INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

HOMA: automatic re-scheduling of multiple invocations in CORBA

Thierry GAUTIER — Hamid-reza HAMIDI

N° 5191

April 2004

Thème NUM

ISSN 0249-6399

Homa: automatic re-scheduling of multiple invocations in CORBA

Thierry GAUTIER*, Hamid-reza HAMIDI†

Thème NUM — Systèmes numériques
Projet APACHE supported by CNRS, INPG, INRIA, UJF

Rapport de recherche n° 5191 — April 2004 — 20 pages

Abstract: This report presents HOMA, an efficient and scalable CORBA-based code coupling environment. HOMA is composed by an IDL compiler and a runtime support. From IDL definitions of CORBA objects, HOMA compiler automatically extracts all the informations which allow efficient and scalable composition of method invocations. The compiler and runtime support rely on two functionalities: the automatic extraction of parallelism between method invocations and the lazy protocol used to communicate effective parameters. The runtime re-schedules the invocations and the associated communications using the parallelism deduced from their IDL definitions. The scheduling strategy is based on ATHAPASCAN. Used together these functionalities enable parallel communication between code coupled applications. This property is conserved by composition of invocations. The technology is based on the generation of extended client stub and server skeleton on top of standard client stub and server skeleton generated by existing IDL/CORBA compiler. Parallelism is handled by the parallel programming environment ATHAPASCAN. Thus, HOMA is highly portable. The target applications are high performance numerical simulations.

Key-words: Code coupling, CORBA, Parallel method invocations, On-line scheduling, Distributed cache, ATHAPASCAN

Laboratoire ID-IMAG

* thierry.gautier@inrialpes.fr
† hamid-reza.hamidi@imag.fr

Homa : ré-ordonnancement automatique de plusieurs invocations de méthode en CORBA

Résumé : Ce rapport présente HOMA, un environnement de couplage de codes qui permet la construction d'applications distribuées qui passent à l'échelle. HOMA est composé d'un compilateur IDL CORBA et d'un support exécutif en CORBA et ATHAPASCAN. Le compilateur de HOMA extrait automatiquement des définitions IDL toute les informations qui permettent d'exploiter efficacement la composition d'invocations de méthode. Le compilateur et le support exécutif exploitent deux fonctions : l'extraction du parallélisme entre plusieurs invocations de méthode et l'utilisation d'un protocole paresseux pour la communication des paramètres effectifs lors des invocations. Le moteur exécutif ré-ordonnance ces invocations et les communications associées en exploitant le parallélisme potentiel induit par les modes de passage des arguments des méthodes définis dans les IDL. Ce ré-ordonnancement se base entièrement sur ATHAPASCAN. Il est donc possible, à l'exécution, d'obtenir des communications parallèles entre des applications parallèles couplées par HOMA. De plus cette propriété est conservée par composition d'invocations. La technologie utilisée se base sur la génération de souches clientes et serveurs au dessus des souches générées par un compilateur standard CORBA et sur l'environnement de programmation ATHAPASCAN pour la gestion du parallélisme. Aucune modification d'un ORB existant n'est nécessaires ce qui rend HOMA très portable d'une implantation CORBA à une autre. Les applications cibles sont des applications de couplage de code en simulation numérique à hautes performances.

Mots-clés : Couplage de codes, CORBA, Parallélisme d'invocations de méthode, Ordonnancement On-line Communication parallèle, Cache distribué, ATHAPASCAN

1 Introduction

Multi-scale and multi-physics simulations are emerging as solutions to achieve high fidelity in complex physical system simulations. The knowledge of physics for each piece of the simulation, computational algorithms and programming models impose that modern simulation code be developed as largely independent collection of software components. For instance, [24] reports simulation of the next generation of aircraft will require several hundreds of software components. Such simulation requires lot of CPU time and, therefore, has to exploit the aggregate power of resources scattered from available clusters of computational grid [13].

This report focus on applications based on the standard CORBA from OMG, which have been used with success in several projects for numerical simulation [3, 24]. CORBA specifies an open, vendor independent and language independent architecture for distributed applications. A CORBA component has an interface described in the Interface Description Language (IDL) which allows high level and modular programming. Moreover some high performance implementations of CORBA are available [33]. Thus, CORBA is well suited for high performance numerical simulation on a computational grid. Some recent projects have proposed to give access to core grid services [29] to CORBA based application, to rely on CORBA to develop Application Service Provider (ASP) [7] or to implant specification of interface for computer aided process engineering (European project CAPEOPEN) [3].

These projects propose quite complex interfaces to access the data of each component of the simulation and to exploit parallel computation. This complexity comes from the fact that CORBA does not address the problem of efficient composition of several invocations between severs in distributed environment.

- The first limitation is due the semantics of CORBA on the communication of effective parameters during the invocation of a method: all effective parameters of an *input* formal parameter of a method are communicated from the client to the server; all *output* parameters are communicated back to the client. The *direction* of a parameter defined in the IDL allows to optimize communication for one invocation between one client and one server. This local optimization does not implies a global optimization when a client invokes several methods on a set of servers to run a complex simulation: all parameters are moved between servers by passing through the client, which becomes the bottleneck for the scalability of using many components. The problem is exacerbated in case of iterative simulation. Thus, the communication should be optimized by considering the whole invocations. In this report, we presents our idea to keep standard CORBA semantics of invocation while generating only communications on the demand, *i.e.* if and only if a data is required on a component for an operation.

- The second difficulty comes from the semantic of the invocation of method in CORBA [26] between a client and a server. An invocation is mostly a blocking instruction: the caller waits until the server returns values. CORBA limits non-blocking invocation to method which do not have output values. Thus, to generate parallel flows of control, a programmer should mix blocking invocations with multithread computation, or he should mix non-blocking invocation with an other way to handle reply from server (event driven model of execution, concept of future [22], ...). Whatever is a concrete choice, the natural way a client should invokes methods has to be forbidden. In this report, we promote to keep the natural semantics of CORBA invocation while using a specific compiler and runtime, called HOMA, which allows to efficiently exploit a parallel computer at runtime.

Moreover, our propositions do not imply any user' source code transformation into the server or the client but only a recompilation, at least of the client, with a new generated client stub and server skeleton. This work is part of the HOMA project which aims at developing efficient solutions for code coupling of distributed CORBA based applications. Nevertheless, our results could easily be applied to other general distributed environment which required to compose functions.

In the next section we review the above discution on a motivating exemple. Section 3 presents the HOMA project. Then, we introduce the abstract interpretation of CORBA client invocations which allows to build at runtime the data flow graph between all invocations. This graph represents the future of the execution: the execution of task is associated to the execution of method on server object. T hen, we present how to execute such tasks to avoid an unbound use of resource by splitting blocking invocation into two non blocking invocations. Section 3.4 presents theoretical results about expected execution time of any CORBA client using HOMA versus standard implementation: the gain could be linear with respect of the number of processors if the application exhibits enough parallelism. Section 5 reports some experiments which fit our theoretical analysis. Then we conclude this paper.

2 A motivating example

This example comes from the project SIMBIO [5] in computational chemistry[1]. The goal of SIMBIO is to build a distributed application to compute complex simulation of molecular structure (a protein) surrounded by solvent.

2.1 SIMBIO

The application couples several parallel codes specialized in their domain. Considering two different scales: at the atomic scale the simulation is governed by quantum mechanics, while at macroscopic scale, the Newton's motion equations are solved to simulate the behavior of the molecular structure using a parallel molecular dynamic application [4]. The solvent is modeled as a charged continuum material. The electrostatic field is governed by Poisson's equations, with charged particles. The algorithm is based on integral formulation with a parallel implementation. Both models interact through a surface that separates the domain of Newton's equations (MD) and the domain of Poisson's equations (CM). The complete description of the numerical algorithm for the coupling is outside the scope of this report. To complete the description of SIMBIO, let us note that other applications are included in the simulation process: one of them needs to save all the positions and velocities of atoms in order make analysis of the trajectories of atoms; others permit to visualize the molecular structure and the surface of separation. Nevertheless, the interaction between the both models is through this surface.

2.2 Data flow of the whole SIMBIO application

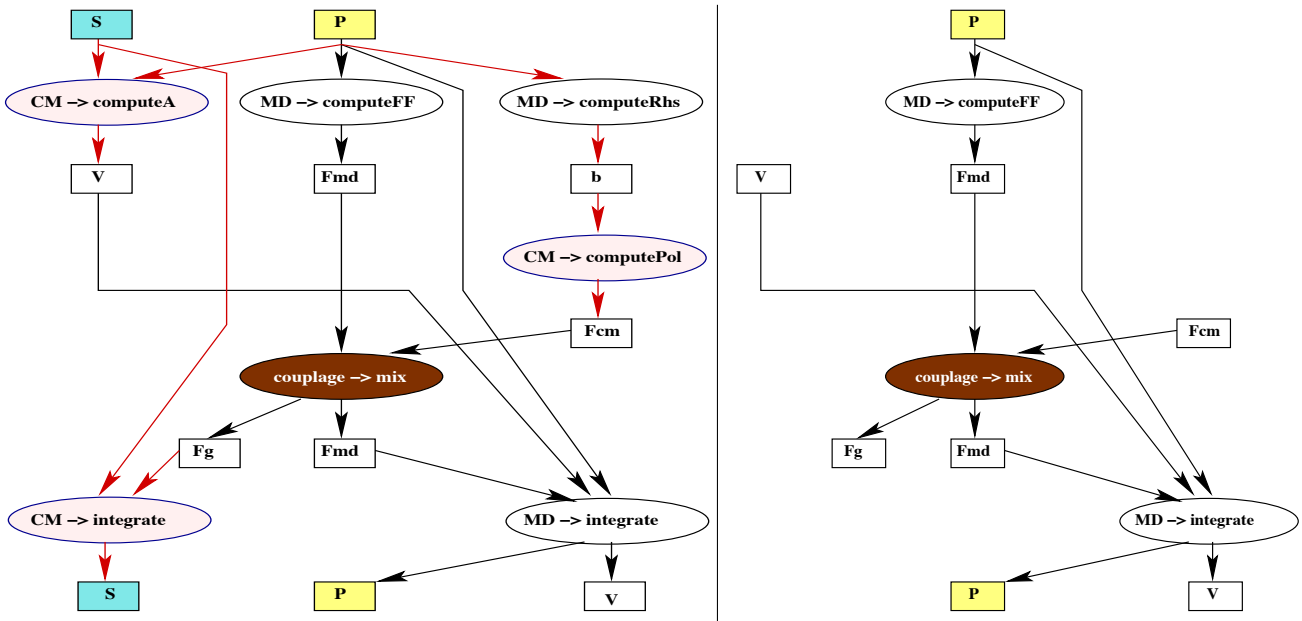
Figure 1 shows the coupling algorithm of SIMBIO simulation application as a multi step integration scheme. The molecular dynamics (*md* object) and the continuum method (*cm* object) are connected by a special task (*couplage* object) that implements the computation of the coupling terms of the physical models. Both objects are parallel: the 'atoms' P and 'surface' S data are in fact distributed objects (vector) among the processors. These objects are aggregate CORBA objects distributed among the processors, such as data parallel objets extension [22, 31, 28].

```
1.  // - main loop
2.  for(int i=0; i <MaxTimeStep; i++) {
3.     md->computeFF( P, F_MD );      // in P, out F_MD
4.     if (cmstep( i )) { //true iff CM step
5.       md->computeRhs( P, b );      // in P, out b
6.       cm->computeA( S, P );        // in S, in P
7.       cm->computePolarization( b, F_CM );   // in b, out F_CM
8.     }
9.       /* Coupling of terms */
10.     couplage->mix( F_CM, F_G, F_MD );   // in F_CM, out F_G, inout F_MD
11.     md->integrate( V, F_MD, P );        // inout V, in F_MD, inout P
12.     if (cmstep( i )) {
13.       cm->integrate( S, F_G );          // inout S, in F_G
14.     }
15.  } // - end main loop
```

Figure 1: Code of the simulation pilot that generates the data flow graph of SIMBIO application. *md*, *cm* and *couplage* are aggregate CORBA objects. Arguments of methods are CORBA sequence data type, eventually distributed.

The description of the data flow for one iteration depends on the current step and was given by a CORBA based client program sketched in figure 1. Figure 2 represents the data flow graph between the invocations of the methods of the figure 1. A box node represents a data, and an ellipse node represents a task. An edge from a task to a data means that the data is written by the task. An edge from a data to a task means that the data is read by the task. Each task in the data flow graph corresponds to the invocation of method on CORBA object (at lines 3, 5, 6, 7, 10, 11, 13 in figure 1).

[1]SIMBIO was an project funding by INRIA.



(a)Data flow graph if *cmstep* condition is true.

(b)Data flow graph if *cmstep* condition is false.

Figure 2: The data flow graph of SIMBIO of one iteration of the program sketched in figure 1. Depending on the value of *cmstep* either the graph a) or b) is build at runtime, then it is scheduled and executed on a distributed architecture.

2.3 Communication complexity

Section 3.4 presents a complete complexity analysis of the target applications but this paragraph focuses on the bottleneck in the program of figure 1 using a basic cost model with strong but realistic assumptions about the communications.

Let us assume that each communication between the MD and CM applications has about the same size L bytes. Typically, L ranges from $KBytes$ to $MBytes$. Thus one iteration of the program of figure 1, with the full evaluation of CM tasks, needs to communicate $5L$ bytes. Using P processes for both the MD and CM applications, each process except the coupling process, send or receives about $5L/P$ bytes of data.

On the top of standard CORBA implementation, the pilot that describes the data flow of iteration, is involved into each communications: the associated process receives and forwards about $5L$ bytes of data. Thus from the point of view of the communication, the pilot is the bottleneck of the code coupling application. This is inherent from the semantic of method invocations in CORBA which requires explicit communication of effective parameters.

As shown in next sections, using HOMA the pilot does not participate at all into the data communication[2] between applications: each process of the *md* and *cm* applications sends or receives in parallel about $5L$ bytes of data. Nevertheless, the bottleneck is due to the implementation of the coupling task (distributed or not), which is in the charge of the programmer and not due to the usage of the underlaying middleware to interconnect applications.

2.4 Execution on distributed architecture

Assuming the data flow graph known, the mapping of tasks onto the processors of distributed architecture allows to exploit parallelism between invocations. HOMA relies on ATHAPASCAN to manage the parallel execution of tasks with respect to data flow constraints. By using abstract interpretation and compilation technics, each invocation to method is captured to generate an ATHAPASCAN task: section 4 deals with the implantation of the compilation process to automatically build at runtime the data flow graph, the section 4.2 gives a detailed presentation of the generation of the client stub and server skeleton.

To generate a efficient ATHAPASCAN program, HOMA compiler uses two technics:

invocation by continuation to replace blocking invocation by two non-blocking invocations,

[2]Except for doing remote invocations, which is not considered in this basic analysis.

communication by necessity to avoid no required communication of data. transforms

The section 3.4 analyses the impact of this two technics on the complexity to execute any HOMA program on distributed architecture. Especially, we show that if the HOMA program (for instance the program in the figure 1) is a coarse grain highly parallel program, then the speed up of using HOMA versus standard CORBA is nearly linear with respect to the number of processor.

The SIMBIO application is used in section 5 to measure the pratical gain of using HOMA versus standard CORBA implantation: we exhibit good speed up as predicted by our analysis.

3 The Homa project

HOMA is a research action to provide an environment which allows to reuse the components and assemble them in order to build high performance and scalable distributed application for numerical simulation using CORBA component technology [27]. The assumption implicitly made in HOMA is that many components are available and the target application is built or extend by assembling these components.

3.1 Abstract interpretation

In order to achieve theoretical and practical efficient execution on a parallel architecture, the knowledge of the data dependencies related to the application appears as the key point for computing a good schedule. HOMA handles at runtime the abstract interpretation to gain information about the parameters required for each invocation. From an IDL definition, HOMA generates at compile time a new client stub which allows to capture the direction mode for each parameters involved in a sequence of invocations. At runtime, the control of the program is interpreted in the standard way while all invocations are intercepted to build the data flow graph of the execution. This graph is bipartite: the nodes form two sets, the "tasks" and the "data". A node of type "task" in this graph represent the invocation to a method on a server. Input arcs are input parameters (`in` in IDL), output arcs are output (`out`) parameters. A node of type "data" represents the value version of a parameter.

The figure 2 represents the data flow graph of our motivating exemple. Each nodes (data and task) have the attributes: the execution site of the task (the site where is instantiated the server object) and the size of the data. Moreover each nodes have a state (ready, not ready). A data is ready if it is written by a task. A task is ready if all its input data are ready.

3.2 Efficient non-preemptive execution

After scheduling the data flow graph (section 3.4), the runtime evaluates each tasks by respecting the order induced by the data flow constraints. The blocking invocation to a server is split in two non blocking invocations. This transformation is done into client stub and server skeleton generated from the IDL by our compiler. We call this transformation the "*invocation by continuation*" [17] as the adapted version of the "*wait by necessity*" [6] principle. In *invocation by continuation*, such as figure 3, a task invoking a blocking method is transformed in two tasks which invoke non-blocking methods. The execution of the former task invokes the first non-

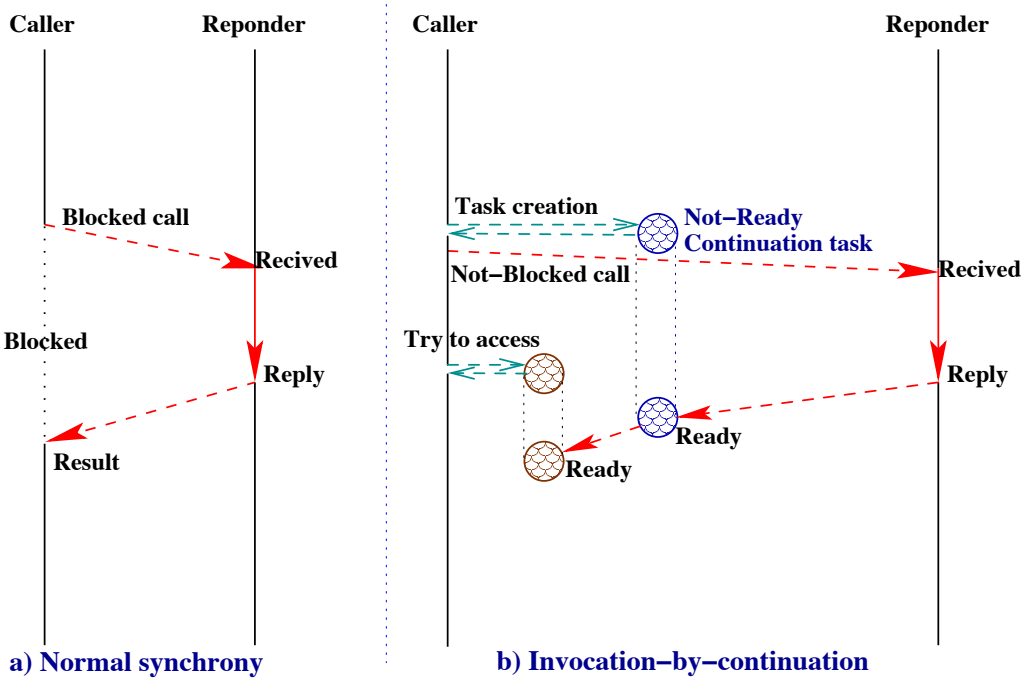


Figure 3: *Invocation-by-continuation* in comparison to normal synchrony.

blocking method, initiates the invocation on the server and addes a task into the macro data flow graph for the *continuation*. The continuation task will became automatically ready by the second non-blocking method invoked by the server skeleton.

The most import fact is that this transformation allows a non preemptive execution [16] of the tasks with efficient execution: because all tasks are never waiting for the results, all processors remains active while there exist ready tasks in the graph. For instance, it is possible to schedule the data flow graph using one and only one thread of control to invoke in parallel methods on several servers.

3.3 Automatic data movement

Due to the data flow representation of the execution of a sequence of invocations, it is possible to generate only the communication required to realize the invocation into a server. The *communication by continuation* strategy makes the need transformation automatically into our HOMA IDL compiler: it translates all parameters involved in invocations into global references. A reference is composed by a name and a version number. If a server which produces a parameter (*out* and *inout*) knows in advance all the servers that will consume it, the skeleton send the data to the consumers just after its production. A receiver will store it into a cache until its consumption. This mode of communication is called the *put method*. If a skeleton requires a data which was not yet received, it makes a request to the process that have produced it: this is the *get method* of communication, such as figure 4. In both methods, the client that generates the sequence of invocations is never involved into the communications, with the exception if it requires to read a data. The put method should be used if the data is small and could be broadcasted to several other servers for future invocation. But, because it requires the storage of data copies on some servers, such method should be carefully used with big data to avoid memory consumption. Moreover, the *invocation by continuation* technic allows parallel communication between servers.

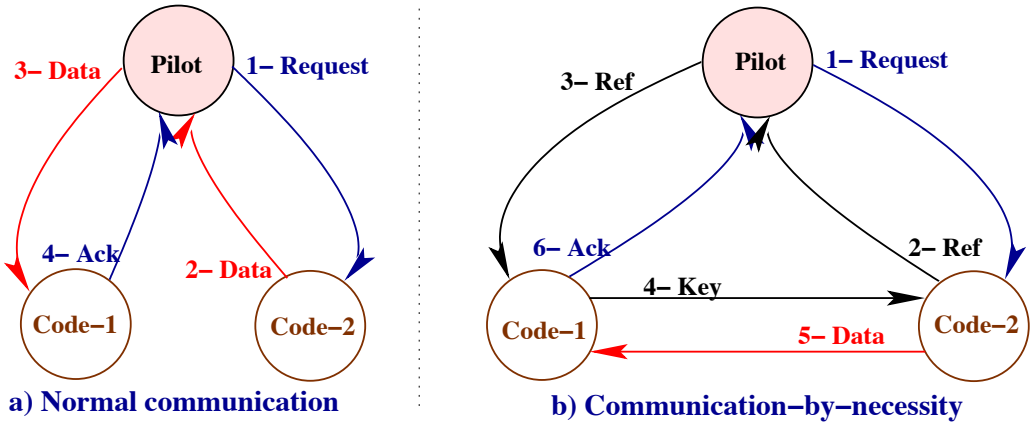


Figure 4: Communication-by-necessity (*get method*) versus the normal communication.

3.4 Theoritical scheduling results

HOMA relies on the macro data-flow execution kernel of ATHAPASCAN [16, 18], a parallel runtime environment to manage parallel execution. Let us recall the notations used in [16] defined on data flow graph: **Sequential time** T_1 denotes the sequential time that is defined from a serial execution of the program. **Parallel time** T_∞ is the arithmetic depth of graph taking into account the weights (computation costs) of task nodes. T_∞ is then a lower bound of the minimal time required by any non-preemptive schedule on an unbounded number of processors ignoring communications times (PRAM model [12]). **Communication volume** C_1 and **delay** C_∞ are evaluated from graph similarly to T_1 and T_∞ but taking into account only the weights (sizes) of data version nodes. C_1 is the sum of the weights over all data nodes; Assuming that the shared memory is emulated in an auxiliary file, C_1 is then an upper bound on the total number of accesses performed in this file during a serial execution. C_∞ is the length of the critical communication path.

Let us assume that the cost to communicate a message of size L is hL. Thanks to the work stealing algorithm detailed in [16], any data flow graph could be executed on a p processors machine in time $T_p = O(T1/p + T_\infty + h(C_1/p + C_\infty)) + O(\sigma)$. Where σ is the size of the data flow graph. This time includes the

overhead to compute the schedule of the program. The result is valid on a homogeneous parallel machine (a cluster), if and only if the tasks are non blocking, and if the communications are direct between processors.

Proposition 1 *The time to execute a CORBA client which makes invocations on a p processors machine using* HOMA *is:* $T_p^{homa} = O(T1/p + T_\infty + h(C_1/p + C_\infty)) + O(\sigma)$.

The cost model of execution of HOMA is inherited from ATHAPASCAN [16], because HOMA translates remote blocking invocation into non-blocking invocations. Section 3.2 presents how a blocking invocation is converted into two non blocking invocations, and section 3.3 shows our technique to make possible direct communication between servers. Such techniques are at the expense of the creation of an extra task for the continuation and an extra communication in case of the *get mode*. But it does not increases asymptotically neither the work of the program nor the volume of communication. Thus, the previous scheduling bound on $T_p = T_p^{homa}$ remains valid for any distributed application in CORBA where the client is compiled and executed with HOMA environment.

Note that if a standard CORBA environment is used and because of limitations explained in the introduction, the execution is sequential and data transit through the client. Therefore, the complexity of any execution is: $T_p^{corba} = O(T_1 + T_\infty + h(C_1 + C_\infty)) + O(\sigma)$.

The following proposition is a direct consequence of the previous results.

Proposition 2 . *If the program is highly parallel* ($T_\infty << T_1$) *and involves large data* ($C_\infty << C_1$), *then the gain of using* HOMA *versus CORBA is nearly linear with respect to the number of processors*($T_p^{corba}/T_p^{homa} = p$).

4 Implantation

4.1 Athapascan

ATHAPASCAN is a macro data-flow language [15, 19, 18] for asynchronous parallel programming. The C++ API permits to define the concurrency between computational tasks that synchronize on the access through a global distributed memory. Parallelism is explicit and functional but detection of synchronizations is implicit.

Tasks are created by the `Fork` instruction and shared objects in the global memory by the declaration of `Shared` variables. The execution relies on an interpretation algorithm that computes a macro data-flow graph [15]. The graph is direct and acyclic (DAG) and it encodes the computation and the data dependencies (read and write). It is used by the runtime support to schedule the tasks and map the data onto the target architecture. Implementation is based on the use of lightweight process (threads) and one-sided communications (actives messages).

```
1.  struct Fibonacci {
2.    void operator()(int n, Shared_w<int> r)
3.      { if (n <2) r.write(n)
4.        else {
5.          Shared<int> r1; Shared<int> r2;
6.          Fork<Fibonacci>()(n-1, r1);
7.          Fork<Fibonacci>()(n-2, r2);
8.          Fork<Sum>()(r, r1, r2);
9.        }
10.  } };
```

Figure 5: Fibonacci example. The task 'Sum' is not presented, it requires three parameters: the first declares to be write and the others to be read.

Figure 5 illustrates ATHAPASCAN on the classical Fibonacci program using a naive algorithm. A task is a function class as in the STL and it declares the mode of accesses on object in the global memory. The interpretation algorithm computes the data flow dependencies on `r1`, `r2` produced by tasks created at line 6, 7 and the task 'Sum' at line 8 that requires to read the value.

ATHAPASCAN defines the following mode of access: a read access (`Shared_r`), a write access (`Shared_w`), an exclusive access (`Shared_r_w`) and a cumulative access [15, 18]. A shared variable could be of any user's defined type if it satisfy some requirements to be "communicable" [15, 18] (definition of operators for the serialization). The runtime support garbages automatically the unused shared data. A multi-threaded runtime support executes ready tasks without blocking [15], because the declarations of shared variables and tasks are non blocking instructions.

4.2 Compilation process

To achieve parallel evaluation of the method invocations and lazy communication of parameters, we propose a chain of compilation (figure 6) from an IDL definition to client stub and server skeleton. Using it, only the code of pilot (client) and the coupled applications (severs) must be recompiled.

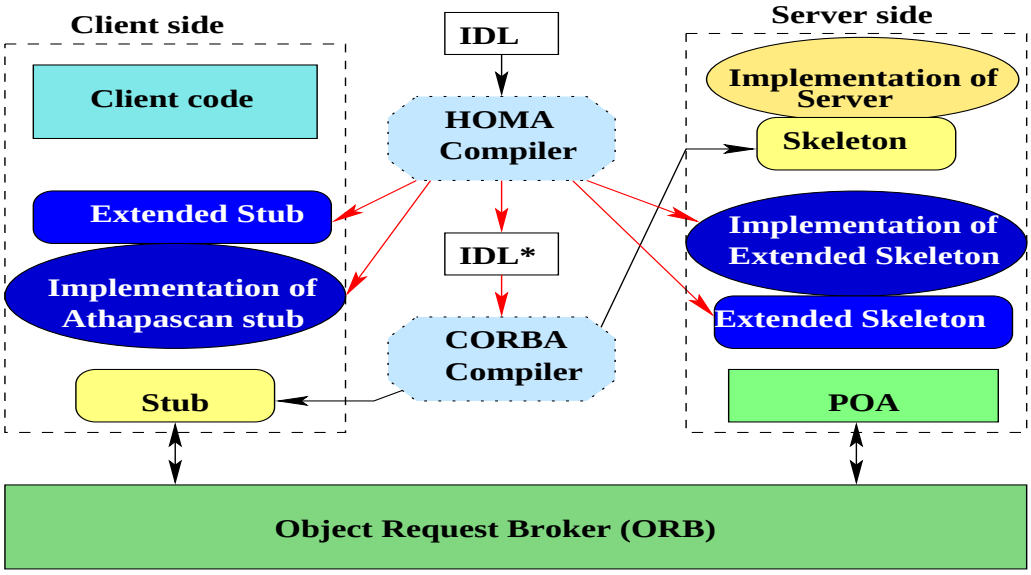


Figure 6: Whole compilation from IDL to extended stub and skeleton.

In the first step, the HOMA compiler produces the extended version of interface (IDL*) with an asynchronous version (*oneway*) of each synchronous method by converting returned value, *out* and *inout* parameters to a single *in* parameter which represents the continuation of the invocation, as presented in figure 8.

```
1.  struct type {                    5.  interface I {
2.    long a;                        6.    long foo(in long a,
3.    sequence<double> b;            7.         out long b, inout type c);
4.  };                               8.  }
```

Figure 7: Example of an IDL definition.

```
1.  #include <homa_rt.idl> // HOMA definition of Key and Continuation
2.  interface I {
3.    long foo(in long a, out long b, inout type c);
4.    oneway void a_foo(in Key a, in Key c, in Continuation cc);
5.  }
```

Figure 8: IDL* definition generated from figure 7.

Moreover, the type of formal parameters are translated to anonymous references (`Key` in figure 8) to allow lazy communication. Then IDL* is compiled using the standard IDL-to-C++ compiler provided by most CORBA implementation. It produces client stub and server skeleton of the extended interface. The HOMA compiler also generates server implementation of the added asynchronous methods in IDL* (`a_foo` in figure 8). Moreover, the HOMA compiler generates a ATHAPASCAN *parallel client stub* that relies on asynchronous methods of the client stub of the extended IDL*. The implementation of the server remains unchanged: the original method `foo` in the IDL file should always be implemented; and the client has the illusion of making invocations to standard CORBA method.

4.2.1 Parallel client stub generation

The purpose of the client stub is to build the request corresponding to a method invocation via the ORB. In order to use ATHAPASCAN, the client that invokes a method should creates a new task. The execution of task corresponds to the execution of remote method on the CORBA object. Because CORBA invocation is by default a blocking call, HOMA compiler generates for each blocking method a non-blocking method defined in IDL* with *oneway* attribute. Moreover, all *inout* and *out* parameters are converted to *in* parameters with value coding the information where to store the output value. From IDL in figure 7, the HOMA compiler generates the parallel client stub of figure 9. We assume that all standard IDL-to-C++ definitions of data structures and interfaces are in the namespace `Orig::` (in practice, some of the definitions are in the CORBA namespace [26] or in the namespaces due to modules in the IDL files). A client that invokes the method `foo` through a variable of type `homa::I_var` implicitly creates an ATHAPASCAN's task. Conversion operators exist between the ATHAPASCAN's `Shared<Key>` anonymous definitions (such as `homa::Long`) and the standard C++ typed definitions (`Orig::Long`). The required operators to serialize data type of shared variable are omitted. Line 10 declares a continuation variable which is an ATHAPASCAN shared variable that envelops the `homa::Continuation` used in method `a_foo`. This shared variable is declared as non ready. By this mechanism, the ATHAPASCAN's task `CONT_foo` becomes ready when the signalization arrived (see next paragraph).

Using this stub, a client executes an ATHAPASCAN program. Next section presents how to generate the tasks from the IDL definition of the method.

4.2.2 Athapascan's tasks generation

In order to use ATHAPASCAN as runtime support to automatically detect such dependencies, IDL definitions are translated to ATHAPASCAN definitions. Each invocation to method on a client program creates a task. Each argument of method and each interface object are translated to a shared object. The formal parameter in the IDL definition is translated to an ATHAPASCAN's shared in the task definition: `in` to a `Shared_r`, `inout` to a `Shared_r` and output parameters (*out*, *inout* and the return value) are passed indirectly through the continuation object declared at line 10. The reference to the CORBA object (`_self`) is explicitly passed to the function (line 10 in figure 9, line 4 in figure 10). In ATHAPASCAN, a task is a function, thus method

```
1.  namespace homa{
2.    typedef Shared<Key> Long;
3.    typedef Shared<Key> type;
4.    ...// same typedef's for the data types
5.    class I { // Definition interface I
6.      Shared<Orig::I_ptr> _self; //Athapascan's reference to CORBA object
7.    public:
8.      //Definition of foo using typedef's in namespace homa::
9.      Long foo(Long& a, Long& b, type& c){
10.        Long retval;
11.        Continuation cc( b, c, retval );
12.        Fork<INVOKE_foo>()(_self, a, c, cc );
13.        Fork<CONT_foo>()( cc, b, c, retval );
14.        return retval;
15.    } /*foo*/ }; /*class I*/ } /*namespace homa*/
```

Figure 9: Client stub for the IDL of figure 7. `I_ptr` and `I_var` definitions are omitted, but are like a pointer to an object of class `homa::I`. `INVOKE_foo` is defined in figure 10.

invocation is translated to function call with an added parameter that represents the object. Two methods cannot be executed concurrently on the same object without any assumption on the method definition (such as leaving invariant the state of the object). Thus, their executions are serialized and the `INVOKE_foo` task requires the reference to the CORBA object using the exclusive mode of access (`Shared_r_w`): two tasks on the same CORBA object are executed in the same order of their creations, *i.e.* in the same order of the invocations if the client uses the standard CORBA stub.

```
1.  namespace homa{
2.  class I {
3.  struct INVOKE_foo {
4.    void operator()(Shared_r_w<Orig::I_ptr> obj, // CORBA object
5.                    Shared_r<Orig::Long> a, Shared_r<Orig::type> c, Continuation cc )
6.    {
7.       obj.access()->a_foo( a.read(), c.read(), cc.reference() );
8.    }/*operator()*/   };/*struct*/   };/*class*/   }/*end of namespace homa::*/
```

Figure 10: Basic implementation of task `INVOKE_foo` associated to `foo`.

Line 7 in figure 10, the method `a_foo` is the non blocking *oneway* method is invoked with the continuation object reference. On the server side, the method `a_foo` executes the method `foo`. Then, it invokes on the client side a method to return the values (anonymous `Key`) which signals the continuation task that `a_foo` has completed its execution.

The `externalize` method is an ATHAPASCAN's method that permits to communicate shared variable that should be produced in an other context. On the server side, the received references to the two externalized Shared variables are converted back to `Shared_w` variables. After the execution of the method, the server writes the results on each shared using the ATHAPASCAN's write method. This is the principle of converting blocking method to non blocking (*oneway*) method: output parameters are passed as extra parameters indicating where to store output value.

Moreover, using such implementation, the thread of control that executes the invocation to the method `foo` is blocked until the completion of the call. Thus the computing resource is not available to execute other useful work. In order to avoid this problem, we generate a quite longer code that split the invocation in to phase using asynchronous call to method. On the client side, the task does invocation to a oneway `foo_async` method that takes has extra parameter a reference to data need for the task to continue. Then the task forks a new task to execute the continuation. The server replies with an invocation to an other oneway/asynchronous method in order to activate the task for the continuation.

4.2.3 Reference generation

A reference into the cache consists of a reference to the considered cache (IOR of the object) and an access key into cache. All the data smaller than a certain threshhold (set up by implementation) are communicated directly and are not stocked in the caches. The stocked data in cache or immediate data are through the type `Any` of CORBA. The union of a reference or of an immediate data is defined in the IDL file of HOMA.

The references are generated by the servers in every method invocation and for every argument in writing (of direction *out*, *inout*, or return value). The client (pilot) also could generate the references and of this manner not to await as the server return them. The advantage is on the future method invocations without awaiting the return of a reference generated by a preceding invocation. But in this case, the synchronisation writer-reader on an argument between the server that produces a value and the one that will consume it will have to be managed besides.

4.2.4 Cache implementation and architecture

Data into cache is stored using the CORBA `Any` type. A cache is a CORBA object. HOMA compiler generates the necessary initialization code to create one cache object per (Unix) process. A reference to a data is the reference to the cache CORBA object plus the reference to its entry into the cache. Moreover, a threshold permits to communicate small data into a reference, we have called `Key` the union of a reference and a value (see figure 8.

Each cache keeps a list of copies to other caches located on different (Unix) processes. A LRU style policy manages the cache for the input data. The output values are stored until they will be deleted. Update of data into the cache is managed by the client ATHAPASCAN program: an access to shared variable in ATHAPASCAN represents a *version* to the associated shared data. Any version destroyed by ATHAPASCAN [15, 18] is reported to operation on the type of shared data, which is implemented to report the destruction on the cache.

5 Experimental results

All of the experiments were made by using the OmniORB3 [14] implementation of CORBA. The programs were carried out on the iCluster of INRIA at Grenoble (PC 733Mhz, 256Mo, network 100Mbit/s).

5.1 Elementary costs

For measuring the different overhead costs of HOMA, the first experience was done. Figure 11 reports the decomposition of the costs of one method invocation using HOMA according to the size of the arguments (a sequence of double). The extended method invocation is composed of two parts. First, the Pilot does one non-blocked invocation on consumer code using HOMA's reference type which is independant from message size then this consumer searches the corresponded data by one blocked invocation on producter code.

In addition to the cache management (remote get, blue zone) and the cost of method invocation (white zone), the costs of conversion of one sequence of double towards the type `Any` (which includes a memory copy) is not negligible (brown zone). The other costs: conversion `Any` towards the sequence of double, and the access of insertion/ extraction functions are negligible.

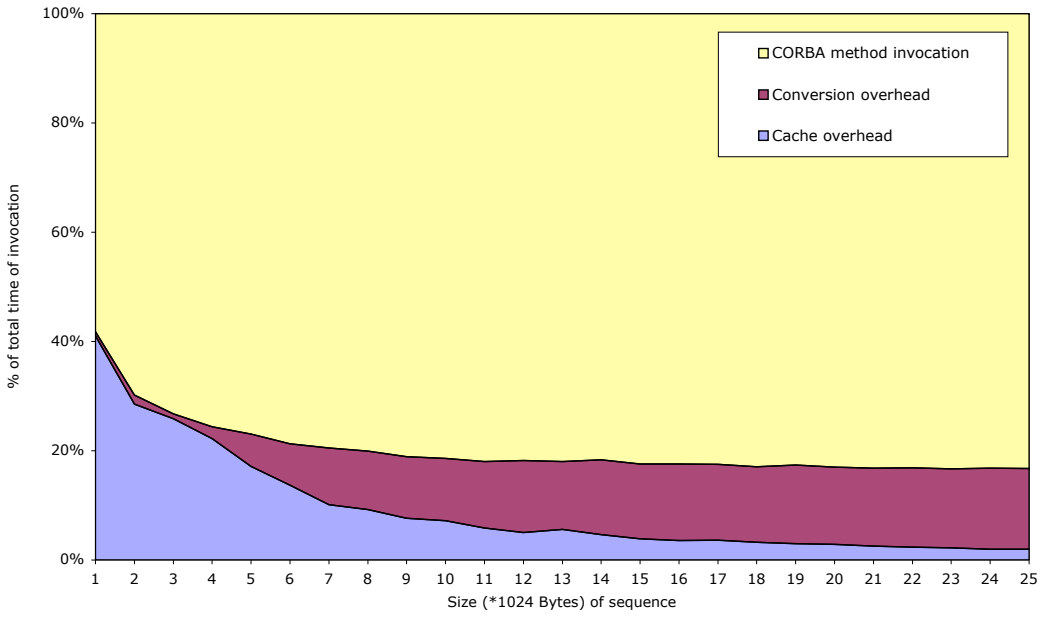


Figure 11: The cost decomposition of one method invocation using HOMA.

5.2 Aggregate bandwidth

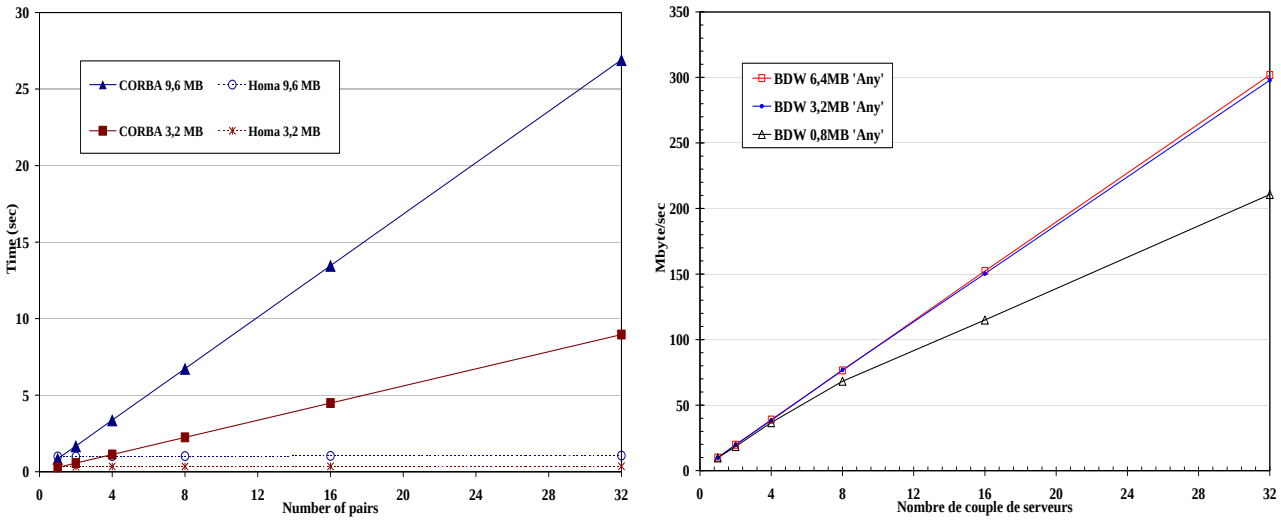
In the second experience, figure 12, the scalability of HOMA versus the standard CORBA is compared. The measured time is the consumation time (line 6,7). The lack of a defered approach for argument passing in CORBA causes the linear dependancy between the invocation time and the number of couples but using HOMA inter-couple data communication is possible.

Figure 13.a shows the results, the measured times of HOMA (horizontal curves) are nearly constant whatever is the number of servers. It shows that communication occurs in parallel, and our parallel client stub generation is effective. Also it shows that the communications are serialisees on the pilot in the case of standard CORBA.

```
1.  // Call the producters                          5.  // Call the consumers
2.  for(int i=0; i < N; i++)                       6.  for(int i=0; i < N; i++)
3.     SS[i] -> produce ( P[i] ); // P[i] is output 7.     CC[i] -> consume ( P[i] ); // P[i] is input
```

Figure 12: The pilot code for measuring the aggregate bandwidth.

All the aims of HOMA is to automate the exploitation of these communications paralleles has to leave same description of the program. The brough gain is more than 26 for 32 couples.



(a) Times HOMA versus CORBA.

(b)Aggregate bandwidth with HOMA.

Figure 13: Performances with different numbers of servers.

Figure 13.b reports the aggregated bandwidth of a client program that iterates through N couples of servers. Each couple is composed by a server that produces a value and a server that consumes it. Each server is mapped on a different machine. Using HOMA the aggregate bandwidth linearly growths as the number of couples, the maximum aggregate bandwidth is about $304 MBytes/s$ for 32 couples. The average point-to-point bandwidth between two servers is about $9.6 MBytes/s$ [3].

The difference of the point-to-point bandwith with respect to MPI or TCP comes from: 1/ the generic cache architecture using the CORBA `Any` data type, involving some extra memory copy; 2/ the "*get*" strategy which involves one extra communication to get the data with respect to a "*put*" strategy. All these factors limit the maximal available bandwidth.

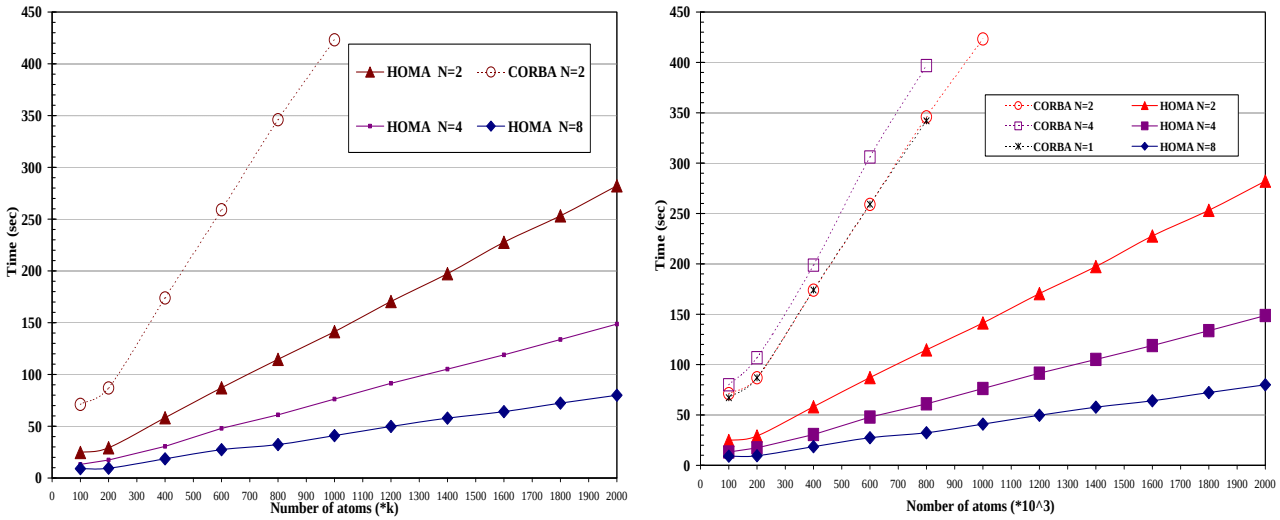
5.3 SIMBIO

This experiment measures the ability of HOMA to make parallel execution of method invocations. We simulate the iterative scheme of our motivating example described in section 2. With sequential execution as in standard CORBA implementation, all method invocations are executed sequentially. For the experiment, we parallelize each object of the figure 1 page 4.

In the experiments each objects are distributed to N sub-objects with the same interface: each invocation is dispatched to several invocations on each sub-objects. Note that the control flow depends on the evaluation of `cmstep` which is only known at runtime. The figure 14.a reports the time of 10 iterations, in case if `cmstep` returns false (less dependencies thus more parallelism) with respect of the size of the sequence and N sub-objects for MD and the half for CM (p=3/2N). Unlike standard CORBA, there is a linear relation between the execution time using HOMA and object parallelization: the time decreases linearly as the number of processors increases. In this case, the gain of HOMA for great number of atoms is very close to the theoretical expectation ($T_p = T_1$ / p).

When the condition ($cmstep(k)$) is true, there are more communication and less parallelism but also the gain of HOMA remains nearly linear versus granularity number, figure 14.b.

[3] Maximum point-to-point bandwith with respect to MPI or TCP is about $11 MBytes/s$ on iCluster.



(a) without the mixing step.

(b)with the mixing step.

Figure 14: Execution time for SIMBIO with several number of objects per application.

6 Related works

6.1 Parallel-codes coupling

Several projects attempt to achieve codes coupling through the use of specific code coupling tools (PAWS [2], MpCCI [20] , ...). They are originally targeted to parallel machines with some on-going works to target Grid infrastructure. Ad-hoc communication layers (MPI, sockets, shared memory segments, ...) are used based on static coupling (at compile time); not "plug and play". All of them lack of explicit coupling interfaces, standardization and interoperability. However they provide powerfull functions such as interpolation, time management and mapping.

6.2 Lazy approach and cache strategy

In order to achieve the necessary throughput and latency in CORBA, we observed two groups of research. First, the attempts to provide an object replication framework specially to used over Internet like CASCADE [10], ScaFDOCS system [23]. The object replication approach is not efficient for the methods with large size arguments which is our target applications, the numerical simulations. Second, the researches for caching results of method invocations so that the subsequent method invocations will return locally cached values without invoking the remote operation. A very limited solution is provided by the so called *smart stub* mechanism in some existing CORBA implementations (e.g. Orbix [21] and VisiBroker). But the burden of maintaining coherency lies entirely on the application programmer. MinORB [25] is a research ORB that allows caching partial results of read method invocations at the client side. However, to our knowledge, there is no any attempt to overcome the lack of supporting lazy approach for data transfering.

6.3 Aggregate object for parallel applications

The projects PARDIS [22] and PACO [31] have studied the integration of data parallel application in the CORBA object model. OMG has recently published a specification to handle data parallel application [28]. All these projects are based on the concept of multi-function [1]: n callers coordinate the execution of m invocations of method on part of distributed object. During this coordination step, the input effective parameters are scattered to the each part of the object and, at the end, output parameters are gathered. Currently Homa deals with efficient and scalable implementation of the composition of method invocations and does not manage any redistribution of parameters. Nevertheless, Homa makes few assumptions about the distributed nature of the objects or parameters and such parallel object extension should be easily reused.

Using PARDIS or PACO, the program in figure 1 involves useless communication of data between the client and all the servers. Thus without code modification, neither PARDIS nor PACO provides an efficient support

robust to the composition, especially for iterative scheme. Nevertheless, PARDIS provides the concept of *future* of a result. It should be used explicitly in the client program while our approach makes it use implicit.

6.4 Web services orchestration

The industry has used a number of terms to describe how components can be connected together to build complex business processes. Workflow and document management systems have existed as a means to handle the routing of work between various resources in an IT organization. With the introduction of web services, terms such as "web services composition" and "web services flow" were used to describe the composition of web services in a process flow. More recently, the terms *orchestration* and *choreography* have been used to describe this. Orchestration describes how web services can interact with each other at the message level, including the business logic and execution order of the interactions. These interactions may span applications and/or organizations, and result in a long-lived, transactional, multi-step process model. Choreography tracks the sequence of messages that may involve multiple parties and multiple sources, including customers, suppliers, and partners. Choreography is typically associated with the public message exchanges that occur between multiple web services, rather than a specific business process that is executed by a single party [30].

The ability to invoke services in an asynchronous manner is vital to achieving the reliability, scalability, and adaptability required by today's IT environments. With asynchronous support, a business process can invoke web services concurrently rather than sequentially in order to enhance performance [30].

To our knowledge of workflow technologies, although like our aims the main objective is the efficient and scalable composition of several services however data communication is not an important challenge such as numerical simulation code coupling(involving large data transfer in iterative scheme).

6.5 Service-based Metacomputing

NetSolve [9], Ninf [32] and DIET [8] are the RPC-based client/agent/server systems that enable users to solve complex scientific problems remotely. Netsolve and Ninf developed their own runtime support for communication but DIET is based on CORBA technology. These systems apply some modification for using existing-services. Single request scheduling was their starting objective. In Netsolve and Ninf, Request Sequencing [11] enables programmers to demand optimization of communication on a sequence of requests. But it is explicit and not transparent. There is nothing support to efficiently exploit data parallel objects.

7 Conclusions

This report presents our approach for an efficient and scalable composition of distributed applications viewed as composition of CORBA method invocations. Too fulfills this goal we presents how to combine two basic functionalities: parallel invocations of method based on ATHAPASCAN, and a lazy communication to deferred data transfer until the request of (server) object that requires it. It is based on a distributed cache. Together these features permit to several servers to communicate in parallel. Moreover, this functionalities could be automatically used in existing server and client programs thanks to an IDL compiler technics. As a result we propose how to extract from the standard CORBA IDL enough information that permits us to automatically write a program that coordinate execution of method invocations. Also, we propose a new cache strategy for code coupling to deferred the result communication as late as possible. Its efficiency is depended to the data size of the result parameter, so for the small size parameters, the data transfer is direct without caching.

Preliminary experiments show the effectiveness of our solution based on a standard CORBA ORB implementation. Some overhead comes from data conversion used by our cache implementation (mainly the use of `Any`) that limits the performance of our runtime. Nevertheless, an aggregate bandwidth of about $300MBytes/s$ was measured ($380MBytes/s$ with specialization) that enables to saturate any switch of a backbone connection between geographically distant clusters, that is the case of distributed coupling parallel applications.

Ongoing works concern the ability for a server to put produced data before a server remotely get it, thanks to the knowledge of the data flow graph of the execution; and the ability to reuse parallel CORBA objects [28, 22, 31]. Moreover, technical work currently deals with trying to reduce the overhead of our implementation by an efficient cache (without copy) and by testing other fastest CORBA implementations.

We also study the integration of scheduling heuristics to deploy or redeploy server objects taking into account the knowledge of the data flow graph of the execution. The purpose is to allow the integration of the execution of an application with a dynamic management of the resources in order to optimize a criteria (*e.g.* minimize the completion time). The target application is numerical simulation in computer aided process engineering [3].

To the knowledge of the authors, HOMA's work is the first attempt to consider the efficiency and the scalability of the composition of CORBA method invocations for high performance applications in numerical simulation.

References

[1] J-P. Banâtre, M. Banâtre, and F. Ployette. The concept of multi-fonction: a general structuring tool for distributed operating systems. In *6th Int. Conf. on Distributed Computing Systems*, Cambridge, May 1986.

[2] P.H. Beckman, P.K. Fasel, W.F. Humphrey, and S.M. Mniszewski. Efficient coupling of parallel applications using paws. Technical report, Los Alamos National Laboratory, USA, 1998.

[3] J.-P. Belaud, B. Braunschweig, and M. Pons. Open software architecture for process simulation : The current status of cape-open standard. In *, European Symposium on Computer Aided Process Engineering*, 2002.

[4] P.-E. Bernard, T. Gautier, and D. Trystram. Large scale simulation of parallel molecular dynamics. In *Proceedings of Second Merged Symposium IPPS/SPDP 13th International Parallel Processing Symposium and 10th Symposium on Parallel and Distributed Processing*, San Juan, Puerto Rico, April 1999.

[5] P.E. Bernard and O. Coulaud. Parallel constrained molecular dynamics. *INRIA, Research report RR-3868*, Janvier 2000.

[6] D. Caromel. Towards a method of object-oriented concurrent programming. *Communication of the ACM*, 36:90–102, 1993.

[7] E. Caron, F. Desprez, E. Fleury, F. Lombard, J.-M. Nicod, M. Quinson, and F. Suter. *Calcul réparti à grande échelle*, chapter Une approche hiérarchique des serveurs de calculs. Hermès Science Paris, 2002. ISBN 2-7462-0472-X.

[8] Eddy Caron, Frédéric Desprez, Frédéric Lombard, Jean-Marc Nicod, Martin Quinson, and Frédéric Suter. A Scalable Approach to Network Enabled Servers. In B. Monien and R. Feldmann, editors, *Proceedings of the 8th International EuroPar Conference*, volume 2400 of *Lecture Notes in Computer Science*, pages 907–910, Paderborn, Germany, August 2002. Springer-Verlag.

[9] H. Casanova and J. Dongarra. NetSolve: A network server for solving computational science problems. In *Workshop of Vector and Parallel computing*, Manno, Switzerland, March 1997. SPEEDUP Society.

[10] G. V. Chockler, D. Dolev, R. Friedman, and R. Vitenberg. Implementing a caching service for distributed corba objects. In *Proc. of IFIP/ACM International Conf. on Distributed Systems Platforms and Open Distributed Processing*, NY, USA, 2000.

[11] C. Arnold Dorian, Bachmann Dieter, and J. Dongarra. Request sequencing: Optimizing communication for the grid. In *EuroPar – Parallel Processing*, 2000.

[12] Steven Fortune and James Wyllie. Parallelism in random access machines. In *Proceedings of the tenth annual ACM symposium on Theory of computing*, pages 114–118. ACM Press, 1978.

[13] I. Foster, C. Kesselman, and S. Tuecke. The anatomy of the grid. *Intl J. Supercomputer Applications*, 2001.

[14] Previously from AT & T Laboratories Cambridge. *OmniORB3 Programming Guide*, June 2002.

[15] F. Galilée, J.-L. Roch, G. Cavalheiro, and M. Doreille. Athapascan-1: On-line building data flow graph in a parallel language. In IEEE, editor, *Pact'98*, pages 88–95, Paris, France, October 1998.

[16] F. Galilée, J.L. Roch, G.G.H Cavalheiro, and M. Doreille. Athapascan-1: On-line building data flow graph in a parallel language. In IEEE, editor, *Pact'98*, pages 88–95, Paris, France, October 1998.

[17] T. Gautier and H.R. Hamidi. Homa : un compilateur idl optimisant les communications des données pour la composition d'invocations de méthodes corba. In *RenPar'2003*, France, Octobre 2003.

[18] T. Gautier, R. Revire, and J.-L. Roch. Athapascan: Api for asynchronous parallel programming. Technical Report RR-0276, INRIA Rhône-Alpes, projet APACHE, February 2003.

[19] T. Gautier, R. Revire, and F Zara. Efficient and easy parallel implementation of large numerical simulation. In Springer, editor, *Proceedings of ParSim03 of EuroPVM/MPI03*, pages 663–666, Venice, Italy, 2003.

[20] M. G. Hackenberg, P. Post, R. Redler, and B. Steckel. Mpcci, multidisciplinary applications and multigrid. In *ECCOMAS 2000*, CIMNE Barcelona, 2000.

[21] IONA. *Orbix Programming Guide*. IONA Technology Ltd., 1995.

[22] K. Keahey and D Gannon. PARDIS: a parallel approach to CORBA. In *Proceedings of the ACM/IEE International Conference of Supercomputing*, 1997.

[23] R. Kordale, M. Ahamad, and M. Devarkonda. Object caching in a corba compliant system. *USENIX Computing Systems Journal*, 9(4), 1996.

[24] I. Lopez, G.J. Follen, R. Gutierrez, I. Foster, B. Ginsburg, O. Larsson, and S. Tuecke. Using corba and globus to coordinate multidisciplinary aeroscience applications. In *Proceedings of the NASA HPCC/CAS Workshop*, pages 15–17, Feb. 2000.

[25] P. Martin, V. Callaghan, and A. Clark. High performance distributed objects using caching proxies for large scale applications. In *Proceeding of the IEEE International Symposium on Distributed Objects and Applications (DOA'99)*, Edinburgh, Scotland, september 1999.

[26] OMG. Common object request broker architecture (corba/iiop). Technical report, OMG formal/2002-12-06, 2002.

[27] OMG. Corba component model. Technical report, OMG, formal/2002-06-65, 2002.

[28] OMG. Data parallel object. Technical report, OMG formal/2002-06-65, 2002.

[29] M. Parashar, G. von Laszewski, S. Verma, J. Gawor, K. Keahey, and N. Rehn. A corba commodity grid kit. In *Concurrency and Computation: Practice and Experience, John Wiley and Sons*, 2002.

[30] Chris Pelts. *Web services orchestration: a review of emerging technologies, tools, and standards*. Hewlett-Packard, Co., January 2003.

[31] C. René and T. Priol. MPI code encapsulating using parallel CORBA object. *Cluster Computing*, 3(4):255–263, 2000.

[32] Mitsuhisa Sato, Hidemoto Nakada, Satoshi Sekiguchi, Satoshi Matsuoka, Umpei Nagashima, and Hiromitsu Takagi. Ninf: A network based information library for a global world-wide computing infrastracture. In *HPCN'97 (LNCS-1225)*, pages 491–502, 1997.

[33] D. Schmidt, A. Gokhale, T. Harrison, D. Levine, and C. Cleeland. Tao: A high-performance endsystem architecture for real-time corba. In *IEEE Communications Magazine feature topic issue on Distributed Object Computing.*, February 1997.

Unité de recherche INRIA Rhône-Alpes
655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier (France)

Unité de recherche INRIA Futurs : Parc Club Orsay Université - ZAC des Vignes
4, rue Jacques Monod - 91893 ORSAY Cedex (France)
Unité de recherche INRIA Lorraine : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex (France)
Unité de recherche INRIA Rennes : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex (France)
Unité de recherche INRIA Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex (France)
Unité de recherche INRIA Sophia Antipolis : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex (France)

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399