 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.

11. J.-M. Jézéquel. EPEE: an Eiffel environment to program distributed memory parallel computers. In *ECOOP'92 proceedings*, Lecture Notes in Computer Science, Springer Verlag, (also to appear in the Journal of Object Oriented Programming, 1993), July 1992.
12. J.-M. Jézéquel, F. André, and F. Bergheul. A parallel execution environment for a sequential object oriented language. In *ICS'92 proceedings*, ACM, July 1992.
13. M. D. McIlroy. Mass-produced software components. In P. Naur J.M. Buxton and B. Randell, editors, *Software Engineering Concepts and techniques (1968 NATO conference of Software Engineering)*, 1976.
14. B. Meyer. Reusability: the case for object-oriented design. *IEEE SOFTWARE*, (3):50–64, March 1987.
15. Bertrand Meyer. *Eiffel: The Language*. Prentice-Hall, 1992.
16. Bertrand Meyer. *Object-Oriented Software Construction*. Prentice-Hall, 1988.
17. MIMDizer user's guide. *Version 7.02*. Technical Report, Pacific Sierra Research Corporation, 1991.
18. Leslie G. Valiant. A bridging model for parallel computation. *CACM*, 33(8), Aug 1990.
19. P. Weis, M.V. Aponte, A. Laville, M. Mauny, and A. Suárez. *The CAML reference manual*. Rapport Technique 121, INRIA, septembre 1990.
20. Akinori Yonezawa, Jean-Pierre Briot, and Etsuya Shibayama. Object-oriented concurrent programming in ABCL/1. In *OOPSLA'86 Proceedings*, September 1986.