Optimizing Metacomputing with Communication-Computation Overlap

Françoise Baude, Denis Caromel, Nathalie Furmento, and David Sagnol

OASIS – Joint Project CNRS / INRIA / University of Nice Sophia – Antipolis
INRIA - 2004 route des Lucioles - B.P. 93 - 06902 Valbonne Cedex, France
FirstName.LastName@sophia.inria.fr

Abstract. In the framework of distributed object systems, this paper presents the concepts and an implementation of an overlapping mechanism between communication and computation. This mechanism allows to decrease the execution time of a remote method invocation with parameters of large size. Its implementation and related experiments in the C++// language running on top of Globus and Nexus are described.

Keywords: Distributed Objects, C++, Metacomputing, Nexus/Globus, Lightweight Process, Remote Method Invocation, Pipelining, Future, Overlapping communication and computation

1 Introduction

1.1 General Objective

Distributed supercomputing applications require large amounts of computational resources that often only computational grids environments can provide. The price to pay when executing on such environments is the mandatory use of a high latency, low throughput network. As a consequence, any solution that could help to lower communication costs would be worth considering.

A basic idea 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 [11] or MPI [15]).

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. But as far as we know, this idea has never been investigated in the area of distributed object-oriented languages.

V. Malyshkin (Ed.): PaCT 2001, LNCS 2127, pp. 190–204, 2001.

1.2 Formulation of the Problem

The general idea featuring the concept of overlapping is that during a remote computation dealing with large data requiring transmission, communication and computation are automatically split in steps with a smaller data volume; then, it is only a question of pipelining these steps in order to achieve overlapping between the current step of the remote computation and the data transmission related to the next step of the remote computation. This requires executing a computation and a transmission step **at the same time**. One way to achieve this is to use non-blocking communications.

Schematically, in the SPMD or SIMD programming models, a similar computation has to be executed on each element of a large but fixed size data structure. So, the compiler or the run-time system is quite easily able to split it into small pieces, send each one in turn, apply the computation on each piece once it is received. If the compiler or the run-time system is not able to automatically decide how to split the data, the programmer can help. Thus, the implementation of this technique has generally been restricted to the field of data-parallel languages for parallel architectures with distributed memory: HPF [3], FortranD [17], but also in LOCCS [8], a library for communication routines and computation.

But, how should the same problem be tackled with, in the area of distributed object-oriented languages ? In this context, the whole computation taking place on the distributed entities can be expressed as remote service invocations through method calls as RMI [16] in Java or RPC in C/C++ [2], even if ultimately very low-level communications, e.g., network communications, are used. In order to exhibit parallelism between distributed computations, a solution is to use asynchronous – or non-blocking – service invocations instead of blocking ones as featured by classical RPCs. Many models and languages have exploited this idea [4]. In particular, we have designed and implemented distributed extensions to object-oriented languages such as Eiffel, C++ and Java, that enforce sequential code reuse in a parallel and distributed setting [6,7]. In such languages extensions, each service invocation can be executed in parallel with the on-going computation. Once the result of the service is required, a wait-by-necessity mechanism comes to help [5]. More information related to this model will be given in Sect. 3.

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.

Main idea. The essence of our proposition is thus to **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. Optimisation of the parameter copying process, as in [18] is a different but complementary approach.

As the rest of the paper will show, the way the technique is designed and implemented implies an easy and flexible usage for programmers, and, in some circumstances, remarkable performance gains on a LAN-based environment as well as on a WAN-based one (i.e. on a grid).

1.3 Design Guidelines

To implement this general idea, several problems have to be solved:

1. design and implement elementary mechanisms, such as: data splitting, computation steps that can deal with partial data, ...;
2. make it as much as possible a transparent mechanism for programmers, but give them the possibility to guide the data splitting;
3. try to determine the appropriate size for data packets (i.e. try to estimate the duration of the different steps).

Our contribution is to design, implement, and evaluate it within the context of an object-oriented language extended with mechanisms for parallelism and distribution, C++// [6]. Only points 1 and 2 are resolved in this paper. *Automatically* solving point 3 would require more precise information about the computation and the underlying communication performances (a strategy for data-parallelism languages running on dedicated parallel machines is developed for instance in [8]).

As communication performances in the context of grid computing are quite unpredictable and vary dynamically, solving point 3 would be essentially manual (even if the programmer could be helped by some performance measurement tool) knowing that the benefits of the overlapping would also vary dynamically. As slicing of data into smaller units and also the corresponding slicing of computations seem to have to be manually done by the programmer, our solution can help: it provides an easy way in terms of programming effort, and a cheap way in terms of running cost, to describe and try to take advantage of pipelining in distributed object-oriented applications.

Structure of the paper. In Sect. 2, requirements and steps for point 1 are discussed. Then, strategies for splitting requests (point 2) are presented. Section 3 introduces an implementation for this technique using the C++// language, whose runtime is based on both standard and lightweight processes (through Nexus and Globus). In Sect. 4, we present some benchmarks whose main purpose are to validate the technique and its implementation while exhibiting some cases where the gain is almost optimal. This work is an extension of [1] in the sense that, excepted the design, the implementation, experiments, analysis and learned lessons are new, as they arise in a broader context (multithreading plus metacomputing).

2 Communication/Computation Overlap

This section presents the overlapping technique and the requirements for its implementation.

2.1 Elementary Mechanisms

The following items are the building blocks of the technique:

- send a request in pieces (without taking into account the strategy used for splitting it);
- be able to rebuild a partial request in such a way that service execution can be started;
- be able to integrate missing data when it arrives even if service execution has started;
- be able to block the computation if it tries to use missing data.

Step for Request Creation. In every system that proposes an RPC mechanism, the remote service request has to contain the method ID and the different parameters of the call which are marshalled using a deep (i.e. recursive) copy of the objects graph[1]. After that, the request is sent.

Requirement 1. Have access to the runtime code that sends requests in order to be able to decide *when* to send a request piece.

Step for Request Rebuilding. Once arrived in the remote system, the request is rebuilt: each parameter is reconstructed with the corresponding data and then the service can start. For implementing the overlapping technique, we have to be able to put a mark for the missing data. This mark informs the service that data are, temporarily, unavailable.

Requirement 2. Have access to the runtime code that deals with the unmarshalling of the request in order to manage marks of missing pieces.

When the remote context receives a new part of a request that is already partially rebuilt, the context has to be able to deal with it in an *automatic* and *transparent* way regarding the service that is already executing.

Requirement 3. A mechanism that receives and manages messages transparently.

Step for Service Execution. The service can run without any problem as long as it does not attempt to access missing data. An automatic and transparent blocking mechanism is required when it tries to use a missing data. In the same way, resumption has to be transparent and automatic. This requires a wait-by-necessity mechanism [5]. Such a mechanism is provided by the classical *future* mechanism as originally designed in Multilisp [12].

Requirement 4. Future types available from the programming language.

Assuming the previous requirement is fulfilled, each missing data at the instantiation time of the request object is replaced with a data type presenting a *future* semantic.

[1] If a field of an object is a reference to a remote object, i.e. a proxy, we just flatten a copy of this proxy.

2.2 Strategies for Splitting a Request

This section deals with the point 2 mentioned in the introduction. The crucial idea is to break, in the most transparent way for the programmers, the request parameters. It requires a modification of the marshalling/unmarshalling routines of objects. Whether these routines are generic or not, we have to be able to overload them.

Requirement 5. Be able to change the default marshalling/unmarshalling routines.

Strategies can be split in two groups whether they modify or not the class of the objects involved in a request.

With Class Modification A **new** class called *later* is introduced, from which all objects that require to be sent latter have to inherit from (see Code 1 for an example). Objects from these classes 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 $dom \rightarrow rang(m1, m2)$ in Code 2 for example). According to the previous requirements, *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.

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 Code 2), 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.

Code 1 (Definition of a later class).

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

Without Class Modification. Two kinds of strategies come to mind:

1. either a new routine could replace the one used by default by the language runtime in order to flatten the objects graph corresponding to a request. The new routine would split the graph, each obtained part being subsequently sent in a new message. Splitting strategies could rely on the algorithm used for traversing the graph (either breadth or depth first);

2. or, if the language allows that a class member function be used for the flatten operation instead of the standard one, a class could define its own customised flatten-splitting routine, in the same spirit as done when defining derived datatypes in MPI. For example, assume one parameter of the request be an instance of a `Matrix` class, the flatten routine overriding the default one could tell independently for each line how to flatten it and when to send it.

Considering the first strategy implies that all arguments that need to be marshalled been split, whatever the potential benefit, i.e. without taking into account the use order of those arguments for instance. Whereas considering the second, while not transparent, gives the opportunity to give a more adequate splitting and even sending order. As such, one can consider that these two strategies lie at the two extremes of the spectrum, while the one using *later* types lies in-between. Indeed, casting an object to *later* or back to its original type, and be careful of argument positions in method signatures is a satisfactory compromise: it is not completely transparent for programmers which thus have some control on the splitting, but it does not require to define a specific marshalling routine for each type, which would be quite boring. So, we decided to only experiment with the strategy which relies on using *later* types.

3 Prototype Environment

We briefly present in this section our implementation of the overlapping mechanism. We use for this a parallel and distributed extension of C++, called C++//, whose runtime is based on communicating lightweight processes using the Nexus library and Globus [9].

3.1 C++//

The C++// language [6] (`http://www.inria.fr/oasis/c++ll/`) was designed and implemented with the aim of importing reuse into parallel and concurrent programming. It does not extend the language syntax, and requires no modification of the C++ compiler, since C++// is implemented as a library and a preprocessor (relying on a Meta-Object Protocol [13] – MOP).

C++// provides a heterogeneous model with both passive and active objects. Active objects act as sequential processes serving requests (i.e. method invocations) in a centralized and explicit manner by default (such objects are instances of subclasses of the specific C++// class `Process`). Communications towards active objects are systematically asynchronous. There are no shared passive objects (only call-by-value between processes, implying making deep copies of request/reply parameters, like serialization in Java RMI).

The MOP is centered around points concerning RPC, where some reification is applied: request send, request receive, reply send, reply receive. These points manipulate requests or replies as first-class objects. Generic flatten and rebuild

functions are used for these objects. The reply of a service invocation is transparently built as a *future*. Access through method invocation to any object of *future* type is reified and blocks the caller if the result is not back yet.

Part of C++// runtime based on NEXUS. NEXUS [10] is a library used for both communications and lightweight processes (threads) in distributed applications, which provides the notion of remote service execution.
A C++// active object is implemented by using a lightweight process on a possibly remote NEXUS *context*. A requests queue of an active C++// object can be remotely referenced thanks to the definition of a NEXUS global pointer. A request for an active object is remotely queued by invoking a remote service at the NEXUS level (named *Queue_a_request*). This service takes as arguments a C++// service request id, and a list of C++// objects as parameters. Each such object is flattened using the generic `flat()` method of C++//. Executing a *Queue_a_request* service implies launching a new thread whose code is the effective queuing of the C++// request in the queue of the target C++// object, after its parameters have been unmarshalled (the generic C++// `build()` method is used for this purpose). Concurrency between request queue filling and request queue extracting is managed with NEXUS local mutual exclusion primitives.

Part of C++// runtime based on GLOBUS. C++//relies on the GRAM mechanism [9] to acquire nodes on a remote host and allocate active objects on a new machine. To help the programmer in this task, C++//provides a simple file to specify the mapping. For example :

```
m0 ll.inria.fr GLOBUS /0/sloop2/dsagnol/ecll/tests/gtk2/sc99Demo_slave
m1 pitcairn.mcs.anl.gov GLOBUS /nfs/dsl-homes02/caromel/sc99Demo_slave
m2 das3fs.tn.tudelft.nl GLOBUS /home/caromel/sc99Demo_slave
m3 bolas.isi.edu GLOBUS /nfs/v6/caromel/sc99Demo_slave
```

The strings *m0* ... *m3* are the virtual names of the machines that we use in the program. With this mechanism, we can change the mapping of the application without recompiling it.

3.2 Implementation of the Overlapping Technique in C++//

At the MOP level, the main modification is to write a new generic function to flatten requests (see Requir. 5): 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 (see Requir. 1), the *Queue_a_request* service is remotely called for the first fragment, while a new defined service *Update_Later* is called for the remaining fragments. The *Queue_a_request* service has been slightly modified in order to manage marks for missing objects (see Requir. 2). The newly defined service *Update_Later* transparently updates the corresponding awaited request parameters (see Requir. 3). As seen here, implementing the overlapping technique requires only minor modifications in the C++// language runtime support.

In order to switch from a *later* type to the original type and vice-versa, the MOP of C++// provides two primitives. Being of type *later* implies being accessed through a proxy, casting to the original type means discarding the proxy and returning a pointer to the original object.

4 Validation

4.1 Benchmark

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.

Program. The test is based on the remote call of the method `OpMatrix::rang()` (see Code 2) 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`.

Code 2 (Definition and use of a C++// remote service with later parameter).

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

```
OpMatrix *dom = CppLL_new(("host"),OpMatrix,());
Matrix *m1 =
    new Matrix(COLUMN, LINE);
Matrix *m2 =
    CppLL_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;
```

The technique should allow to overlap the remote execution requiring only `m1` (i.e. the method `m1→square()`) with 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)`). Compared with an execution not using the overlapping technique, the duration of `m1→square()` should increase, since, at the same time, the remote processor has also to manage the reception and update of the matrix `m2`.

In the framework of this test, we measure various durations (see Fig. 1). The first, *total_duration* is the total duration of the complete call as perceived by the caller. This is the duration that will be reduced using the overlapping technique. Duration $d1$ is the time when using only `m1` in the computation (`m1→square()`), while $d2$ is the time requiring both matrices (i.e. `m2→plus(m1)`). Both computations depend on the matrix size (for simplicity, both matrices are of the same size). Moreover, in order to experiment with longer computations thus with situations where there is more opportunity for some overlapping to occur, the duration $d1$ can vary: a parameter, say $p \geq 1$, is given to the test, and the computation inside `m1→square()` is called p times.

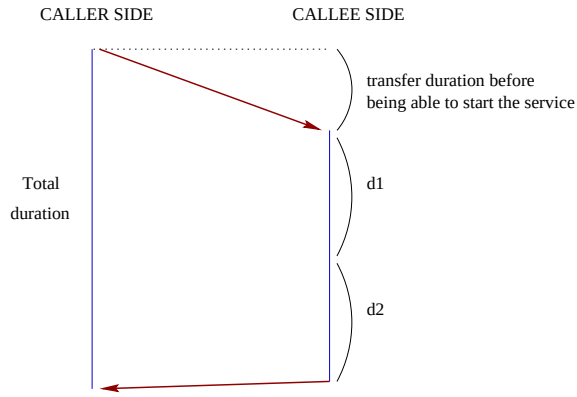

Fig. 1. Temporal decomposition of a – blocking – remote service call

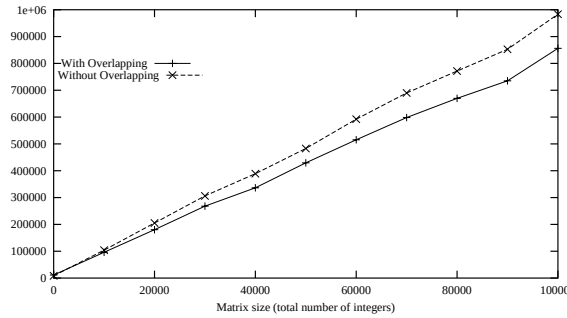

Fig. 2. Execution of the remote service (caller side, total_duration in μs)

4.2 Results using C++//

Apart from proving the correctness of the overlapping technique implementation, we will show that the obtained results are scalable and can yield optimal gains. The formal definition of what we mean by gain will be given at the end of this subsection. We begin by some LAN-based tests (two Sun Solaris 2.6 workstations with 128 MB of RAM, interconnected by a 10 Mbits Ethernet are used), followed by some GLOBUS -based ones.

The two curves plotted in Figs. 2 and 3 show that when the remote computation duration that does not access to *later* parameters increases ($d1$), then the benefit also increases. Indeed, because of the use of lightweight processes, computation using only `m1` and reception of `m2` have more opportunity to be interleaved when $d1$ increases.

Moreover, the reception related operations do not disturb very much the on-going computation (see Fig. 4), although they arise while `m1→square()` is being executed. To claim this, we must be sure that the reception related operations indeed arise while `m1→square()` is being executed (and not latter when

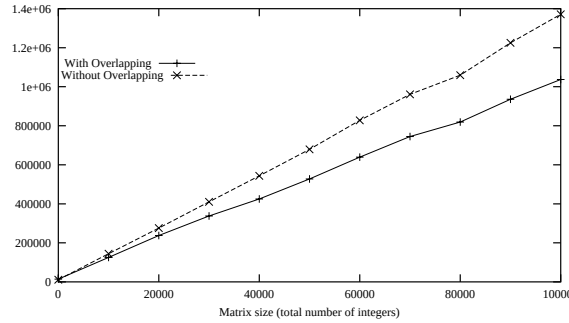

Fig. 3. Execution of the remote service (caller side, total_duration in μs). `m1→square()` ($d1$) is 4 times longer than in Fig. 2

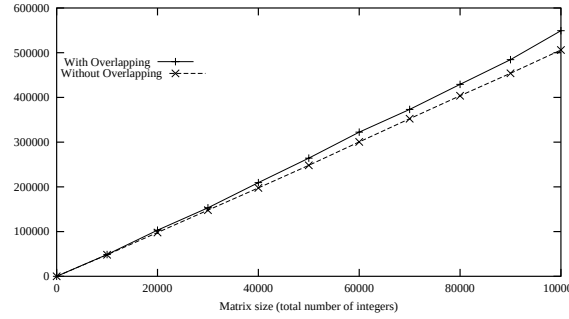

Fig. 4. Duration in μs of the remote computation not using *later* parameters ($d1$). The tests correspond to those of Fig. 3

`m2→plus(m1)` is already started). The answer is given by Fig. 5 where one can observe that almost no reception related operations have arisen in `m2→plus(m1)`.

The overlapping technique used in this context where lightweight processes are available, scales very well, as Fig. 6 shows it. As distributed computing on grid environments is mainly justified by huge data sets, this is an interesting property. 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 [1] 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.

We have also tested the use of a multi-processor workstation for the remote service execution. All experiments we have conducted on this platform occurred while it was unloaded, so that we could assume that at least 2 CPUs were idle. In this case, the computation not using *later* parameters (i.e. `m1→square()`)

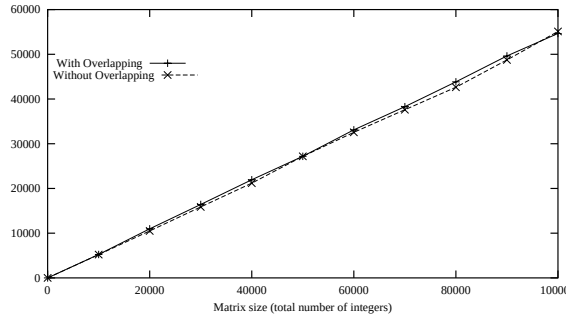

Fig. 5. Duration in μs of the remote computation using *later* parameters ($d2$). The tests correspond to those of Fig. 3

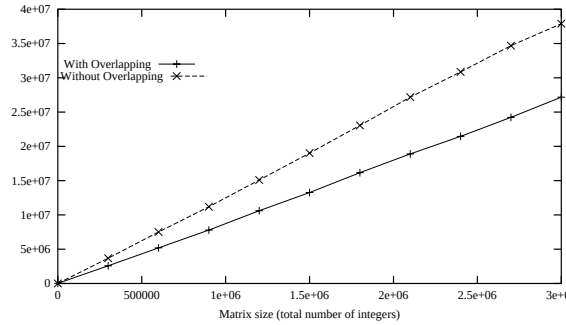

Fig. 6. Execution of the remote service with large matrix sizes (caller side, total_duration in μs). Same configuration as tests of Fig. 3

is absolutely not disturbed compared with experiments where the overlapping technique is not active. This confirms the fact that the reception of *later* parameters is effectively executed in parallel with the computations not using *later* parameters. This gives us confidence that the way the technique is implemented, i.e. based on lightweight processes, provides really concurrent activities that can even be executed in parallel, in this case yielding an unmeasurable overhead.

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

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

$duration_{\ldots\_using\_overlap}$ represents the total duration of the remote service execution in either case (using or not the overlapping technique). 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).

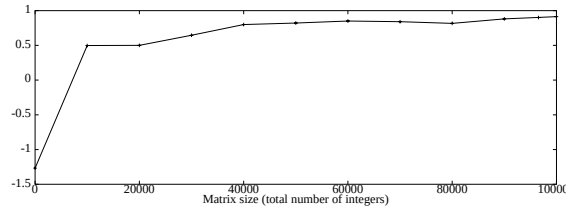

Fig. 7. Benefit (G) obtained from using the overlapping technique on a LAN. Correspond to the tests of Fig. 3

Expected values of G are in $[0, 1[$: it means that the transfer duration of *later* parameters has been overlapped by some useful computation occurring at the callee side (i.e. `m1→square()`). To avoid negative values for G, the only condition is that *later* parameters and computation duration be sufficiently large, such as to mask the – small – overhead of the technique (see Fig. 7). Obtaining a value of G greater than 1 is not related to the overlapping technique but of the variable network loads (especially noticeable on a WAN, see Fig. 9 and [1]).

4.3 Discussion

Using an environment where computation and reception executions are parallel or pseudo-parallel enables to really take advantage of our technique, thus leading to a gain close to the optimal possible value, as computed by G and shown in Figs. 7 and 8.

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 overlapping occurs. This is why the grid-based experiments plotted in Figs. 8 and 9 assigned $d1$ to be 300 times higher than in experiments plotted in Fig. 3. Even if the matrix size was 4 times smaller, this arbitrary choice for such a high value for $d1$ lead to a sufficiently high remote computation duration, in the same order of magnitude as communication delays. It is reasonable to expect that 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.

An other important factor is related to the transmission delays. If they are very low because either the number of transmitted bytes is small, or the network speed is really good as on a LAN, then the technique can yield to a gain but which can prove in fact to be negligible (for instance, if we spare a few milliseconds only). If we now integrate the overhead oft the technique (a few milliseconds of computation time only), then we can see that the benefit (even if optimal if all the transmission has been overlapped) can sometimes be overridden by the overhead. This can effectively arise on LAN-based environments as Fig. 7 plots it for small matrix sizes (observe the negative values for G). On the contrary, on

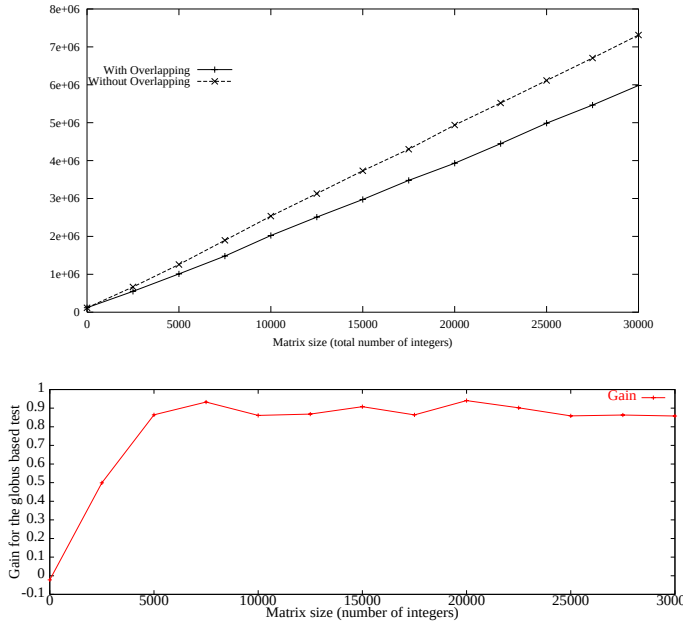

Fig. 8. Execution of the remote service (caller side, total_duration in μs) and corresponding gain. This corresponds to one Globus-based test between Argonne and INRIA during night period, with (d1) 300 times longer than in Fig. 3

WAN-based environments, sparing the transmission time of even a few bytes[2] yields a positive gain that the overhead of the technique can not override (due to so high transmission delays): observe for instance in Fig. 8 the fact that G is greater than 0.

We thus advocate to turn the overlapping technique on for every remote service invocation whose related communications occur on a WAN. Depending on the remote computation algorithm and its parameters usage (which implies how to best split parameters transmission through their cast into *later* type), the benefit can in some cases even rise close to the optimal possible value, i.e. where the whole transmission time of *later* parameters has been spared.

5 Conclusion

In this paper, we have defined and implemented a mechanism to overlap computations with communications in distributed object-oriented languages.

[2] More precisely, the total duration for the test in Fig. 8 using matrices $m1$ and $m2$ of 2500 integers decreases from 680816 microseconds not using the overlapping technique to 556446 microseconds when using it.

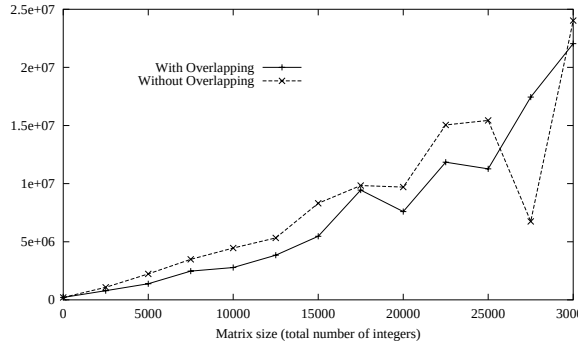

Fig. 9. Execution of the remote service (caller side, total_duration in μs). This corresponds to one Globus-based test between Argonne and INRIA during day period, with (d1) 300 times longer than in Fig. 3

Performances. This mechanism is interesting for environments based on lightweight processes, because they enable to make the transfer of *later* objects parallel with the on-going remote service execution. The technique scales very well, and its use dramatically decreases the total duration of the service execution as soon as operations on non-*later* parameters take enough time to enable the parallel execution of *later* parameters transmission. In this last case, this becomes clearly an advantage for applications running on high latency WANs (see Figs. 8 and 9) where several seconds in transmission time can be spared. Nevertheless, be aware that there is a small overhead when accessing objects of *later* type because the access is reified. An experiments-&-measurements analysis tool could help programmers to decide when to turn the overlapping mechanism on or off. Such a tool could extract the same kind of numerical results than described in Sect. 4 (e.g. extract $d1$, $d2$, ... dn and *total_duration* out of the experiments with or without using the overlapping mechanism, compute the related value for G).

Ease of use. As exemplified in Code 2, the programmer has to manually split data into smaller units, but this only requires to change the type of the parameters (make them inherit from the *later* class). To take advantage of the mechanism, the remote computation does not necessarily need a specific design (or redesign). The only 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. This ease of use is an argument in favour of a systematic usage of the technique, even if the benefits are not always here as they could depend from unpredictable communication durations especially on the grid.

Implementation. The requirement to implement the overlapping technique in an object-oriented distributed language is mainly to have free access to the transport layer and a MOP for the language. If so, essentially only the flatten and rebuild phases of remote procedure calls need to be modified: the object

representing the remote call has to be fragmented into several pieces. Those phases need only to use a mechanism offering a *future* semantic. Transparent reception and management for later fragments is required at the runtime support level. Such a mechanism is of widespread use, and is in particular available in NEXUS, and in PM^2 [14], both of them acting as "low-level" runtime supports for parallel and distributed computations.

References

1. F. Baude, D. Caromel, N. Furmento, and D. Sagnol. Overlapping Communication with Computation in Distributed Object Systems. In HPCN Europe'99 *LNCS 1593*, 744-753, 1999.
2. A.D. Birrell and B.J. Nelson. Implementing Remote Procedure Calls. *ACM Transactions on Computer Systems*, 2(1): 39–59, Feb. 1984.
3. T. Brandes and F. Desprez. Implementing Pipelined Computation and Communication in an HPF Compiler. In *Euro-Par'96*, J:459-462, Aug. 1996.
4. J.-P. Briot, R. Guerraoui and K.-P. Lhr. Concurrency and Distribution in Object-Oriented Programming. *ACM Computing Surveys, 30(3), Sep. 1998.*
5. D. Caromel. Towards a Method of Object-Oriented Concurrent Programming. *Communications of the ACM, 36(9):90-102, Sep. 1993.*
6. D. Caromel, F. Belloncle and Y. Roudier. *Parallel Programming Using C++*, chapter The C++// System, p 257-296. MIT Press, 1996. ISBN 0-262-73118-5.
7. D. Caromel, W. Klauser and J. Vayssiere, *Towards Seamless Computing and Metacomputing in Java*, Concurrency Practice and Experience, 10(11-13), Nov. 1998.
8. F. Desprez, P. Ramet, and J. Roman. Optimal Grain Size Computation for Pipelined Algorithms. In *Euro-Par'96*, T:165-172, Aug. 1996.
9. I. Foster, C. Kesselman. Globus: A Metacomputing Infrastructure Toolkit. *International Journal of Supercomputer Applications*, 11(2):115-128, 1997.
10. I. Foster, C. Kesselman, and S. Tuecke. The Nexus Approach to Integrating Multithreading and Communication. *JPDC*, 37:70-82, 1996.
11. A. Geist *et al.* PVM Parallel Virtual Machine: a user's guide and tutorial for networked parallel computing. *MIT Press, 1994.*
12. R. Halstead. Parallel Symbolic Computing, *Computer*, 19(8):35–43, Aug. 1986
13. G. Kiczales, J. des Rivières, and D.G. Bobrow. *The Art of the Metaobject Protocol.* MIT Press, 1991.
14. R. Namyst and J-F. Méhaut. PM^2: Parallel Multithreaded Machine. A Computing Environment for Distributed Architectures. In *ParCo'95*, Gent, Belgium, Sep. 1995.
15. M. Snir and W. Gropp *et al. MPI: The Complete Reference.* MIT Press, 1998.
16. Sun Microsystems. Java RMI Tutorial, Nov. 1996. `http://java.sun.com`.
17. C.W. Tseng. *An Optimizing Fortran D Compiler for MIMD Distributed-Memory Machines.* PhD thesis, Rice University, Jan. 1993.
18. C. Videira Lopes. Adaptive Parameter Passing. In *ISOTAS'96*, Mar. 1996.