Mathematical Modelling and Numerical Analysis
Modélisation Mathématique et Analyse Numérique

ESAIM: M2AN
M2AN, Vol. 36, N° 5, 2002, pp. 837–861
DOI: 10.1051/m2an:2002039

DISTRIBUTED OBJECTS FOR PARALLEL NUMERICAL APPLICATIONS

Francoise Baude[1], Denis Caromel[1] and David Sagnol[1]

Abstract. The C++// language (pronounced C++ *parallel*) was designed and implemented with the aim of importing *reusability* into parallel and concurrent programming, in the framework of a MIMD model. From a reduced set of rather simple primitives, comprehensive and versatile libraries are defined. In the absence of any syntactical extension, the C++// user writes standard C++ code. The libraries are themselves extensible by the final users, making C++// an open system. Two specific techniques to improve performances of a distributed object language such as C++// are then presented: Shared-on-Read and Overlapping of Communication and Computation. The appliance of those techniques is guided by the programmer at a very high-level of abstraction, so the additional work to yield those good performance improvements is kept to the minimum.

Mathematics Subject Classification. 68N15, 68N19.

Received: 11 December, 2001. Revised: 23 May, 2002.

1. Introduction

Reusability has been one of the major contributions of object-oriented programming; bringing it to parallel programming is one of our main goals, and a major step forward for software engineering of parallel systems. Part of the challenge is to combine the potential for extensive reuse with the high performance which is usually required of parallel and real-time systems.

Working mainly within the framework of physically distributed architectures, we are concerned with both explicit and implicit parallelism in both the problem and solution domains. Our applications include parallel data structures, computer-supported cooperative work (CSCW), and fault-tolerance and reliability in safety-critical and real-time systems.

To achieve this end, we began design and implementation of C++// early in 1994, and we are pursuing now this research in the context of the Java language, with the definition of a library called *ProActive PDC* [18]. The C++// language benefited from previous research done on the Eiffel// language [12, 15]. Important ideas and techniques from that work have reappeared in the definition of a reduced set of simple primitives that are then composed to create comprehensive and versatile libraries, which —most importantly— can then be extended by end users.

Another important characteristic of our system is the complete absence of any syntactical extension to C++. C++// users write standard C++ code, relying on specific classes to give programs a parallel semantics. These

Keywords and phrases. Concurrency, data-driven synchronization, dynamic binding, inheritance, object-oriented concurrent programming, polymorphism, reusability, software development method, wait-by-necessity, overlap, object sharing.

[1] OASIS, Joint Project CNRS, INRIA, University of Nice Sophia Antipolis, 2004 route des Lucioles, BP 93, 06902 Valbonne Cedex, France. e-mail: Denis.Caromel@sophia.inria.fr

programs are then passed through a pre-processor, which generates new files. The original and new code is then compiled and linked with a standard C++ compiler. When appropriate, all names related to the `C++//` system include the `ll` root in their name (for "parallel"). During the presentation of our system, we will conform to the following symbols when introducing:

a *model* principle or rule: ◊ a new *syntax*, class, or member: Ⓢ
a *file* used in our system: ⊡ *examples*: ∇

We hope these conventions ease reading and quick referencing through the paper.

This article begins by describing the basic features of our programming model, which is an MIMD model without shared memory. Section 3 deals with the control programming of processes, *i.e.* the definition of concurrent process activity. A recommended method for parallel programming in `C++//` is outlined in Section 4.1. Those parts of the programming environment which handle compilation and mapping are described in Section 4.2, and an overview of the implementation techniques which make the system open and user-extensible is given in Section 4.3. Finally, we present two performance optimizations that to be applied demand some implication from the user, but at a high-level of abstraction: on one side, sharing objects among processes in case they are in the same address space, on the other side, overlapping communication with computation in the framework of remote method calls. This paper ends up with conluding remarks in Section 7.

2. Basic model of concurrency

This section describes four important characteristics of our parallel programming model: parallel processes, communication between them, synchronization, and data sharing. As described below, we adopt a MIMD model without shared memory, which means that there are no directly-shared objects in our system.

Along with simplicity and expressiveness, reusability is one of our major concerns. More specifically, we want to allow users to take an existing C++ system and transform it into a distributed one, so that they may derive parallel systems from sequential ones [14].

2.1. Processes

One of the key features of the object-oriented paradigm is the unification of the notions of module and type to create the notion of class. When adding parallelism, another unification is to bring together the concepts of class and process, so that every process is an instance of a class, and the process's possible behavior is completely described by its class.

◊ *Model:* the process structure is a class; a process is an object executing a prescribed behavior.

However, not all objects are processes. At run-time, we distinguish two kinds of objects: *process objects* (or active objects), which are active by themselves, with their own thread of control, and *passive objects*, which are normal objects. This second category includes all non-active objects. An example of the arrangement of processes and objects at run-time is given in Figure 1.

At the language level, there are two ways to generate active objects. In the first, an active object is obtained by instantiating a standard sequential C++ class using `Process_alloc`:

Ⓢ *Syntax:*

```
A* p;          // A is a normal sequential class
p = (A *) new Process_alloc ( typeid (A), ...);
```

In this case, a standard sequential class `A` is instantiated to create an active object, which then has a FIFO synchronization: method invocations are serviced in the order in which they are made. The `Process_alloc` class is part of the `C++//` library, while `typeid` is the standard C++ run-time type identification (RTTI) operator. We will refer to this technique as the *allocation* style of process creation, and say that it produces an *allocation process*, or *allocation active object*. The allocation style is convenient, but limited because it only allows us to create processes with a FIFO behavior.

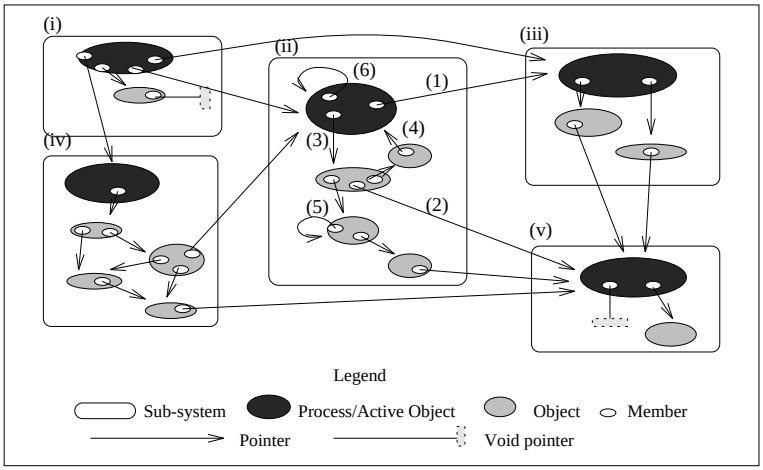

FIGURE 1. Processes and objects at run-time.

The second technique, which we call *class-based*, is more general:

◊ *Model:* all objects which are an instance of a class which publicly inherits from the `Process` class are processes.

This `Process` class is part of the `C++//` library. To use class-based process creation, the programmer must therefore derive a specific class, called a *process class*, from `Process`, as in:

Ⓢ *Syntax:*

```
class Parallel_A : public A, public Process {
  ...
};
⋮
Parallel_A* p;
p = new Parallel_A(···);
```

As with the allocation-based technique, instances of sub-classes of `Process` have a default FIFO behavior. However, as we will see in the following sections, it is possible to change this to create other behaviors. We say that the class-based technique generates *class-based processes*, or *class-based active objects*.

As shown in Figure 1, passive objects (*i.e.*, objects which are not active) belong at run-time to a single process object. This organizes a parallel program into disjoint sub-systems, each of which consists of one active object encapsulating zero or more passive objects. Figure 2 presents the two styles of active object definition.

2.2. **Sequential or parallel processes**

A major design decision for any concurrent programming system is whether processes are sequential (*i.e.*, single-threaded) or able to support internal concurrency (*i.e.*, multi-threaded). Because our system is oriented towards reuse and software engineering of parallel systems, rather than operating systems programming, we made the following choice:

◊ *Model:* a process is sequential, it is single-threaded.

We believe that single-threaded processes are easier to reuse, and easier to write correctly.

The model does not allow the user to program multi-threaded processes, but this does not prevent multi-threading at the operating system level. As we will see in Section 4.2, several sequential processes can be implemented with one multi-threaded operating system process for the sake of light-weightness.

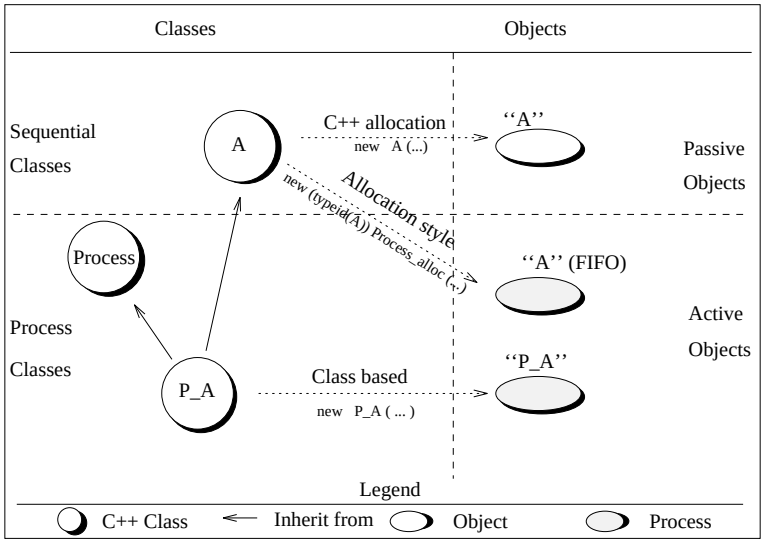

FIGURE 2. Allocation and class based active objects.

2.3. **Communication**

Since a process is an object, it has member functions. When an object owns a reference to a process, it is able to communicate with it by calling one of its public members. This is `C++//`'s inter-process communication (IPC) mechanism:

◊ *Model:* communications towards active objects appear syntactically programmed as member function calls.

The syntax of an IPC is unified with a standard call:

Ⓢ *Syntax:*

```
p -> f ( parameters );
```

This idea, introduced by the Actors model [3,30], means that what is sometimes called a process entry point is identical to a normal routine or member function.

While this idea is widely used in parallel object-oriented systems, there are many differences in the definition of the semantics of method-based IPC. In `C++//`:

◊ *Model:* communications are asynchronous between processes.

Function calls towards passive objects retain the synchronous semantics of standard C++. This choice encourages parallel execution of objects, and makes each process code more independent and self-contained. As we will see further, it is also very important for supporting reusability. Synchronous function call is also possible in `C++//`, but must be specified explicitly in either the function call or the process definition.

Systematic asynchronous IPC structures a `C++//` system into independent asynchronous *sub-systems*: all the communications between sub-systems are asynchronous. Figure 1 demonstrates five sub-systems.

2.4. **Synchronization**

Asynchronous communication can be difficult for programmers to manage. For instance, since even function calls to processes are asynchronous, before using result values one usually needs to explicitly add synchronizations in order to make sure they have been returned by the processes. Commonly, such models lack synchronization. We use a simple rule to address this drawback: *wait-by-necessity.*

◊ *Model:* a process is automatically blocked when it attempts to use the result of a parallel member function call that has not yet been returned.

Thus, a caller does not wait for the result of an asynchronous function call until that value is explicitly used in some computation. Should a value not have been returned at that point, the caller is automatically blocked until the value becomes available. This mechanism implicitly synchronizes processes; the two primitives `Wait` and `Awaited` are provided for explicit synchronization.

Ⓢ *Syntax:*

```
v = p->f(parameters);
⋮
v->foo();                // Automatically triggers a wait
                         // if v is awaited
⋮
if (Awaited(v)){        // test the status of v
  ...
}

Wait(v);                 // explicitly triggers a wait
                         // if v is awaited
⋮
obj->g(v);               // no wait if pointer access
v2 = v;
```

Program 1.1. Wait-by-necessity.

Program 1.1 summarizes the semantics of wait-by-necessity. The result of a function call not yet returned is called an *awaited object*. Our semantics define that no wait is triggered by assigning a pointer to such an object to another variable, or by passing such a pointer as a parameter. A wait occurs only when the program accesses the awaited object itself (which is syntactically a pointer access to the object) or transmits (copies) the object to another process.

Wait-by-necessity is a form of future [28], and is related to concepts found in several other languages: the `Hurry` primitive of Act1 [33], the `CBox` objects of ConcurrentSmalltalk [42], and the future type message passing of ABCL/1 [43]. However, an important difference is that the mechanism presented here is systematic and automatic, which is reflected in the absence of any special syntactic construction. This has a strong impact on reusability.

In order to avoid the run-time overhead involved in the implementation of wait-by-necessity, it is possible to avoid implicit synchronization and use the explicit synchronization primitives instead. This is a tradeoff between programming ease and reusability on one hand, and efficiency and speedup on the other.

2.5. **Sharing**

If two processes refer to the same object, method calls to that object may overlap, which raises all of the problems usually associated with shared data. To address this issue, each non-process object in `C++//` is a private object, and is accessible to only one process:

◊ *Model:* there are no shared passive objects.

We say that a private object belongs to its process's sub-system. The programming model ensures the absence of sharing:

◊ *Model:* the semantics of communication between processes is a copy semantics for passive objects.

- A process is an active object, sequential and single-threaded.
- Communications between active objects are syntactically programmed as member function calls, and are asynchronous.
- Wait-by-necessity: a process is automatically synchronized, *i.e.* it waits, when it attempts to use the result of a member function call that has not been returned yet.
- There are no shared passive objects.
- The semantics of communication between processes is a copy semantics for passive objects.

FIGURE 3. Basic features of the C++// model.

All parameters are automatically transmitted by copy from one process to another. A deep copy of the object is achieved: when an object o is copied, all the objects referred to by pointers in o are deep copied as well. The implementation automatically and transparently handles the marshalling of data and pointers implied by this, as well as circular object structures:

Ⓢ *Syntax:*

```
p -> f ( parameters ); // passive objects are automatically
                       // passed by copy between processes.
```

Processes, of course, are always transmitted by reference.

Figure 1 shows how shared objects do not appear in C++// programs. Each passive object is accessible to exactly one active object; each of the five sub-systems in this program consists of one active object and all the passive objects it can reach. The arcs labelled (1) and (2) are always activated as asynchronous communication (IPC), while the arcs labelled (3) to (6) are activated as normal function calls. As a consequence of the absence of shared objects, synchronization between sub-systems only occurs when one sub-system waits for a result value from another process.

Prohibiting shared data has also important methodological consequences. The absence of shared objects allows either an immediate reuse (through the automatic copy), or the identification of new processes to program in order to implement the shared objects. Finally, due to the absence of interleaving, it helps ensuring correctness of such parallel applications derived from sequential ones.

As we finished the basic characteristics of the programming model, Figure 3 summarizes them.

The model has some limitations: in order to be able to use polymorphism between standard passive objects and process objects, all public functions have to be virtual, otherwise, the non-virtual function calls will not be transformed into IPC. This drawback can be alleviated with C++ compilers providing "*all-virtual*" option; here there is a choice between paying the price of all virtual functions, and reusability. Of course, this feature requires recompiling all files involved, but is probably a small price compared to the reuse that can be obtained.

3. Control programming

So far, we have only examined and defined the features of C++// which deal with the global aspects of the programming model, such as the nature of processes and their interactions. This section describes how the control flow of processes is specified, *i.e.* how behavior, communication, and synchronization of active objects are programmed.

3.1. **Centralized and explicit control**

Control can be decentralized, that is, distributed throughout a program or, alternatively, control can be centralized, *i.e.* gathered into one place in the definition of a process, independently of the function code.

Decentralized control makes reuse of function code difficult for two reasons. First, functions designed in a sequential framework cannot be reused in a parallel one just as they are, as elements of control must be added

to them. Second, when a new process class is obtained through inheritance, the new class often needs to change the synchronization scheme used in the original class. If control is embedded in function bodies, this may not be feasible. This leads to the following choice:

◊ *Model:* processes have a centralized control.

It allows function reuse for both sequential and process classes without changes to the bodies of such classes.

Program 1.2 presents partial code for the library class `Process`. After creation and initialization, a process object executes its `Live` routine. This routine describes the sequence of actions which that process executes during its lifetime. The process terminates when the `Live` routine completes.

Ⓢ *Syntax:*

```
class Process {
 public:
  Process (···){            // process creation
    ⋮
  }

  virtual void Live(){      // process body
    ⋮  // default FIFO behavior
  }
  ⋮
};
```

Program 1.2. The `Process` class.

Another design decision that must be made in concurrent object oriented systems is whether process control is implicit or explicit. Control is explicit if its definition consists of an explicitly programmed thread of control. Otherwise, control is implicit, which in practice usually means that it is declarative.

Our argument is that (see [16] for a complete discussion):

1. sometimes programmers need explicit control;
2. implicit control permits reuse of synchronization;
3. no universal implicit control abstraction exists; and
4. explicit control allows us to build implicit control abstractions.

As a consequence, the basic mechanism for programming process behaviors in `C++//` is:

◊ *Model:* explicit control.

Explicit control programming consists of defining the `Live` routine of the `Process` class and its heirs (Program 1.2) using the sequential control structures of C++. All of the expressive power of C++ is available, without any limitation. For instance, the process body of a bounded buffer can be defined as in Program 1.4.

Besides explicit control, other features are needed in order to construct abstractions for concurrent programming. These features permit `C++//` to explicitly service requests. First, defining a process's thread of control often consists of defining the synchronization of its public member functions. Since such an activity requires dynamic manipulation of C++ functions, we need:

◊ *Model:* member functions as first class objects.

In practice, only some limited features, such as the ability to use routines as parameters, and system-wide valid function identifiers, are needed.

To fill that need, we provide the function `mid()` to return function identifiers. Its usage is:

Ⓢ *Syntax:*

```
member_id f;

f = mid(put);
f = mid(A::put);
f = mid(A::put, A::get);
f = mid(A::put(int, P *));
```

In order to deal with overloading, this function returns either a single identifier, or a representation of all adequate functions.

In the same way, because we need to explicitly program request servicing, we must be able to manipulate requests as objects (*i.e.*, to pass them as parameters of other functions, to assign them to variables, and so on). We require:

◊ *Model:* requests as first class objects.

In `C++//`, a particular class (`Request`) models the requests; every request is an instance of this class.

Finally, to be able to fully control request servicing, programmers must have

◊ *Model:* access to the list of pending requests.

This is given through the `Process` class, with a specific member named `request_list` that contains the list.

With these three facilities in place, it is possible to program the control of processes in diverse and flexible ways.

3.2. **Library of service routines**

Service primitives are needed to allow programmers to program control explicitly. Usually, programmers are given only a few such primitives, mainly because they are made part of the language itself as syntactical constructions (*e.g.*, the `serve` instruction of Ada). With the primitives we define, it is possible to program a complete library of service routines [13]. Some of these are shown in Program 1.3, where `f` and `g` are member identifiers obtained from the function `mid()` introduced in the previous section.

Ⓢ *Syntax:*

```
// Non-blocking services
serve_oldest();        // Serve the oldest request of all
serve_oldest(f);       // The oldest request on f
serve_oldest(f,g, ···);   // The oldest of f or g
serve_flush();         // Serve the oldest, wipe out the others
serve_flush(f);        // The oldest on f
⋮
// Timed blocking services: block for a limited time only
tm_serve_oldest(t);    // Serve the oldest, wait at most t
tm_serve_oldest(f,t);  // The oldest request on f
⋮
// Waiting primitives
wait_a_request();      // Wait until there is a request to serve
wait_a_request(f);     // Wait a request to serve on f
```

Program 1.3. A library of service routines.

These functions are defined in the class `Process`, and can be used when programming the `Live` routine. There is no limitation in the range of facilities that can be encapsulated in service routines. Timed services

are an example of such expressiveness; selection based on the request parameters is another. Moreover, if a programmer does not find the particular selection function he needs, he is able to program it. Thus, libraries of service routines specific to particular programmers or application domains can be defined.

Another important point concerns efficiency: concurrent policies are determined within the context of each process, based on local information rather than by using IPC, avoiding problems like polling bias [27]. This is an important advantage in distributed programming.

```
class Buffer: public Process, public List {
  :
 protected:
  virtual void Live()
  {
    while (!stop){
      if (!full)
        serve_oldest ( mid(put) );
      if (!empty)
        serve_oldest ( mid(get) );
    }
  }
};
```

Program 1.4. An explicit bounded buffer example.

As an illustration of the use of explicit control programming, Program 1.4 presents a `C++//` implementation of a bounded buffer. This definition implements a specific policy: when the buffer is neither `full` or `empty`, the buffer alternates service on `put` and `get`. This policy is clearly not the only possible one.

This is an example of explicitly fine-tuning the synchronization of processes. While this might be very important in some contexts, we might want to program within a more abstract framework in others, ignoring the implementation details, through the definition of libraries of abstractions [16].

As we finished the control programming of processes, Figure 4 summarizes the basic features on the `C++//` model.

- Processes have a centralized and explicit control
- Member functions and requests are first class objects.
- The list of pending requests is accessible.
- A library of service routines provides for explicit control programming.
- A library of abstractions allows for implicit and declarative control.

FIGURE 4. Control programming in `C++//`.

4. PROGRAMMATION, ENVIRONMENT AND IMPLEMENTATION

4.1. **A programming method**

Because it is rather difficult to evaluate performances of distributed system before they actually run, we believe that definition of processes has to be postponed as much as possible, and should be flexible and adaptable. The programming guide we develop in this section applies this principle, made possible by the features of the `C++//` model.

The first step is a standard, sequential, object-oriented design, possibly including object identification, interface and topology design, and sequential implementation [9, 35]. The next steps deal with parallel design and are specific to the language and technique we developed.

Processes are a subset of classes. Because object-oriented design usually gives a finer-grained decomposition than structured design or information hiding methods (because every type is a module, and every module is a type), there is no need for re-structuring. The classes remaining passive are commonly used without any changes.

Here is an example of how an entity `a` declared:

```
A* a;
```

get assigned with a process object of type `P_A` (heir of `A`):

```
a = new P_A ( ··· );
```

A function call on `a` is now executed on an asynchronous basis (the caller does not wait for its completion). This automatic transformation of synchronous call into asynchronous one is crucial to avoid routine redefinition. An inherited routine may use the result of a function call issued to the process:

```
res = a->fct( parameters );
⋮
res->g( parameters);
```

In this case, the *wait-by-necessity* handles the situation. Without this automatic *data-driven synchronization* one would have to redefine the current routine in order to add explicit synchronization.

These model properties ensure that most of the inherited routines remain valid for the process class. However, some special cases need re-programming.

Figure 5 presents the significant steps of the method.

1. Sequential design and programming.
2. Process identification.
3. Process programming:
   - Define each process class (`Process` or abstraction class).
   - Define the activity (`Live`).
   - Use the process classes with polymorphism.
4. Adaptation to constraints:
   - Refine the topology.
   - Define new Processes.

FIGURE 5. The 4 steps of the method.

4.2. **Environment**

This section briefly describes the facilities supporting the development of `C++//` programs, including the compilation of source code, executable generation (see Fig. 6), and more specifically a mechanism for mapping active objects onto machines.

Mapping assigns each active object created during the execution of a `C++//` program to an operating system process on an actual machine or processor. In order to avoid confusion, we call the sub-system consisting of one active and all its passive objects a *language process*, and use the term *OS process* for the usual notion of an operating system process.

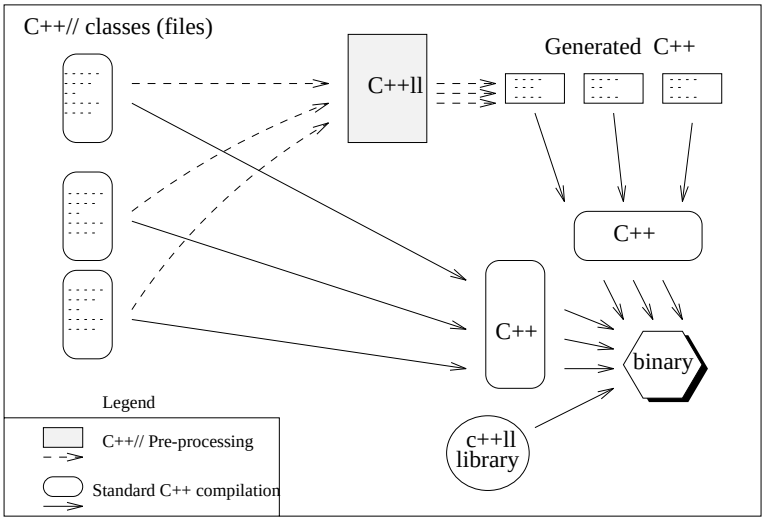

FIGURE 6. Compilation of a C++// system.

The mapping of a language process to an OS process on a particular processor is controlled by the programmer through the association of two criteria:

◊ *Model:*
(1) the machine where the language process is to be created;
(2) the light-weight or heavy-weight nature of the language process.

The machine itself can be specified in two ways. The first method is to specify a virtual machine name, which is simply a string. This name is related to an actual machine name by a translation file called `.c++ll-mapping`. The C++// system looks for this file first in the directory in which the process is running, and, if it is not found there, in the user's home directory. An example of such file is:

⊡ *File:*

```
FILE .c++ll-mapping

// virtual name   actual name
   Server         Inria.Sophia.fr
   S1             wilpena.unice.fr
   S2             192.134.39.96
   S3             // current machine
   P1             I3S-1
   ⋮              ⋮
   P6             INRIA-1
```

The other technique used to specify a machine is to use a language process that already exists. In this case, the new process is created on the machine where that language process is running. With this technique, processes can be linked together to ensure locality.

The *light-weight* switch permits creation of several language processes inside a single OS process. In the *heavy-weight* case, only one language process is mapped to each OS process. The user accesses these switches

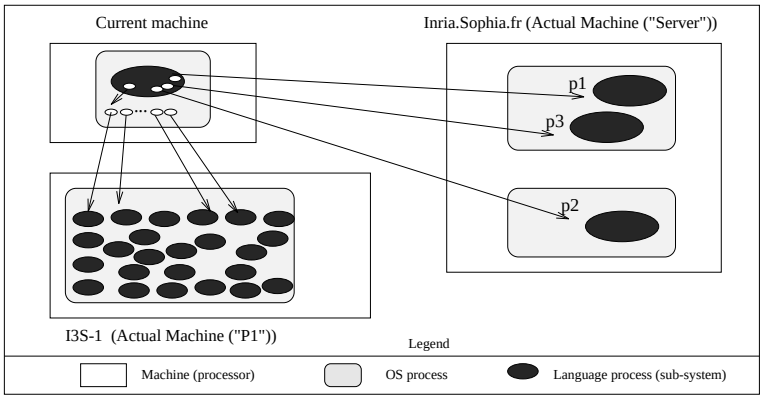

FIGURE 7. Example of mapping.

through a class called `Mapping`:

Ⓢ *Syntax:*

```
class Mapping {
 public:
  virtual void on_machine(const String& m); // set a virtual machine name
  virtual void with_process(Process* p);    // set the machine to be the same as
                                            // for the already existing process p
  virtual void set_light();                 // set to light-weight process
  virtual void set_heavy();                 // set to heavy-weight process
};
```

When a program creates a language process, an object of type `Mapping` can be passed to `new` in order to specify the desired mapping of the new process. Program 1.5 presents the syntax used for this. With the `.c++ll-mapping` file taken from above, Program 1.5 produces part of the mapping presented in Figure 7.

∇ *Example:*

```
A *p1;                // A is a normal sequential class: instantiation style
P_A *p2, *p3;         // P_A is a process class: class based style
Mapping *map1, *map2; // mapping objects
⋮
map1->set_heavy();
map1->on_machine("Server");
p1 = (A *) new (typeid(A), map1) Process_alloc(···);
                      // p1 on a new OS process,
                      // on machine with actual name "Server"
map2->set_heavy();
map2->with_process(p1); // p2 on a new heavy-weight process,
p2 = new (map2)  P_A(···); // same machine as p1
···
map2->set_light();
p3 = new (map2) P_A(···); // p3 on a light-weight process,
                          // same machine as p1,
                         // inside the same OS process as p1
```

Program 1.5. Mapping processes to machines.

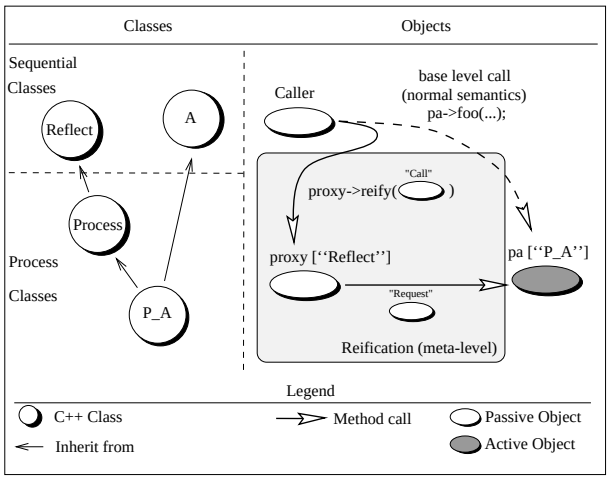

FIGURE 8. Reification of calls.

4.3. **Implementation**

This presentation goes beyond implementation details since the technique we use—reification—also provides for customization and extension of our system.

4.3.1. *A reflection-based system*

The `C++//` system is based on a Meta-Object Protocol (MOP). There are various MOPs [32], for different languages and systems, with various goals, compilation and run-time costs, and various levels of expressiveness. MOP techniques have been used in many contexts in particular for parallel and distributed programming [11, 20, 34].

Within our context, we use a reflection mechanism based on reification. Reification is simply the action of transforming a call issued to an object into an object itself; we say that the call is "reified". From this transformation, the call can be manipulated as a first class entity, *i.e.* stored in a data structure, passed as parameter, sent to another process, etc.

A *meta-object* (Fig. 8) captures each call directed towards a normal *base-level* object; a meta-object is an instance of a *meta-class*. In some ways, a *proxy*, a local object that permits to access a remote one [8, 22, 39], is a kind of meta-object.

4.3.2. *A* MOP *for* `C++`*: basic classes*

The main principle of our MOP for C++ is embodied in a special class, called `Reflect`, which presents the following behavior:

◊ *Model: all* classes inheriting publicly from `Reflect`, either directly or indirectly, are called *reified classes*, a reified class has *reified instances*; all calls issued to a reified object are reified.

This last requirement is important for reusability, as it permits users to take a normal class, and then globally modify its behavior, to transform it into a process.

Figure 8 illustrates reification. The creation of an instance of a `Reflect` class returns a meta-object (a proxy) for the type being passed in as the allocator's first parameter. From this mechanism, we implement the basic classes of our programming model described in Section 2.1.

4.3.3. *Customization and extension of* `C++//`

The MOP we just presented is independent of any parallel programming model. The classes of the MIMD model (such as `Process`) we described in this paper are programmed on top of the MOP, without any compiler

modification. An important consequence of this is that other parallel programming models, such as shared-memory MIMD or SPMD, can be defined on top of the MOP. The wait-by-necessity implementation, for instance, is achieved through a class `Future` which uses reification by inheriting from `Reflect`. Such an open system, or open implementation [32], is extensible by the end-user or by developers of new libraries, and adaptable to various needs and situations, such as the ones presented in the two following sections.

Notice that this MOP has been adopted as the level 0 of a standard framework for parallel C++ systems, designed in the context of the EUROPA Parallel C++ working group, formerly funded by the EU [17]. In this framework, other C++ libraries for parallel computing, such as UC++ (featuring constructs which are variations of those found in `C++//`, like active objects, asynchronous communications and futures) were implemented on top of this MOP. A similar effort in the United States is HPC++, and now, OpenHPC++ [24], which supports constructs to develop both task as well as data parallel applications in C++. [26] explains how it would be possible to implement the parallel STL model of HPC++ on top of the MOP adopted within the EUROPA working group.

They are some other few extensions of C++ for parallel computing [41] that are built upon reification mechanisms, in particular, onto the meta-object protocols OpenC++ [20] and MPC++ [31].

5. Sharing passive objects among active objects

This section presents a mechanism for sharing objects [19] when two active objects that reside in the same address space (in the same process) want to access the same passive object in read mode.

5.1. **The SharedOnRead framework**

A crucial point of the standard `C++//` model, is where active objects are created. A `C++//` programmer has several choices to determine the machine (or node) where a new active object will be created: (1) give the machine name, (2) provide an existing active object in order to use the same machine. But central to the issue is that, in both cases, the programmer has two options to create the new active object: (a) in an existing address space, (b) in a new address space. In case (a), several active objects will be able to share the same address space — threads belonging to a same heavy-weight process are used to implement active objects. Let us note that even when active objects in the same address space communicate, passive objects are still transmitted by copy. This potentially time and space consuming strategy is mandatory if we want the program semantics to be constant, whatever the mapping is. However, in some cases, sharing is actually possible, and copying large objects could be avoided. The SharedOnRead mechanism was defined to make possible such optimization, and to provide a general strategy that keeps the semantics unchanged when the mapping varies.

5.1.1. *Strategy*

Upon a communication between two subsystems that are within the same address space, instead of copying a parameter if it is of type SharedOnRead, we just share it; otherwise, there is no alteration of the copy semantics if the two subsystems are not in the same address space (*cf.* Fig. 9). The SharedOnRead is dearly related to the *copy-on-write* mechanism that can be found in operating system (Mach [37] or Orca [6, 29] are using it). However, the strategy is slightly different in copy-on-write techniques: one wants to copy only when it is necessary, instead, with the SharedOnRead mechanism, one wants to share data on read operations whenever it is possible (*i.e.* same address space).

While this idea is quite simple, a mechanism is needed in order to maintain the copy semantics of `C++//` that should apply everywhere, even when the two subsystems are mapped in the same address space. We can notice that objects will be able to be shared by several subsystems only as long as they are not modified by one of the subsystems. In order to be accurate, the strategy needs to make a distinction between *read* and *write* accesses, and also needs to know when a subsystem *forgets* a SharedOnRead object (the subsystem suppresses its reference to this object).

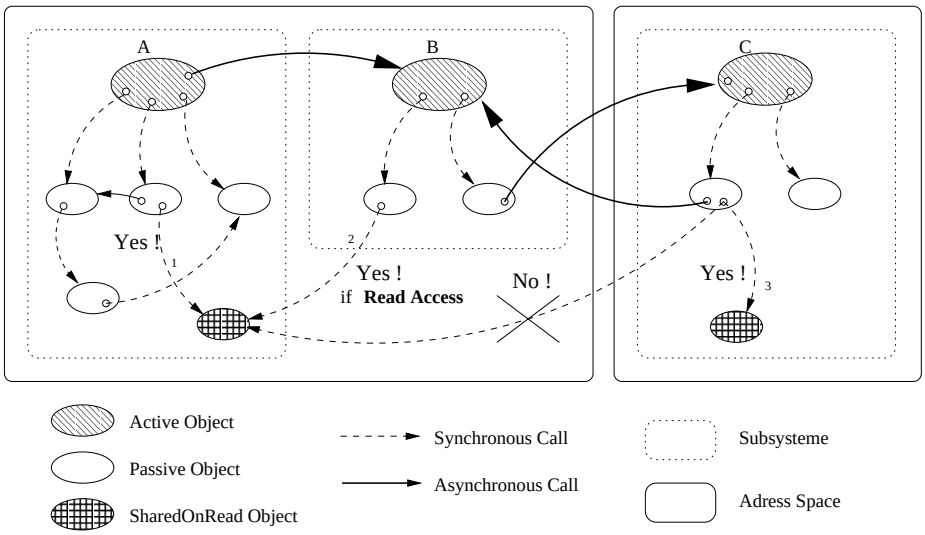

FIGURE 9. SharedOnRead objects within several address spaces.

The requirements at the design level are the following:

◊ *Model:*
(1) When a SharedOnRead object is used as a communication parameter between two subsystems being in the same address space, the original object is not copied, but instead a new reference is memorized (a counter is incremented within that object).
(2) A *read* access is freely achieved (from both the owner's subsystem or another one).
(3) Upon a *write* access (from both the owner's subsystem or another one), if the counter value is more than 1, a copy of the object is made. The modification applies to the copy. The counter of the previously existing object is decremented; the counter of the copy is set to 1.
(4) Upon a *forget* operation, the counter is just decremented. When reaching zero, the object is automatically garbaged.

5.1.2. *Programmer interface and implementation*

As a design decision, we choose to give users the control over which objects should be SharedOnRead and which should have the standard systematic copy behavior.

◊ *Model:* A SharedOnRead object is an instance of a class that inherits directly or indirectly from the `C++//` SharedOnRead class.

The SharedOnRead class is:

Ⓢ *Syntax:*

```
class SharedOnRead : public Reflect {
public:
    virtual void read_access(mid_type);
    virtual void write_access(mid_type);
    virtual void access_declaration();
    virtual void forget();
};
```

and as such provides several member functions whose usage is now described. `read_access` and `write_access`, are both used to specify how the data members are accessed by a given method. If `read_acess` is selected by

the programmer for a given method, the programmer declares that, for this function, data members are never changed, alas if `write_access` is selected, data members can be changed. These two functions take a `mid_type` parameter which is unique for all the member functions in the program; the `mid` function (see Sect. 3.2) provides that unique identifier for a function name. The `access_declaration` function has to be redefined in order to specify for each public member function its read or write nature: write access is the default behavior. A SharedOnRead user has to take care of that and it is his responsibility to check for each method in the class which ones are leaving the object in the same state and which ones are making modification to the object state. Lastly, `forget` must be called so as to declare that this SharedOnRead object is not used anymore. This have to be dealt with explicitly because SharedOnRead objects cannot be aware that they are not referenced anymore.

Program 1.6 gives an example of the use of the SharedOnRead mechanism, in which the programmer has just to define a new class that inherits from an existing class and the SharedOnRead one, the only additional programming work being to redefine the `access_declaration` function.

∇ *Example:*

```
class Block { // A Matrix block
public:
    virtual void reach(int** ai, int** aj, double** a);
    virtual void update(int**& ai, int**& aj, double**& a);
};

class SorBlock: public SharedOnRead, public Block {
public:
    SorBlock();
    virtual void access_declaration() {
       read_access(mid(reach));
       write_access(mid(update));
    }
};
```

Program 1.6. Programming a SharedOnRead block in a matrix-based example.

The implementation of the SharedOnRead class is based on the reification mechanism provided by `C++//`. Inheriting from the class `Reflect`, all the SharedOnRead member functions are reified: the functions are not directly executed but are derouted in a specific *proxy* where all the necessary implementation is defined and achieved (update of the counter, copy when necessary, etc.). After this step, the *proxy* executes the function.

5.2. Benchmark application

5.2.1. *Parallel linear algebra*

We have tested the SharedOnRead mechanism on basic linear algebra operations commonly used in iterative methods [38]. The key point for efficient parallel implementation of iterative methods are good performance of the distributed sparse matrix/vector product, and distributed dot product since these operations are at the heart of all basic Krylov algorithms [36]. In this context it is crucial to avoid unnecessary copies in matrix operations.

As defined in [36] we may focus on a reduced set of operations:

- dense SAXPY, $Y := \alpha X + \beta Y$ with $Y, X \in \mathbb{R}^{n \times p}, \alpha, \beta \in \mathbb{R}$;
- dense or sparse matrix product $Y := \alpha A \cdot X + \beta Y$ with $Y \in \mathbb{R}^{m \times p}, A \in \mathbb{R}^{m \times n}, X \in \mathbb{R}^{n \times p}, \alpha, \beta \in \mathbb{R}$.

Parallel numerical linear algebra is concerned with data distribution of the matrix arguments in the above operations. In our case we will only consider a block distribution scheme of matrices which is widely used [21] and well suited for these applications. With these kind of distributions, we have to do dense matrix saxpy operations and dense or sparse matrix products.

These operations are implemented as a method of a class `Matrix` representing a block partitioned matrix. The methods are:

- `Matrix::scal(double alpha, Matrix* B, double beta)`
  This method performs the matrix saxpy operation $this = \beta \cdot this + \alpha \cdot B$.
- `Matrix::axpy(double alpha,Matrix* A,Matrix* X,int mm)`
  This method performs the matrix operation $this = \alpha \cdot A \times X + this$, and $this = \alpha \cdot A \times X$ if `mm==1`.

Here, the distributed objects are the blocks of the matrix which may be called CSC since they are Compressed Sparse Column (potentially sparse) matrices.

5.2.2. *From sequential to parallel matrices in `C++//`*

A sequential matrix contains a list of CSC objects, each of these objects holding a `Block`. This `Block` is responsible for allocating all the arrays representing the matrix. If we want to parallelize these classes using C++//, we only have to redefine the `Matrix` constructors. These constructors create the distributed CSC objects of type `Block` or `SorBlock` (see Program 1.6) depending if we want to use the SharedOnRead mechanism or not. A distributed CSC object (`CSC_ll` object) can be created just by inheriting from CSC and the `C++//` class `Process`. All the functions presented above (scal, axpy, ...) come unchanged from the sequential classes.

Program 1.7 presents the sequential version for a dense axpy function, which will be reused unchanged in the `C++//` version. Thanks to polymorphism compatibility, the two A and X variables can be `CSC_ll` objects. If the SharedOnRead is used in the matrices construction, the `block()` function returns a `SorBlock` object: as such, as `bl1` and `bl2` must be accessed only in read mode, we will use them directly if they are located in the same address space, without generating any copy. On the contrary, the `mine` variable, which represents the local `Block` of the `CSC` object, will be modified, so `update` has been declared in write mode in Program 1.6.

```
void CSC::axpy(double alpha, CSC* A, CSC* X, double beta) {
   Block* bl1 = A->block();
   Block* bl2 = X->block();
   Block* mine = block();
   bl1->reach(&tia, &tja, &ta);
   bl2->reach(&tix, &tjx, &tx);
   mine->update(&tiy,&tjy,&ty);
   ...
}
```

Program 1.7. Sequential dense CSC_Block product.

5.2.3. *The MPI version*

Our objective is to apply the SharedOnRead mechanism and prove that this yield to performances as good as the ones obtained with MPI, but with more transparency and flexibity for the programmer due to the object-oriented model of programming. The MPI [2] implementation requires to redefine all the functions in order to take into account the fact that not the same operations must be executed depending on the processor they are executing on: MPI is very intrusive. This means that the programmer has to add a lot of MPI calls in the original sequential code in order to derive the parallel version. Moreover, this makes it difficult to the programmer to go back and forth from the sequential to the parallel version. Furthermore, MPI is a message passing library thus parallelism is explicit, and the programmer has to directly deal with distribution and communication.

5.2.4. *Performances*

The following tests were performed on a network of 4 Solaris Ultra 1 with 128MB of memory and a 10Mb/s Ethernet link. The MPI tests use the LAM library (`http://www.mpi.nd.edu/lam/`).

Since the runtime is not based on a virtual shared memory, the standard distribution algorithm can imply that several active objects (representing CSCs holding matrix blocks) get mapped within the same address space on each workstation, while others get mapped within different address spaces. Recall that the SharedOnRead

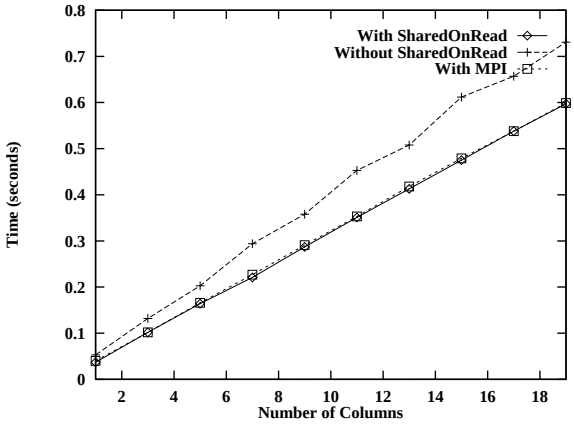

FIGURE 10. SAXPY with 4 computers.

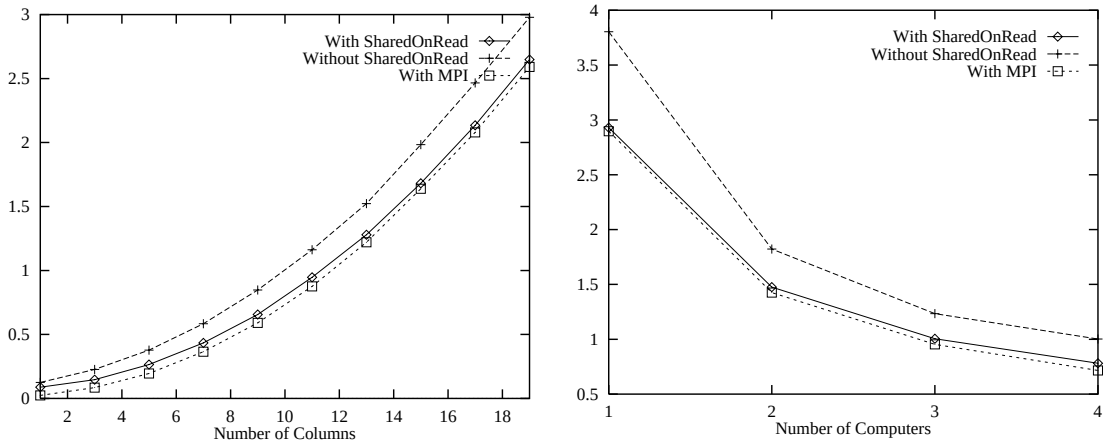

FIGURE 11. Dense matrix product (a) duration in seconds using 4 computers (b) speed-up.

optimization applies when a computation occurs within the same address space. As demonstrated below this is sufficient to achieve consequent speedup. If we were on an SMP architecture, then the benefits would be even greater since there would be opportunity for sharing all the matrix blocks.

Figure 10 presents the performances for a `scal` (*cf.* Sect 5.2.1) calculation. Matrices used during these tests were rectangular matrices with 90449 rows and a variable number of columns. The use of SharedOnRead objects demonstrates a speed-up between 20 and 25% compared to the non optimized `C++//` version. When compared with the MPI version, we cannot distinguish any difference between `C++//` with SharedOnRead and MPI. One important point to notice is the fact that the non optimized `C++//` version (without SharedOnRead objects) presents more and more overhead when the matrix size increases. The main reason is that this version requires many communications between the different active objects even if they are located in the same address space. In the SharedOnRead version, the two blocks represented by two active objects are mapped within the same process, so communications and copies are avoided.

Figure 11a presents performance results for a dense matrix product. Again, the performances of the experiment using the non optimized `C++//` version is between 20 and 30% slower than the MPI one. Between the

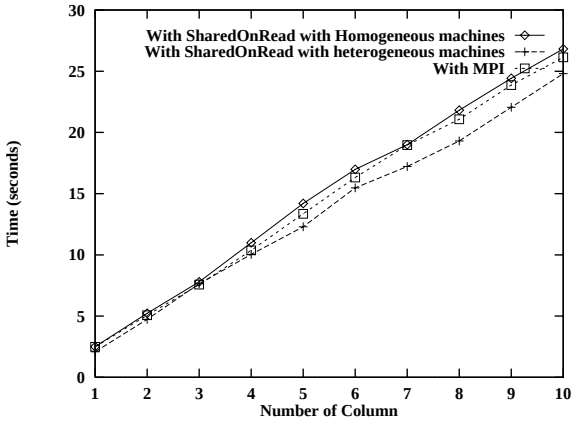

FIGURE 12. Sparse matrix product with 4 computers.

C++// version and the MPI one, the overhead is constant. But the SharedOnRead version and the MPI one do not behave exactly the same: the C++// solution sums the local matrix in a sequential way because it reuses the sequential code, whereas the MPI version requires communication during the reduction step. Figure 11b presents the speed-up obtained with 4 computers for the 3 different experiments. All calculations on dense matrices are perfectly scalable; with 4 computers, the speed-up is around 3.9. We can observe that the overhead of the non optimized C++// version is constant whatever the number of computers we use.

At last, Figure 12 presents performance results for a sparse matrix product. A first point to notice in such a case is that the *add* function of the matrix had to be rewritten in the C++// version: the reduction being critical in this benchmark, it was important to compute it in parallel. The second important aspect of this benchmark deals with the platform architecture. All the previous tests were made on homogeneous computers: the same CPU, at the same frequency, with the same amount of memory. This last test was performed on an heterogeneous platform. Two of the four Ultra 1 were replaced with two Ultra 2 with 192 MB of memory. Within this new architecture, the MPI program has the same performance as in the homogeneous test. In the C++// case, the benchmark demonstrates that the object-oriented version is actually more efficient than the MPI one. While MPI is subject to synchronization barriers, the C++// version automatically takes advantage of the asynchronous model. In that case, the reduction of the two local matrices of the fastest computers can start even if the computation on the slowest computers has not finished.

6. Overlapping communication with computation

This section presents the concepts and an implementation of an overlapping mechanism between communication and computation [7]. This mechanism allows to decrease the execution time of a remote method invocation, especially in the context of important transfers, such as matrices.

A general idea to lower communication costs is to overlap communication with computation, thus yielding to a pipeline effect regarding messages transmission. Any attempt to exploit this opportunity needs to rely on non-blocking elementary communications, such as for instance, asynchronous send and receive primitives as provided by well-known message-passing libraries (*e.g.* PVM [1] or MPI [2]).

For code readability and portability purposes, one additional requirement is to make the use of the overlapping technique as much transparent as possible for programmers. As such, we reject distributed hand programmed solutions where the programmer would himself split the data to be sent into smaller pieces, asynchronously send each piece in turn thus "feeding" the pipeline, while at the receiver side, explicitly and repetitively receive each new piece and goes on with it in the related computation.

Previous attempts to automatically make use of an overlapping mechanism between communication and computation have been successful in the context of data-parallel compiled languages for parallel architectures with distributed memory: HPF [10], FortranD [40], but also in LOCCS [23], a library for communication routines and computation. Here, we explain how the same problem has been tackled with, in the area of distributed object-oriented languages where the whole computation taking place on the distributed entities can be expressed as remote service invocations through method calls.

6.1. How the overlapping technique is designed and implemented

In the implementation of such remote method invocation-based settings, all arguments of the method call must generally be received before the method execution starts. The essence of our proposition is thus to:

◊ *Model:* apply a classical pipelining idea to the arguments of a remote call.

Once the first part of the arguments has arrived, the method execution will be able to start. Moreover, it is only the type of the arguments that will automatically indicate how to split the data to send. In this way, programmers will be able to express, at a very high level, opportunities to introduce an overlapping of communications with computation operations.

◊ *Model:* A new class, inheriting from `Reflect` (see Sect. 4.3.2) and called `later` is introduced in `C++//`, from which all objects that require to be sent latter have to inherit from.

Ⓢ *Syntax:*

```
class Matrix_Later : public later, public Matrix {...};
```

Objects from this `later` class must not be sent (eventually also, not be marshalled) during the first inspection of the objects belonging to the request, but later, each one in a new message (as would be done for $m2$ when calling `dom→ rang(m1,m2)` in Program 1.8 for example). `later` objects behave the same as `future` objects: automatic blocking when one tries to access to the value, transparent update of the object with the incoming value.

∇ *Example:*

```
class OpMatrix : public Process
  virtual int rang(Matrix *m1,
                   Matrix *m2)
    m1->square();
    m2->plus(m1);
    int res = m2->result();
    return (res);
  }
};
```

```
OpMatrix *dom =
    new ("host") OpMatrix(...);
Matrix *m1 =
    new Matrix(COLUMN, LINE);
Matrix *m2 =
    new Matrix_Later(COLUMN, LINE);
// set the values for m1 and m2
CLOCK_Call_Time_START;
int res = dom->rang(m1, m2);
CLOCK_Call_Time_STOP;
```

Program 1.8. Definition and use of a `C++//` remote service with a `later` parameter.

This technique applies whether objects of `later` type sit at the first level (*i.e.* they are parameters of the remote call as $m2$ in Program 1.8), or at lower levels (*i.e.* they are parts of non-`later` parameters; for example each line of a matrix could be declared `later` whereas the matrix itself not). Notice that if needed, it is possible to cast an object declared as inheriting from `later` to the original type (*e.g.* from `Matrix_Later` to `Matrix`), and vice-versa. For example, if a `later` object must be used at the very beginning of the next remote call, it would be worth to cast it now to its original type in order to send it immediately. So, to take advantage of the mechanism, the remote computation does not necessarily need a specific design (or redesign). The only

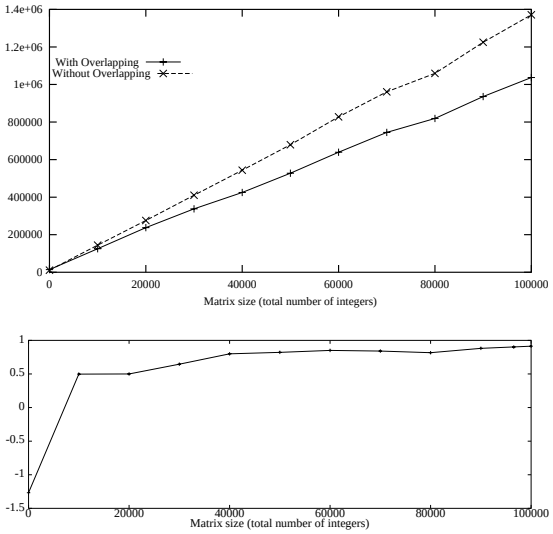

FIGURE 13. Execution of the remote service (caller side, total_duration in μs) and corresponding benefit (G) obtained from using the overlapping technique on a LAN.

important point is that the order the various parameters are first used should closely follow the order they are sent and received. So, the position of `later` parameters in method signatures becomes important.

Implementing the overlapping technique requires only minor modifications in the language runtime support. At the MOP level, the main modification is to write a new generic function to flatten requests: this function builds a first fragment which holds the request header and the non-`later` parameters, and then one fragment for each parameter of `later` type. Then at the runtime level, the first fragment is sent and its service will consist to create a `C++// future` type for each missing part of the request, *i.e.*, for each `later` part of the request parameters. Concerning the remaining fragments, they will be subsequently sent and served as follows: transparently update the corresponding awaited request parameters, *i.e.* the corresponding `future` objects.

6.2. **Benchmark**

6.2.1. *Description of the experiment*

We designed a simple test and benchmarked it. This test must not be considered as a real application, but as a means to validate the effectiveness of the technique. It is based on the remote call of the method `OpMatrix::rang(...)` (see Program 1.8) which takes two matrices, squares the first one, and adds the second one. As the second matrix `m2` is of type `Matrix_Later`, it can be used as a parameter of `OpMatrix::rang(...)`. The remote service can start as soon as the request id and the non-`later` parameters have been received. Experiments not using the overlapping technique are easily conducted: define `m2` as an instance of `Matrix` instead of `Matrix_Later`.

The technique should allow to overlap the transmission and reception of the `later` parameter (*i.e.* the matrix `m2`) that is only useful for the second part of the service execution (*i.e.* `m2→plus(m1)`) with the remote execution requiring only `m1` (*i.e.* the method `m1→square()`). Compared with an execution not using the overlapping technique, the duration of `m1→square()` (noted $d1$ in the following) should increase, since, at the same time, the remote processor has also to manage the reception and update of the matrix `m2`.

6.2.2. *Results*

Two Sun Solaris 2.6 workstations with 128 MB of RAM, interconnected by a 10 Mb/s Ethernet are used. The curves plotted in Figure 13 show a decrease of the `dom→ rang(m1,m2)` execution time, and an almost optimal gain as computed by G.

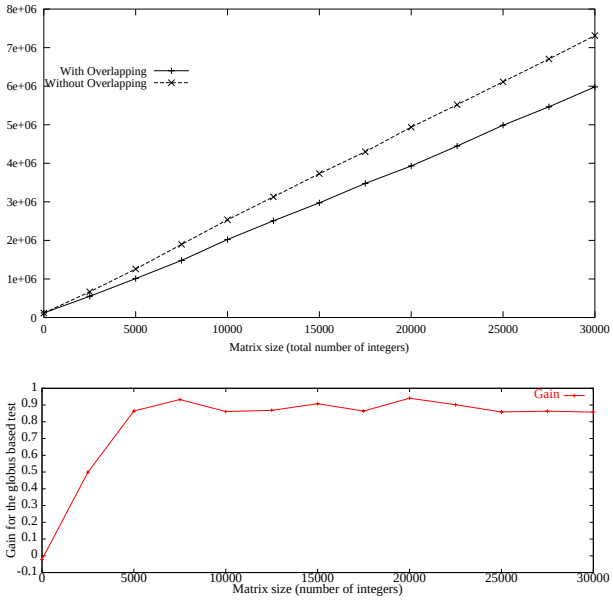

FIGURE 14. Execution of the remote service (caller side, total_duration in μs) and corresponding gain. This corresponds to one Globus-based test between USA and France during night period, with (d1) 300 times longer than in Figure 13.

Let us define the gain (G) in order to give a concrete estimation of the benefit.

$$G = \frac{duration_{not\_using\_overlap} - duration_{using\_overlap}}{later\_parameters\_transfer\_duration} \,. \tag{1}$$

The duration for transferring `later` parameters, *i.e.* `m2`, is estimated by sending a `C++//` object of the same size, not counting the—small—additional cost that would be required for managing a `later` parameter (a few milliseconds).

Scalability. The overlapping technique used in this context where lightweight processes are available, scales very well. Moreover, we deduce against our past experiences that only runtime supports using lightweight processes can scale so well. Indeed, benchmarks conducted in the context of `C++//` on top of PVM proved that the amount of data that could be sent and received while the remote service is in progress, is bounded by the remote receiving buffer size. The fundamental reason is that the transport-level layer can not gain the receiver process attention while this latter is engaged in a remote computation (*i.e.* `m1→square()`), due to the lack of a dedicated concurrent receiving thread.

WAN-based results. On WAN-based environments (see Fig. 14), sparing the transmission time of even a few bytes[1] yields a gain that the overhead of the technique can not override (very small compared to the high transmission delays). But, one should notice that the duration of the remote computation is of course an other crucial point. Indeed, if it is really too short compared with the transmission speed, almost no communication overlap occurs. This is why the Globus-based [25] grid experiment plotted in Figure 14 assigned $d1$ to be 300 times higher than in experiments plotted in Figure 13. In concrete situations, such a high-computation duration is not an unrealistic experimental assumption, as transmitting a large or even huge volume of data to remote computers (especially on a grid) is justified by the need to execute quite costly computations on these data.

[1]More precisely, the total duration for the test in Figure 14 using matrices $m1$ and $m2$ of 2500 integers decreases from 680 816 μs not using the overlapping technique to 556 446 μs when using it.

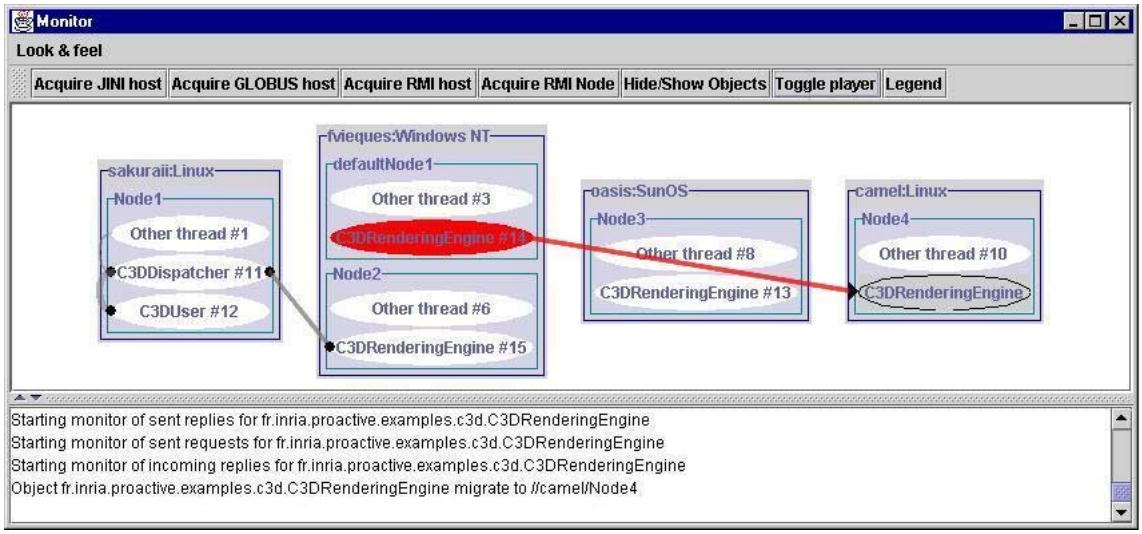

FIGURE 15. *Drag-and-drop migration* allows to graphically move objects between machines.

7. Conclusion and perspectives

The work presented here focussed on reuse, flexibility, and extendability. At different levels (service routines, abstractions for control programming, libraries defining specific programming models, etc.), the system we propose tries to be both abstract with information hiding principles (black boxes simple to use), and open for customization and extension. This approach tries to give some answers to the complexity and diversity of parallel programming.

Granularity is probably another crucial point of parallel programming. In order to reach performances, a challenging task is to achieve an appropriate matching between the granularity of program activities, and the capability of the underlying parallel architecture. We believe the reusability object-oriented languages make possible to be an important answer to the problem. In particuler, `C++//` makes it easy to turn an object into an active object or vice-versa, in order to adapt the granularity of program activities without not too much changes in the program.

In order to reach performances, two mechanisms have been described that can help to optimize parallel programming without too much a burden for the programmer: (1) the SharedOnRead that can help a distributed object-oriented language to be competitive with MPI, without writing large amount of code to obtain a parallel version; (2) a mechanism to overlap computations with communications in order to take advantage of pipelining in distributed object-oriented applications, without having to explicitly program the slicing of data and corresponding computations into smaller units. Both techniques apply in other distributed frameworks, such as Corba, Java RMI.

Specifically, in the framework of Java, we have defined and implemented the ProActive library [18] that offers a similar model with Java and its virtual machine. The framework being much more dynamic there are several new features that we are able to provide. First of all, it is not only possible to create remotely accessible objects, but also to "*turn active*" an existing object. Within the context of dynamic class loading, a JVM can receive an object for which the class was previously unknown, and the library make it possible to make this object remotely accessible. Another strong new feature is the *mobility of computations*. A migration primitive allows an active object to move from one machine to another, while maintaining functional all the remote references to and from itself. The graphical environment IC2D (Interactive Control and Debugging of Distribution) makes it possible to monitor and steer distributed and parallel applications. In Figure 15, the machines, the JVMs, and the active objects are graphically represented, as well as the communications that take place. Moreover, *drag*

and drop migration allows to move around objects at execution, potentially from one continent to another in a metacomputing framework.

Finally, another important aspect of distributed programming is the striking correctness problems it raises. This paper didn't address them, but it is another area of investigation for our group [4, 5]. We hope formal techniques, together with parallel object-oriented programming, will permit some advances in that matter.

References

[1] *Parallel Virtual Machine: a user's guide and tutorial for networked parallel computing*. MIT Press (1994).

[2] *MPI: The Complete Reference*. MIT Press (1998).

[3] G. Agha, *Actors: A Model of Concurrent Computation in Distributed Systems*. MIT Press (1986).

[4] I. Attali, D. Caromel and M. Oudshoorn, A Formal Definition of the Dynamic Semantics of the Eiffel Language, in *Sixteenth Australian Computer Science Conference (ACSC-16)*, G. Gupta, G. Mohay and R. Topor Eds., Griffith University, February (1993) 109–120.

[5] I. Attali, D. Caromel and M. Russo, Graphical Visualization of Java Objects, Threads, and Locks. *IEEE Distributed Systems Online* **2** (2001).

[6] H.E. Bal, M.F. Kaashoek, A.S. Tanenbaum and J. Jansen, Replication techniques for speeding up parallel applications on distributed systems. *Concurrency Practice & Experience* **4** (1992) 337–355.

[7] F. Baude, D. Caromel, N. Furmento and D. Sagnol, Optimizing Metacomputing with Communication-Computation Overlap, in *6th International Conference PaCT 2001*, number 2127, V. Malyshkin Ed., LNCS, 190–204.

[8] A. Birrell, G. Nelson, S. Owicki and E. Wobber, Network Objects. Technical Report SRC-RR-115, DEC Systems Research Center (1995).

[9] G. Booch, Object-Oriented Development. *IEEE Transaction on Software Engineering* (1986).

[10] T. Brandes and F. Desprez, Implementing Pipelined Computation and Communication in an HPF Compiler, in *Euro-Par'96*, number 1123, LNCS.

[11] F. Buschmann, K. Kiefer, F. Paulish and M. Stal, The Meta-Information-Protocol: Run-Time Type Information for C++, in *Proceedings of the International Workshop on Reflection and Meta-Level Architecture*, A. Yonezawa and B.C. Smith Eds. (1992) 82–87.

[12] D. Caromel, Service, Asynchrony and wait-by-necessity. *Journal of Object-Oriented Programming* **2** (1989) 12–22.

[13] D. Caromel, Concurrency: an Object Oriented Approach, in *Technology of Object-Oriented Languages and Systems (TOOLS'90)*, J. Bezivin, B. Meyer and J.-M. Nerson Eds., Angkor, June (1990) 183–197.

[14] D. Caromel, Concurrency and Reusability: From Sequential to Parallel. *Journal of Object-Oriented Programming* **3** (1990) 34–42.

[15] D. Caromel, Towards a Method of Object-Oriented Concurrent Programming. *Communications of the ACM* **36** (1993) 90–102.

[16] D. Caromel, F. Belloncle and Y. Roudier, The C++// Language, in *Parallel Programming Using C++*, MIT Press (1996) 257–296.

[17] D. Caromel, P. Dzwig, R. Kauffman, H. Liddell, A. McEwan, P. Mussi, J. Poole, M. Rigg and R. Winder, EC++ – EUROPA Parallel C++: A Draft Definition, in *Proceedings of High-Performance Computing and Networking (HPCN'96)*, Vol. 1067, LNCS, 848–857.

[18] D. Caromel, W. Klauser and J. Vayssiere, Towards Seamless Computing and Metacomputing in Java. *Concurrency Practice and Experience* (1998).

[19] D. Caromel, E. Noulard and D. Sagnol, Sharedonread optimization in parallel object-oriented programming, in *Computing in Object-Oriented Parallel Environments, Proceedings of ISCOPE'99*, LNCS, San Francisco, Dec (1999).

[20] S. Chiba and T. Masuda, Designing an Extensible Distributed Language with Meta-Level Architecture, in *Proceedings of the 7th European Conference on Object-Oriented Programming (ECOOP '93)*, O. Nierstrasz Ed., Springer-Verlag, Kaiserslautern, *Lecture Notes in Computer Science* **707** (1993) 482–501.

[21] J. Choi, J. Dongarra, S. Ostrouchov, A. Petitet, D. Walker and R.C. Whaley, A proposal for a set of parallel basic linear algebra subprograms. Technical Report Lapack Working Note 100, May (1995).

[22] A. Dave, M. Sefika and R.H. Campbell, Proxies, Application Interfaces and Distributed Systems, in *proceedings of the 2nd International Workshop on Object-Orientation in Operating Systems (OOOS), Paris (France)*, IEEE Computer Society Press, September (1992).

[23] F. Desprez, P. Ramet and J. Roman, Optimal Grain Size Computation for Pipelined Algorithms, in *Euro-Par'96*, number 1123, LNCS.

[24] S. Diwan and D. Gannon, Capabilities Based Communication Model for High-Performance Distributed Applications: The Open HPC++ Approach, in *IPPS/SPDP* (1999). `ftp://ftp.cs.indiana.edu/pub/sdiwan/capab.ps.gz`

[25] I. Foster and C. Kesselman, Globus: A metacomputing infrastructure toolkit. *International Journal of Supercomputer Applications* **11** (1997) 115–128.

[26] D. Gannon, S. Diwan and E. Johnson, HPC++ and the Europa Call Reification Model. *ACM Applied Computing Review* **4** (1996).
[27] N. Gehani, Concurrent Programming in the ADA Language: the Polling Bias. *Software–Practice and Experience* **14** (1984).
[28] R. Halstead, Multilisp: A Language for Concurrent Symbolic Computation. *ACM Transactions on Programming Languages and Systems*, October (1985).
[29] S.B. Hassen and H. Bal, Integrating task and data parallelism using shared objects, in *FCRC '96: Conference proceedings of the 1996 International Conference on Supercomputing: Philadelphia, PA, USA, May 25–28, 1996*, ACM Ed., ACM Press, New York (1996) 317–324.
[30] C. Hewitt, Viewing Control Structures as Patterns of Passing Messages. *J. Artificial Intelligence Res.* **8** (1977) 323–64.
[31] Y. Ishikawa, A. Hori, M. Sato, M. Matsuda, J. Nolte, H. Tezuka, H. Konaka, M. Maeda and K. Kubota, Design and implementation of metalevel architecture in C++ - MPC++ approach, in *Reflection'96*, April (1996).
[32] G. Kiczales, J. des Rivières and D.G. Bobrow, *The Art of the Metaobject Protocol*. MIT Press (1991).
[33] H. Lieberman, Concurrent Object-Oriented Programming in Act 1, in *Object-Oriented Concurrent Programming*, A. Yonezawa and M. Tokoro Eds., MIT Press (1987).
[34] P. Madany, N. Islam, P. Kougiouris and R.H. Campbell, Practical Examples of Reification and Reflection in C++, in *Proceedings of the International Workshop on Reflection and Meta-Level Architecture*, A. Yonezawa and B.C. Smith Eds. (1992) 76–81.
[35] B. Meyer, *Object-Oriented Software Construction*. Prentice-Hall (1988).
[36] E. Noulard, N. Emad and L. Flandrin, Calcul numérique parallèle et technologies objet. Technical Report Rapport PRISM 1998/003, ADULIS/PRiSM, Juillet (1997). Révision du 30/01/98.
[37] R. Rashid, R. Baron, A. Forin, D. Golub, M. Jones, D. Orr and R. Sanzi, Mach: a foundation for open systems (operating systems), in *Workstation Operating Systems: Proceedings of the Second Workshop on Workstation Operating Systems (WWOS-II), Pacific Grove, CA, USA, September 27–29, 1989*, IEEE Ed., IEEE Computer Society Presspages (1989) 109–113.
[38] Y. Saad, *Iterative Methods for Sparse Linear Systems*. PWS Publishing Company, New York (1996).
[39] M. Shapiro, Structure and Encapsulation in Distributed Systems: the Proxy Principle, in *Proceedings of the 6th International Conference on Distributed Computing Systems, Cambridge, MA, USA*, IEEE, May (1986) 198–204.
[40] C.W. Tseng, *An Optimizing Fortran D Compiler for MIMD Distributed-Memory Machines*. Ph.D. thesis, Rice University (1993).
[41] G. Wilson and P. Lu Eds., *Parallel Programming Using C++*. MIT Press (1996).
[42] Y. Yokote and M. Tokoro, Concurrent Programming in ConcurrentSmalltalk, in *Object-Oriented Concurrent Programming*, A. Yonezawa and M. Tokoro Eds., MIT Press (1987).
[43] A. Yonezawa, E. Shibayama, T. Takada and Y. Honda, Modelling and Programming in an Object-Oriented Concurrent Language ABCL/1, in *Object-Oriented Concurrent Programming*, A. Yonezawa and M. Tokoro Eds., MIT Press (1987).