Parallel Operators

Jean-Marc Jézéquel*, Jean-Lin 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. Encapsulating parallelism and synchronization code within object-oriented software components is a promising avenue towards mastering the complexity of the distributed memory supercomputer programming. However, in trying to give application programmers benefit of supercomputer power, the library designer generally resorts to low level parallel constructs, a time consuming and error prone process. To solve this problem we introduce a new abstraction called *Parallel Operators*. A Parallel Operator exists simultaneously on all processors involved in a distributed computation: it acts as a single ubiquitous entity capable of processing shared or distributed data in parallel. In this paper we reify this concept in our Eiffel Parallel Execution Environment (EPEE) context, and we show that it is both natural and efficient to express computations over large shared or distributed data structures using Parallel Operators. We illustrate our approach with a Parallel Operator based solution for the well-known N-body problem.

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

1 Introduction

Many 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. Indeed, it is widely believed that the DCSs commercially available today are difficult to use, which is not surprising since they are traditionally programmed with software tools dating back to the days of punch cards and paper tape.

It is now well established that an object-oriented approach can greatly alleviate the pain of programming such supercomputers [7, 9]. One of the most successful approaches relies on the Single Program Multiple Data (SPMD) programming model and is based on the idea that tricky parallel codes can be encapsulated in object-oriented software components presenting their clients with a familiar, more or less sequential-like interface [17, 12, 2, 5]. In EPEE for instance, the modularity and encapsulation properties available through the class concept are used to abstract the data representation that can then be either replicated, distributed, or virtually

* IRISA/CNRS, currently visiting Dept. of Info. Science, Tokyo University, Japan, under a JSPS grant
** IRISA/INRIA

shared (provided a Distributed Shared Memory (DSM) is available) among the processors of a DCS [4]. In this context, sequential code can safely be reused through inheritance in a distributed environment.

In order to give application programmers the benefit of DCS computational power, the library programmer still has to redefine and reprogram some features using explicit parallelism, which is both time consuming and error prone. Even if some very basic parallel patterns can be reused in new applications, they usually require syntactic contortions [6] or language extensions [10]. To solve this problem we introduce a new abstraction called *Parallel Operators*. A Parallel Operator exists simultaneously on all processors involved in a distributed computation: it acts as a single ubiquitous entity capable of processing distributed (or shared) data in parallel. In this paper we reify this concept in the EPEE framework, and we show that it is both natural and efficient to express computations over large shared or distributed data structures using Parallel Operators. Section 2 defines more precisely the notion of operator, and discuss how computations over large data structures can naturally be structured with operators. Section 3 shows how the concept of a Parallel Operator is used to go from a sequential application to an efficient parallel one. Throughout this paper, we illustrate our approach with a Parallel Operator based solution for the well-known N-body problem.

2 Using Operators in Computation Structuring

2.1 What is an Operator?

The concept of operator developed in this paper is inspired from both the G. Booch's classification of object relationships, and a reflection on the various design patterns involved in the processing of large collections of data.

We first planned to use the name agent for this design pattern, following the G. Booch [1] and the Harrap's Dictionary definitions. But the term agent is widely and increasingly used in the computer science community although not everyone agrees on the meaning.

Booch's definition of agents refers to a classification of object relationships that divides objects into *Actors*, *Servers* and *Agents*. Booch defines an agent as : "*An object that can both operate upon other objects and be operated upon by other objects; an agent is usually created to do some work on behalf of an actor or another agent*". The main concept of Booch's definition is based on the notion of a service provider. An agent is an object that *does* something on a second object and on behalf of a third one, even if the target and the client of the service can be the same object.

In the Artificial Intelligence (AI) community, the notion of agent has another, roughly similar, definition to the one given above. An AI agent is an autonomous entity that progresses in an environment where other agents exist. A system is then modeled by the interactions between several agents, each of them performing a limited set of actions. The common point between the object-oriented definition of agent and the AI definition lies in the notion of autonomy of agents, which is interpreted here as a loosely coupled design pattern. Furthermore, in both cases, an agent is only designed to perform a very specific operation.

To avoid confusion, we will use the term of *operator* in place of agent. The idea of operator is better known to object-oriented computer scientists and does not interfere with the AI world for which the definition of agent is the most famous at this time.

2.2 Beyond the Encapsulation Principle

One of the purposes of the operator design pattern is to better separate the responsibilities of each object in the modeling of a problem. A widely used approach to design classes is to follow the line of the encapsulation principle, which states that all methods manipulating data must be encapsulated along with these data. Taking for example a container class holding several items to be updated, the encapsulation principle would lead us to encapsulate the update method within the container class.

On second thoughts, this approach might not always be the best one. A container class is not necessarily an abstraction that provides operations intended to modify the state of the items holden in its structure. The only operations a class container should provide are related to the management of the data structure used to hold these items (accessing, adding and removing items, etc.). The update process is better modeled with an operator that performs the update operation on the elements stored in the container.

The main interest of such an approach in problem modeling is its ability to express the separation between classes related to the problem domain from those related to a specific implementation of the solution. Following this precept, it becomes easier to build an application in an incremental way, first providing a non optimal release to check whether the model described in the semantic classes is correctly designed. Having achieved a correct problem modeling, the programmer can think of optimizing time and space trade-offs just by modifying the implementation classes that are independent of the problem domain classes.

2.3 Modeling Computations over Large Data Structures

This section provides a definition of the patterns of collaboration between the few key abstractions used in designing operator-oriented applications. This design pattern is based on the relationships of four key abstractions, divided into *(i)* problem domain abstractions: *Operator, Element* and *(ii)* implementation abstractions: *Container, Element Provider.*

- Problem Domain Abstractions:

  Operator: An *operator* as considered here is the key abstraction suitable to model a regular operation to be performed over a collection of elements.

  Element: The *element* abstraction embodies the data manipulated by the application. Elements are the targets of operator computations.

- Implementation Abstractions:

  Container: A *container* stores elements using a specific data structure. The function of a container is also to retrieve, to access, to delete, etc, stored elements

Element Provider: The *element provider* is in charge of traversing a set of elements to be processed by an operator and generally stored in a container. Some element providers of the EPEE toolbox are also capable of generating themselves the elements they provide. Furthermore, this abstraction implements a traversal policy and do not access elements directly when associated with a container.

Thus an *operator* uses an *element provider* to reach the *elements* it has to process, these *elements* being stored in a *container* known to the *element provider*. It is possible for an operator, using the polymorphism of its element provider attribute both to traverse a container in various ways and to access elements independently of the container internal representation. These last two properties will be widely exploited in the design of parallel operators working in a distributed environment.

We can also note that we have only addressed the context of operators dealing with a collection of elements, regardless of the internal representation of the data structure used to stored these elements. But another kind of operator is also useful: those dealing with the mere organization of a data structure (*e.g.*, a sorting operation). This family of operators will not be addressed in this paper because their parallelization needs a different approach. Yet the general method we describe here is still applicable, even if its explanation would need too much space to be presented here.

2.4 Implementing the Model in Eiffel

We describe the implementation of this operator-based model in Eiffel [13], because this language features all the concepts we need (*i.e.* strong encapsulation, static typing allowing efficient code generation, 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++).

The implementation presented here is based on a four-part hierarchy reflecting the four key abstractions previously defined.

An Operator is implemented as a deferred class[3], where only the specific operation to be performed on each element is left deferred (see Example 2.1). In the same way, the class ELEM_PROVIDER is also a deferred class because we do not know how to access the items of a container, but we must be able to start a traversing, to detect the end of the traversing, and to go through the next item in the traversal (see Example 2.2). We must point out that the specification of the ELEM_PROVIDER class does not specify any order on the traversed elements.

The elements are referenced in this implementation through the generic formal parameter E because no assumption on instances of this class is necessary. Finally, the top of the class hierarchy of this implementation is shown in Figure 1 where possible effective classes are also mentioned for the Element Provider and Container hierarchies. The deferred features of the deferred classes are in italic.

[3] Also called an *abstract* class, that is a class where at least one method has no implementation: it is a deferred feature in Eiffel or a pure virtual function in C++.

```
deferred class OPERATOR[E]
feature
    run is do do_all end -- run
    attach ( prov : ELEM_PROVIDER[E] ) is
        require valid_prov: prov /= void
        do provider := clone(prov) end
    item_action (element : E) is
        -- action to perform on provided elements
        require
            element_exist: element /= void
        deferred
        end -- itemaction
feature {NONE}
    provider : ELEM_PROVIDER[E]
    do_all is
        -- do itemaction on all
        -- provided elements
        do
            from provider.start
            until provider.exhausted
            loop
                item_action(provider.item)
                provider.next
            end -- loop
        end -- doall
invariant
    provider_exist: provider /= void
end -- OPERATOR
```

Example 2.1

```
deferred class ELEM_PROVIDER[E]
feature
    start is
        -- go to the traversal starting position
        deferred
        end -- start
    next is
        -- advance to the next element
        deferred
        end -- next
    item : E is
        -- element under the provider
        deferred
        end -- item
    exhausted : BOOLEAN is
        -- have all elements been seen*
        deferred
        end -- exhausted
feature {NONE}
    container : CONTAINER[E]
            -- Optionnal
invariant
    container_exist: container /= Void
end -- ELEMPROVIDER
```

Example 2.2

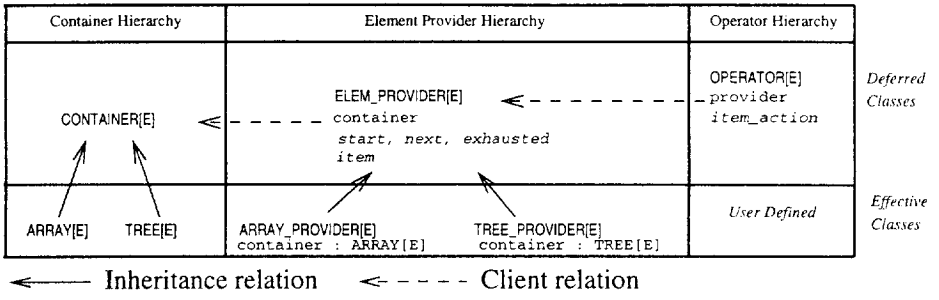

Fig. 1. Top class hierarchy

2.5 Example: N-Body Simulation

We present an actual use of the ideas developed in this paper through the modeling and implementation of the well known N-Body problem. A large amount of literature has already been devoted to this subject. It is mainly related to the optimization of the naive algorithm, the accuracy trade–off in the optimized version [15], or the parallel implementation of N-Body algorithms [14].

We consider a set of masses located in a three dimensional space, in which each mass interacts with all the others. A body is a mass with an additional speed and acceleration. The purpose of the application is to compute the time–related evolution of the system. It relies on a two-step algorithm based on time slicing. First, we compute the contribution of each mass in the (Newtonian) flow exerted on each body, and then we update the speed and the position of the bodies, taking into account all the contributions. The container used to store the bodies is called a UNIVERSE, which is basically an array of bodies.

Updating the Bodies. The operator in charge of updating the bodies of a universe is called UPDATOR and is very simple to build (see Example 2.3). Since the regular operation to be performed on each body is to call its *update* method, the UPDATOR operator only needs to designate it as *item_action*, provided its provider attribute is initialized to be a relevant element provider, i.e. an instance of an ARRAY_PROVIDER[BODY] able to reach the whole set of bodies of the Universe if this class inherits from ARRAY[BODY].

Computation of the Contributions. To compute the contributions, we first use a naive algorithm whose complexity is in $o(n^2)$. While not the best one, it allows us to introduce our methodology to design an evolutive application that can easily migrate to a distributed environment. It will be shown below that the optimal $o(n \log(n))$ algorithm is a direct consequence of choosing a tree based data structure mapped over the universe container for the computation of the contributions.

```
class UPDATOR
inherit
    OPERATOR[BODY]
creation make
feature
    make ( prov : ELEM_PROVIDER[BODY] ) is
        do
            attach(prov)
        end
                                                                    10
    item_operation (b : BODY) is
        do
            b.update
        end
end -- UPDATOR
```

Example 2.3

The computation of the contributions is performed in a two-stage approach. First we consider the contribution of a set of masses over a single body. This is the first stage of the computation that leads to the first operator in our application, called BODY_CONTRIBUTOR. The purpose of this operator is to add the contribution of all the masses it receives from its element provider to a given body. This is done via the *add_contribution_of(b:MASS)* method provided in the BODY class.

The second stage of the computation of the contributions is to consider the contributions of one set of masses to a second set. This computation is performed by another operator, UNIV_CONTRIBUTOR, that computes the contributions of each masses it receives from its element provider to each body included in the universe it is in charge of. To achieve this, the operator UNIV_CONTRIBUTOR uses a BODY_CONTRIBUTOR in its *item_action* method, as represented in Figure 2.

N-Body Simulation with Operators. All the components needed to build an N-body simulation have been described: we now have to assemble and initialize them. This is the purpose of the class N_BODY_SIMULATION presented in Example 2.4.

For instance, in order to use the universe contributor operator the programmer must specify the element providers for both the UNIV_CONTRIBUTOR and the BODY_CONTRIBUTOR operators at instantiation time. This is done when initializing the operator:

!!contributor.make (univ_prov, body_prov)

Where both *univ_pro* and *body_prov* are instance of ELEM_PROVIDER[BODY] subclass and initialized to provide bodies holden in the universe container. Their name are recall that *univ_pro* will works with the *Univ_Contributor* operator and the *body_prov* will works with the *Body_Contributor* operator.

Optimization. The basic optimization principle used in the N-body problem is based on two ideas:

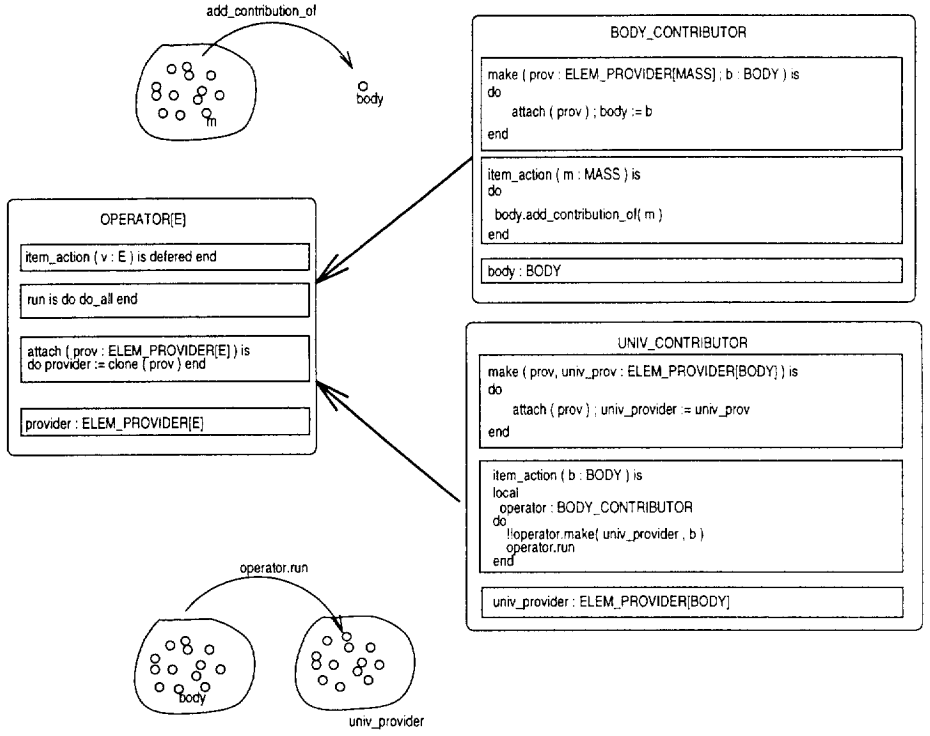

Fig. 2. Operators for contribution computations

- Spatially dividing the universe into nested regions, so as to get a tree structure.
- Using the dipole approximation that basically consists in identifying the set of bodies inside a region with their center of gravity, provided the region is far away enough from the considered target body.

Implementing this kind of optimization in our Operator Model is achieved through the design of a new container based on a tree of masses (representing the centers of gravity of each region) and whose leaves are filled with the references to the bodies contained in the universe. A new element provider is also designed to traverse the tree and provides either all its sub-trees or (if the dipole approximation applies) their center of gravity, thus transparently allowing a $o(n \log(n))$ complexity for the BODY_CONTRIBUTOR operator computation.

```
class N_BODY_SIMULATION
creation make
feature
    universe : UNIVERSE
      -- an array of bodies
    contributor : UNIV_CONTRIBUTOR
      -- an operator computing the
      -- contributions from one
      -- universe to another
    updator : UPDATOR                                    10
      -- an operator updating body states
    univ_prov : ARRAY_PROVIDER[BODY]
      -- provider for the contributor
      -- and updator operator
    body_prov : ARRAY_PROVIDER[BODY]
      -- for the BODYCONTRIBUTOR operator
feature -- entry point of the simulation
  make is
    -- Create a universe with bodies
      -- stored in a data file and                       20
      -- simulate its evolution over time
    do
      !!universe.read_from("input_data.dat")
      from
        -- Initialize the element providers
        !!univ_prov.make(universe)
        !!body_prov.make(universe)
        -- Initialize the operators
        !!contributor.make(univ_prov,body_prov)
        !!updator.make(univ_prov)                        30
      until universe.end_of_times
      loop
        contributor.run
        updator.run
        universe.advance_time
      end -- loop
    end -- make
end -- NBODYSIMULATION
```

Example 2.4

3 From Sequential to Parallel with Parallel Operators

3.1 The EPEE Parallel Programming Model

The kind of parallelism we consider takes inspiration from Valiant's Block Synchronous Parallel (BSP) model [16]. A computation that fits the BSP model can be seen as a succession of parallel phases separated by synchronizations and sequential phases.

In EPEE [5], Valiant's model is implemented based on the Single Program Multiple Data (SPMD) programming model. Data can either be distributed across the processors, or virtually shared if a Distributed Shared Memory is available on the target architecture. EPEE provides a design framework where data representations are totally encapsulated in regular Eiffel classes [4], without any extension to the language nor modification of its semantics. In both cases, the user still views his program as a sequential one and the parallelism is derived from the data representation: each processor runs a subset of the original execution flow (based on the Owner Write Rule, that is, a processor may only write its data partition). The SPMD model preserves the conceptual simplicity of the sequential instruction flow, while exploiting the fact that most of the problems running on high-performance parallel computers are data-oriented and involve large amounts of data in order to generate scalable parallelism.

From the application programmer's point of view, the distributed execution of

some part of his code is hidden through the use of software components that are responsible for the correct management of the parallel phases of the application. All it is needed to benefit from a parallelized execution would be to pick up the suited reusable components in the EPEE toolbox that match the global behavior expected during the parallel phases. Thus, the programmer manipulates concepts and abstractions whereas the matching components of the EPEE toolbox are designed to implement the details and to cooperate smoothly with each other.

The EPEE toolbox also includes cross-compilation tools that mainly consist of script files that deal with compiler flags and options correctly. Any application designed with EPEE thus can be compiled for any target mono-processor or multi-processor (provided an Eiffel run-time system exists for this processor). EPEE also includes two highly portable communication libraries intended to deal with the two communication paradigms used in EPEE: the message passing and the shared memory. The Parallel Observable Machine (POM), which provides sophisticated facilities for tracing a distributed execution [3] is used to implement message passing communications. The Shared Virtual Memory Library (SVM Lib) is used to allow to allocation of objects in shared memory. These libraries are available for several platforms (*e.g.*, Intel iPSC/2, Paragon XP/S, Silicon Graphics Power Challenge and networks of UNIX workstations) using various communication kernels or operating systems (*e.g.*, PVM, BSD sockets, NX/2, SunMos, IRIX, etc.).

3.2 The Parallel Operator Model

The Parallel Operator Model is designed as an extension of the Operator Model to be used within the EPEE framework. Here are the new meanings of its key abstractions.

Shared Element. A shared element is an object allocated in a shared memory space using the facilities offered by the EPEE toolbox. The EPEE support for shared objects is based on a shared memory mechanism, called a Distributed Shared Memory (DSM) when it is provided at the operating system level on the target architecture. A DSM provides a virtual address space that is shared by the processes of different processors on distributed systems. More details on the mechanisms developed to make shared objects available in the EPEE framework can be found in [8]. The only thing the reader must keep in mind is that the shared objects are fully compatible with normal ones. This property is exploited to reuse the sequential containers that may now hold references to shared objects in the distributed version of an application.

Distributed Element Container. A distributed element container is an abstraction that is able to manage both local and non local items transparently. Basically, a distributed element container is implemented as a container for which the data structure is spread over all the processors. When a client object requests a reference to an item, the distributed element container is able to detect whether the requested item is local, and can thus reply with the local reference of the item. Or if the container detects that the requested item is allocated on another processor and then it asks the owner of the real item for a copy.

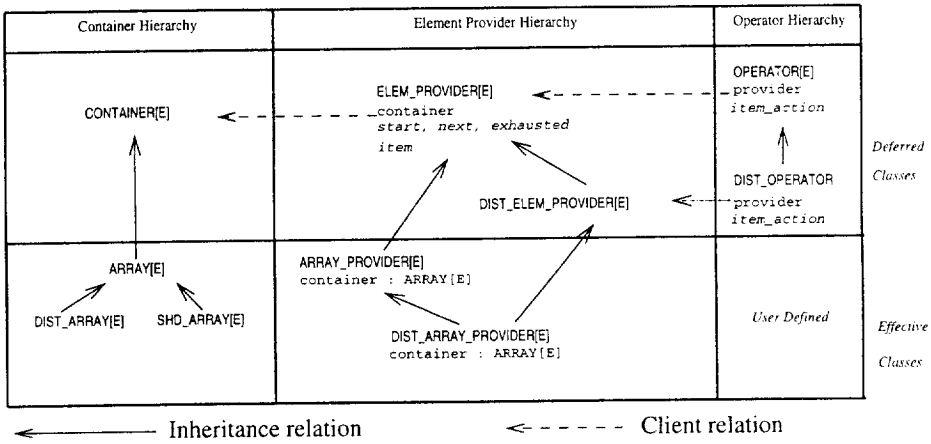

Fig. 3. Top class hierarchy in the parallel operator model

Element Distributed Provider. The distribution of control over the processes of a data parallel program is achieved by a domain decomposition of the iterations used over the (large) data structure. Following this idea, when a sequential iteration accesses N variables, the data parallel iteration on each processor accesses $\frac{N}{p}$ variables only (if p processors are available).

The choice of the $\frac{N}{p}$ variables to be accessed on a given processor is made using the *owner write rule*. This rule states that only the items owned by the processor can be accessed for modification [5].

When the data are actually distributed across the memory of each process, the ownership exactly matches the distribution. When the data are allocated in shared memory, the ownership is translated to a virtual ownership. The virtual ownership states that even if all the processes might potentially modify all the data, only one process per shared datum can actually modify it. In our model design the control distribution is implemented with an Element Distributed Provider (EDP). Thus the same EDP can be used for shared and distributed data to reach the elements to be processed in a distributed iteration.

Parallel Operator. A Parallel Operator is the entry point of a parallel phase from the application programmer's point of view. Its role is basically to run the computation on each processor for the data made available through its EDP. The blocking semantics to be preserved in a distributed environment states that the execution flow goes out of the method call only when the execution is terminated. In a distributed environment a routine call execution is considered to be terminated when each execution flow in each processor has reach the end of the routine. As a consequence, the

faster processors need to wait for the other ones to preserve the routine call semantics. It is the Parallel Operator that implements such a mechanism through the use of appropriate synchronizations. This is done using the Sync–Exec–Sync scheme as formally described in [11]. We can briefly recall the purposes of the synchronizations before and after the distributed execution of a computation:

- The first synchronization ensures that the values of the data read during the *Exec* stage are updated, i.e. are the same as they would have been under a sequential execution.
- The second synchronization ensures that the data read during the parallel phase will not be overwritten until global completion of the operator execution.

The actual implementation of the synchronizations depends on the memory model used in the container class holding the elements processed by a parallel operator. Thus, an parallel operator triggers the synchronization through an EDP which itself asks the container for the actual implementation of the synchronizations.

Summary of the Responsibilities. The patterns of collaboration among these four abstractions may be summarized as follows. A Parallel Operator manages the synchronizations ensuring a correct execution according to a SPMD programming model along with the Sync–Exec–Sync scheme. The EDP associated with the Parallel Operator implements a specific access policy on the subset of the whole data set using a data domain decomposition scheme. For a Parallel Operator the access to each element is made transparently from the container data structure through its EDP. Finally, the actual retrieval of an element, given a position in the structure, is performed by the container associated with the EDP. This last abstraction is in charge of dealing with the problem of data allocation that can either be shared or distributed.

3.3 Using Parallel Operators

To use parallel operators in an actual distributed application, the programmer first has to choose a memory model for his application. A distributed memory model or a shared memory model is available through EPEE library classes for basic data structures (DIST_ARRAY, SHD_ARRAY, etc.) used for containers. For each container class there is an EDP implementing a domain decomposition policy. These EDPs are independent of the memory model encapsulated in the container class but not of the data structure implemented by these containers.

The parallelization of the operator–based sequential application can then be easily done using the multiple inheritance mechanism. The main idea is to build specialized parallel operators by inheriting from both:

- the sequential operators to reuse the code for the local processing
- and the parallel operator class to achieve the distributed processing.

It should be noted that the programmer who uses operators must be aware of certain problems that can arise when one operator is used within another. Since

there is no ordering specified on the elements provided by an ELEMENT_PROVIDER instance, a programmer can use nested operators only if no data dependencies exist between the data processed by the various operators. Processing the data in parallel therefore does not bring new problems, provided only one parallel operator runs at a given time. This constraint is motivated by both the semantics of a parallel phase in the BSP model (one parallel phase cannot launch a second one) and by the implementation of the parallel operators (the synchronizations of parallel operators would not work correctly). This constraint could be described as a class invariant in the parallel operator class, and forced upon the programmer through typing, because a user-defined parallel operator inherits from the general parallel operator class, and thus from its invariant.

3.4 Parallel N-Body Simulation

Using the method presented in the previous sections, we now describe the parallelization of the N–body application. We start with the modifications of the data allowing distributed computation and then we discuss the parallelization of the two main operators. We conclude with the main code of the distributed version of the N–body simulation program.

The parallel implementation of the N–body application is based on the use of shared bodies, that is BODY objects allocated in shared memory. To do so, we build a SHD_BODY class by inheriting from both the BODY and the SHD_OBJECT[4] classes.

The container used to store the bodies and to implement the UNIVERSE class can still be the same. This is possible because the class SHD_BODY conforms to the class BODY. Thus, no change is needed in the UNIVERSE class excepting for the method creating the bodies stored in the container.

Figure 4 shows how two processes can access the same shared bodies through the use of a local container. In this figure, each circle represents an instance of the class SHD_BODY. Shared bodies can be accessed for read purposes by the p processes of the parallel application using sequential element providers. However, when using an EDP with a local container, only a sub–set of the bodies are accessed, as represented by the dashed arrows starting from the DIST_UNIV_PROVIDER objects.

Parallelization of the Update Computations. To obtain a parallel operator which updates the bodies in parallel, we inherit from both the UPDATOR operator to get the definition of the ITEM_OPERATION and from the DIST_OPERATOR of the EPEE toolbox (see Figure 5). At initialization time, the parallel operator DIST_UPDATOR has to be provided with an EDP matching the container chosen during the UNIVERSE implementation. This is done using the class constructor: *updator.make(dist_provider)*, where *dist_provider* is an instance of DIST_ARRAY_PROVIDER and is initialized to work with the UNIVERSE container.

Parallelization of the Computation of the Contributions. In the computation of the contributions, only the UNIV_CONTRIBUTOR operator is parallelized. The

[4] Provided in the EPEE toolbox

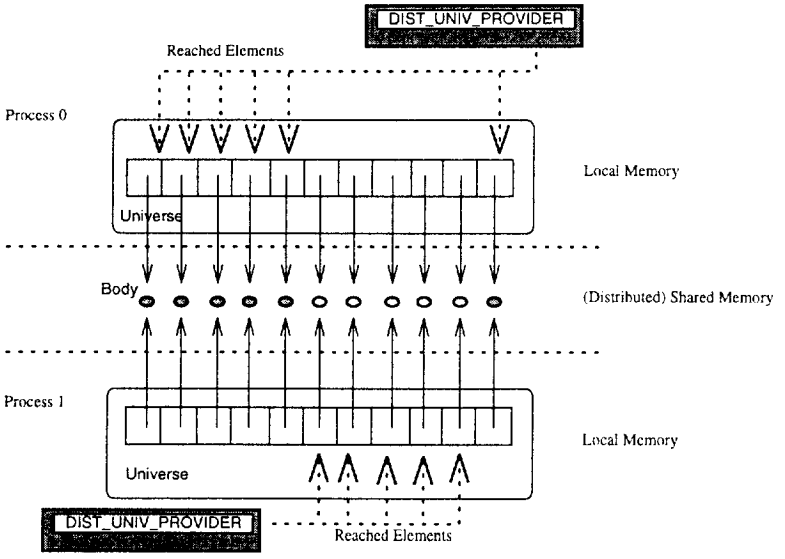

Fig. 4. Distributed access to shared bodies.

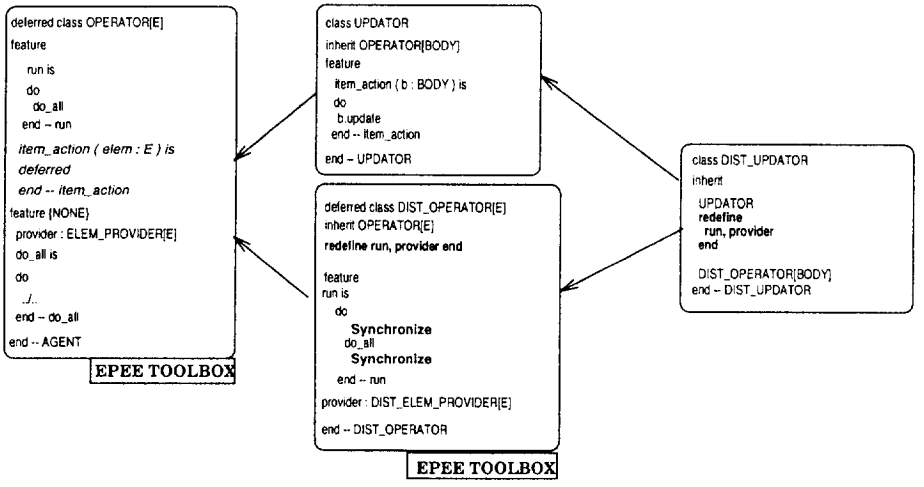

Fig. 5. The parallel updator operator

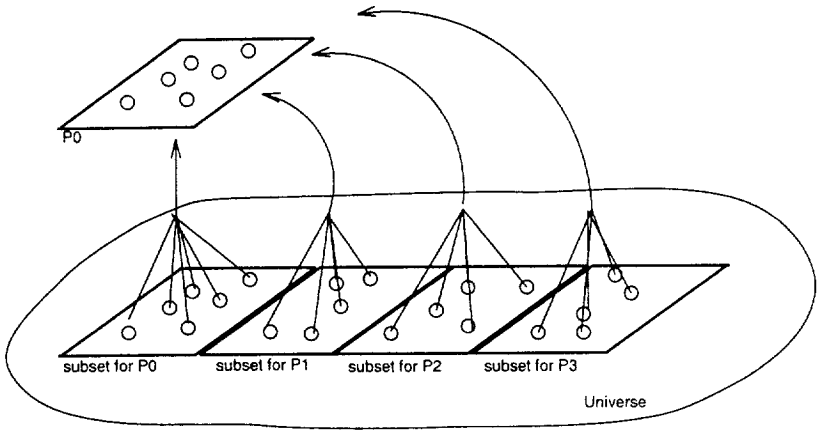

Fig. 6. Parallelization scheme for the contribution

parallel version of this operator still uses the sequential operator BODY_CONTRIBUTOR (itself working either with a container structured as an array or a tree).

For each processor, the parallel UNIV_CONTRIBUTOR operator works on one subset of the universe only. For each body of its sub-universe it computes the contribution of the whole universe using the sequential operator BODY_CONTRIBUTOR as defined previously. In this design, the parallelism comes from the fact that each sub-universe computation is run concurrently on each processor using the parallel operator. Figure 6 shows the behavior on the process 0 of the parallel operator DIST_UNIV_CONTRIBUTOR: for each body of the subset assigned to this process, the operator computes the contribution of all the bodies in the universe (itself either an array or a tree).

To implement this design we must point out that the notion of subset of the universe is purely virtual because no object in the system represents this abstraction. A subset of the whole universe can only be viewed through a specific element provider that reaches the bodies attached to this virtual subset of the universe only. The choice of the bodies assigned to one or another subset is effected using an EDP. The actual code of the DIST_UNIV_CONTRIBUTOR is described in Figure 7.

Parallel Version of the N-Body Simulation. Once both the universe container has been redefined to hold shared bodies and parallel operators have been defined, only limited modifications need to be applied to the sequential version of the code of the N-body simulation program. Indeed, in Example 3.1, we just redefine the type of the universe and that of the parallel operators, while reusing all the sequential code through inheritance.

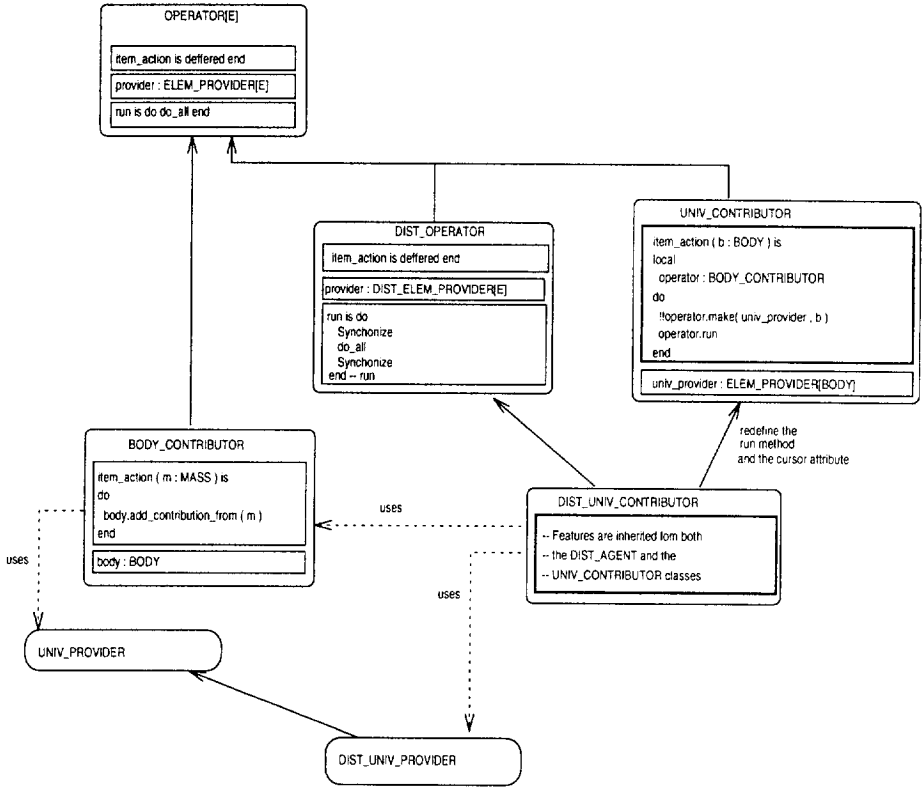

Fig. 7. The parallel contributor operator

3.5 About the Efficiency of Parallel Operators

Since we use rather advanced features of Eiffel (*e.g.*, multiple and repeated inheritance) to implement Parallel Operators, the question of their runtime efficiency arises. This in fact is a crucial point, since the main rationale for using distributed parallel computers lies in their potential computing power. If we were to end up with a parallelized code running slower than the (best) sequential one, we would have completely missed the point. Fortunately, this is not the case.

The design model we have presented in this paper allows the programmer to build applications which can easily migrate to distributed environments. Furthermore, thanks to the dynamic biding of the attributes in the operator and element provider classes, operators can dynamically change their behaviors at runtime. The drawback of this approach is that a sub-optimal code may be generated, because this generality and dynamism have a price. However, generality can be easily traded

```
class DIST_N_BODY_SIMULATION
inherit
    N_BODY_SIMULATION
        redefine
            universe, universe_provider, contributor, updator
        end
creation make
feature
    universe : SHD_UNIVERSE
            -- Now an array of shared bodies                                    10
    contributor : UBIK_UNIV_CONTRIBUTOR
            -- redefined to a parallel operator
    updator : UBIK_UPDATOR
            -- redefined to a parallel operator
    universe_provider : DIST_ARRAY_PROVIDER
            -- Distributed provider for the parallel operators
end -- DISTNBODYSIMULATION
```

Example 3.1

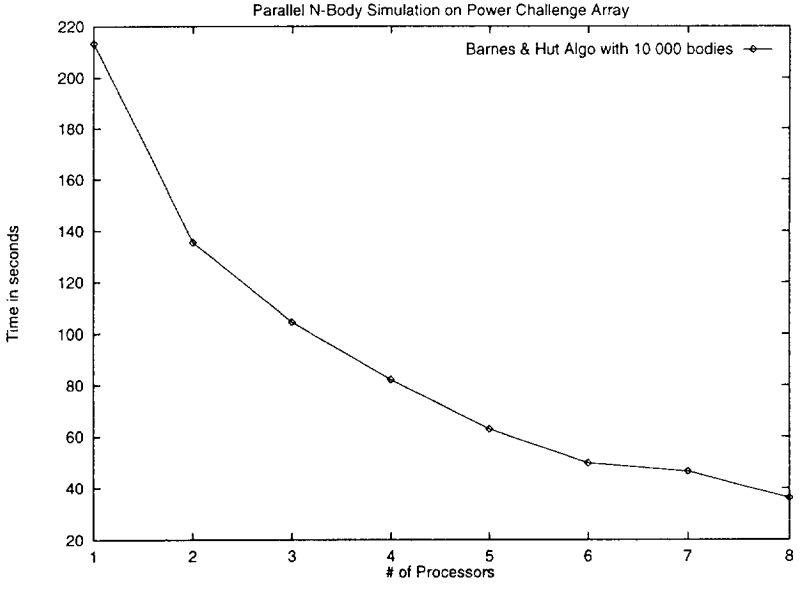

Fig. 8.

for performances if this is a critical key point in the application, as is the case for our N-body simulation. Indeed the programmer can specialize his operators and element providers using the redefinition of their attributes to match their known effective type (*e.g.*, provider can be redefined to be a DIST_ARRAY_PROVIDER[SHD_BODY] in the DIST_UPDATOR), thus allowing a compile-time binding of features (since most of the feature name resolutions can now be done at compile time).

For example, the call for the *run* method from the Parallel Operator DIST_UPDATOR can be statically bound, as well as for the inner call for the feature *update* on a SHD_BODY object (though the latter needs a simple data flow analysis). In these cases, the general dynamic binding mechanism can be discarded and replaced with a mere procedure call. Furthermore, since the feature size of the various operator classes is very small (typically a single line), it is possible to avoid the overhead of procedure calls through in-line expansions which most Eiffel compilers do automatically whenever the methods are small enough.

The generated code can then look exactly the same as if it had seen coded by hand by a proficient parallel programmer. A trivial example is the call for *updator.run* method, which is optimal because it involves absolutely no (machine level) message exchange: the best FORTRAN handwritten version would have exactly the same behavior and performance.

We have made some experiments of the N-Body simulation with the EPEE environment on the Power Challenge parallel computer. The results are shown in Figure 8 and are obtained using one to the eight processors available on this computer.

4 Conclusion and Future Work

In this paper, we have introduced a new abstraction called *Parallel Operator*. A Parallel Operator exists simultaneously on all processors involved in a distributed computation: it acts as a single ubiquitous entity able to process data in parallel. Parallel Operators help application programmers to make the most of the distributed memory supercomputers computational power, because they no more need to get involved in the tricky details of loop parallelization.

We have described an approach allowing an easy parallelization of object-oriented computations structured around operators acting as a design pattern for the programming of distributed systems. We have shown that it is both natural and efficient to express computations over large shared or distributed data structures using Parallel Operators. We have illustrated our approach with a Parallel Operator based solution for the well-known N-body problem.

While structuring object-oriented computations with operators slightly departs from established principles, it should be noted that it is often the most natural way to express solutions within a statically typed object-oriented language. For example, if block closures (as in Smalltalk) are not first class objects, the use of operators is one of the most elegant solutions to model a problem where data must be sorted according to various criteria. Thus it should not come as a surprise that some popular data structure libraries (such as Booch's components for Eiffel or C++) use the concept of operator extensively. This opens a very promising perspective: the parallelization of Booch's components using Parallel Operators within EPEE.

Another domain where we plan to apply the concept of Parallel Operator concerns distributed linear algebra over large sparse matrix computations. Indeed these computations can be modeled with Parallel Operators operating over large sparse distributed data structures. Since a Parallel Operator is decoupled from actual sparse matrix representations, it is much easier for the application programmer to benefit from internal representations using sophisticated data structures on shared or distributed memory.

References

1. Grady Booch. *Object-Oriented Analysis and Design with Applications.* Benjamin Cummings, 2nd edition, 1994.
2. J. Dongarra and al. An Object-Oriented Design for High-Performance Linear Algebra on Distributed Memory Architectures. In *Proceedings of the Object-Oriented Numerics Conference (OON-SKI'93)*, 1993.
3. F. Guidec and Y. Mahéo. POM: a Virtual Parallel Machine Featuring Observation Mechanisms. PI 902, IRISA, January 1995.
4. F. Hamelin, J.-M. Jézéquel, and T. Priol. A Multi-paradigm Object Oriented Parallel Environment. In H. J. Siegel, editor, *Int. Parallel Processing Symposium IPPS'94 proceedings*, pages 182–186. IEEE Computer Society Press, April 1994.
5. J.-M. Jézéquel. EPEE: an Eiffel Environment to Program Distributed Memory Parallel Computers. In *ECOOP'92 proceedings*, number 611. Lecture Notes in Computer Science, Springer Verlag, (also published in the Journal of Object Oriented Programming, 1993), July 1992.
6. J.-M. Jézéquel. Transparent Parallelisation Through Reuse: Between a Compiler and a Library Approach. In O. M. Nierstrasz, editor, *ECOOP'93 proceedings*, number 707, pages 384–405. Lecture Notes in Computer Science, Springer Verlag, July 1993.
7. J.-M. Jézéquel. *Object Oriented Software Engineering with Eiffel.* Addison-Wesley, March 1996. ISBN 1-201-63381-7.
8. J.-M. Jézéquel and J.-L. Pacherie. Shared Objects in KOOPE. Technical Report 2.3, Irisa-Intel ERDP, June 1995.
9. Michael F. Kilian. Object-oriented programming for massively parallel machines. In *1991 International Conference on Parallel Processing*, 1991.
10. H. Konaka, T. Tomokiyo, M. Maeda, Y. Ishikawa, and A. Hori. Data parallel programming in the parallel object-oriented language OCore. In *Proc. of the Workshop on Object-Based Parallel and Distributed Computation (OBPDC'95)*, Tokyo, June 1995. Springer-Verlag, LNCS, to be published.
11. Y. Mahéo. *Environnements pour la compilation dirigée par les données : supports d'exécution et expérimentations.* PhD thesis, IFSIC / Université de Rennes I, juillet 1995.
12. M. Makpangou, Y. Gourhant, J. Le Narzul, and M. Shapiro. Fragmented objects for distributed abstractions. *IEEE Software*, May 1991.
13. B. Meyer. *Object-Oriented Software Construction.* Prentice-Hall, 1988.
14. John W. Romein and Henri E. Bal. Parallel n-body simulation on a large-scale homegeneous distributed system. In Peter Magnusson Seif Haridi, Khayri Ali, editor, *EURO-PAR'95 Parallel Processing*, volume 966 of *Lecture Notes in Computer Science*, pages 473–484. Springer, August 1995.
15. John K. Salmon and Michael S. Warren. Skeletons from the treecode closet. *Journal of Computational Physics*, page 24, July 1993.

16. Leslie G. Valiant. A bridging model for parallel computation. *CACM*, 33(8), Aug 1990.
17. Youfeng Wu. Parallelism encapsulation in C++. In *International Conference on Parallel Processing*, 1990.