Implementing a parallel algorithm with CORBA: an experience with the WZ factorization

P. Chatonnay, D. Laiymani, J.M. Nicod, and L. Philippe

Laboratoire d'Informatique de Besancon Université de Franche-Comté, Route de Gray F-25030 Besancon cedex, France

Abstract. The aim of this paper is to present the implementation of a classical parallel numerical algorithm using CORBA in the context of parallel programming in networks of workstations (NOWs). Our results on a parallel WZ factorization concerns both programming methods and performances. We show that implementing a typically parallel application with a specific CORBA implementation allows to obtain performances comparable to those obtained by message passing libraries such as PVM and MPI. The performances obtained, particularly in terms of communication costs, are very encouraging for future implementations of parallel applications using CORBA. However, the implementation, in terms of communication scheme, is not obvious.

Keywords: *CORBA, Message passing, NOW, Numerical computing, WZ factorization, Performance comparison.*

1 Introduction

The generalization of interconnected workstations in information systems have lead to a new kind of execution support for parallel applications: the NOWs (Network Of Workstations). On these systems, applications are generally composed of communicating tasks which use a message passing library (for instance PVM [1], MPI [2]) to take advantage of the computing power of the whole workstation set. However, many distributed applications are built upon the client/server model leading to the definition of a standard (CORBA [3]) which provides an execution support standard for distributed object-based applications. This standard sets the services and facilities needed by an object-based application to be executed in a distributed environment. Currently, several implementations of CORBA are available and it is interesting to notice that this standard is more and more used in the industrial community.

With high speed networks, NOWs can be viewed as parallel architectures leading to good performance for very time consuming parallel applications and particularly for parallel numerical applications. In this context, as the same architecture is used for both distributed and parallel applications, why do not use the object programming facilities to develop parallel applications? In this case, the CORBA Object Request Broker (ORB) would be used as a parallel execution support for NOWs in place of classical message passing libraries. Its benefits will mainly be in high level transparency and interoperability mechanisms which will facilitate the distribution management. Moreover, object programming concepts are largely known as a good base for easy to use programming environments.

The aim of this paper is to show that implementing a typically parallel application with a specific CORBA implementation (Chorus COOL ORB [4]) allows to obtain performances comparable to those obtained by the classical message passing libraries (PVM, MPI). We chose the parallel WZ matrices factorization algorithm [5] to perform this comparison. The result of our first experiments make us optimistic in front of the use of CORBA as a parallel execution support environment for NOWs.

In section 2 of this paper, we shortly describe the WZ factorization algorithm and its parallelization. We present, in section 3, its implementation using object design. Section 4 exposes the results, in terms of execution time, of the CORBA implementation of this application. We end by some concluding remarks and open works.

2 The WZ factorization algorithm

Now, we shortly expose the WZ factorization method. As stated above, we have chosen this numerical problem in order to show that parallel object oriented applications may present performances comparable to those obtained by the classical message passing libraries such as PVM or MPI.

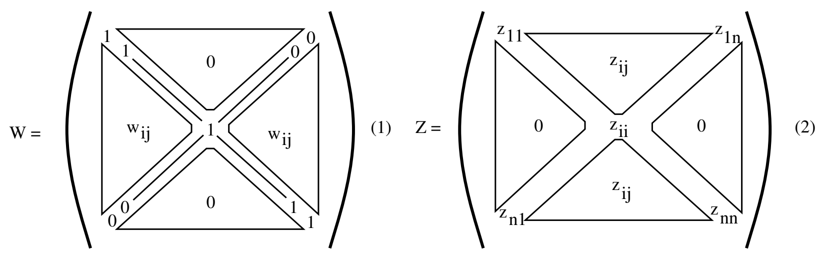

Fig. 1. W and Z matrices

Let A be a $(n \times n)$ non singular matrix. The WZ factorization introduced in [5,6] consists in expressing A in $A = WZ$ where W and Z matrices have the form presented in figure 2. As we will see in the remainder, the WZ factorization is an interesting variant of elimination methods because it is suitable to parallel computations since $O(n)$ steps are required to compute W and Z.

2.1 The sequential algorithm

The different components of W and Z are computed in q steps, $0 \leq k \leq q$, where $q = \lceil \frac{n}{2} \rceil$. During the kth step of the algorithm rows k and $n-k-1$ of Z and columns k and $n-k-1$ of W are computed according to the following equations (for $k \leq j \leq n-k-1$):

$$z_{kj} = a_{kj} \text{ and } z_{n-k-1,j} = a_{n-k-1,j}$$

Now the elements of W are computed by solving the following (2×2) linear systems (for $k \leq j \leq n-k-1$):

$$\begin{aligned} z_{kk}w_{jk} + z_{n-k-1,k}w_{j,n-k-1} &= a_{jk} \\ z_{k,n-k-1}w_{jk} + z_{n-k-1,n-k-1}w_{j,n-k-1} &= a_{j,n-k-1} \end{aligned}$$

As a preparation for the next stage, we must update the A matrix according to the equation below, for $k \leq i,j \leq n-k-1$

$$a_{ij} = a_{ij} - w_{ik}z_{kj} - w_{i,n-k-1}z_{n-k-1,j}$$

2.2 A WZ factorization parallel approach

In [7] Garcia et al propose a specific data decomposition for the parallelization of the WZ factorization in order to balance the workload of the processors. This decomposition assumes a logical grid of $p \times p$ processors and an in-place algorithm (i.e. matrix Z contains matrix A). Solving the (2×2) linear systems of equations, requires to compute a determinant composed of elements of Z which are specularly symmetric with the central axis of the matrix. To avoid data exchange between processors in this operation, it is relevant to store these elements in the same processor. So, it appears that folded matrices used in [7] are particularly appropriate.

The folding consists in partitioning A into four $\frac{n}{2} \times \frac{n}{2}$ matrices, say Z^0, Z^1, Z^2 and Z^3 such that for $0 \leq i,j < \frac{n}{2}$

$$\begin{array}{ll} z^0_{ij} = a_{ij} & z^1_{ij} = a_{i,n-j-1} \\ z^2_{ij} = a_{n-i-1,j} & z^3_{ij} = a_{n-i-1,n-j-1} \end{array}$$

So, the folding represents, with the subscript index, the elements which are specularly symmetric to the four quadrants of the original matrix. The structure of the W matrix used in the parallel algorithm is the same as Z. Furthermore, in order to obtain a good load balancing we adopt a cyclic scheme for the data distribution onto the processor network.

One induced algorithm is proposed and presented by Benaini and Laiymani in [8] (see also [9,10]). At step k the *pivot processor* is the processor which holds

$$Z_k = \begin{pmatrix} z_{kk} & z_{k,n-k-1} \\ z_{n-k-1,k} & z_{n-k-1,n-k-1} \end{pmatrix}$$

In the same way, a processor belongs to the *pivot column* (resp. *pivot row*) of the logical grid if it is in the same column (resp. row) as the *pivot processor*. Note that only the diagonal processor of the logical grid will be pivot processor during the execution of the algorithm. With the data distribution exposed above, the communication scheme of the WZ factorization is as follows. The pivot processor computes the determinant δ of $\begin{vmatrix} z_{kk} & z_{k,n-k-1} \\ z_{n-k-1,k} & z_{n-k-1,n-k-1} \end{vmatrix}$. So all the processors of the pivot rows send to the processors of their columns, their local part of row k and $n-k-1$ of Z. Then processors of the pivot column can compute columns k and $n-k-1$ of W.

Now to update the matrix Z, the processors of the pivot column send to the processors of their row, their local part of column k and $n-k-1$ of W. In this way the matrix Z can be updated.

This algorithm presents an important implicit parallelism since two columns resp. two rows of W resp. Z can be computed during the same iteration (for more details, the reader may refer to [9,7]). Nevertheless, between each iteration a data re-distribution phase has to be performed. Consequently this parallel algorithm is both very CPU and communication time consuming leading it to be a very interesting candidate for the evaluation of a parallel support execution.

The parallel execution of this algorithm with message passing libraries is natural even if the programming effort is important. Now, with an object oriented parallel approach, since communications are performed by synchronous method invocation, this implementation is very different. The next sections show that this way of programming do not affect the overall performances of the application.

3 CORBA Implementation of the WZ factorization

In this part, we describe an implementation of the parallel algorithm using object programming technics and a distributed object platform. First we present the objects architecture of the application and the objects mapping on the execution platform. Then we expose our choices for the interfaces and interactions between objects. Last, we explain the execution concepts of the application on a CORBA platform.

3.1 Architecture of the object implementation

The classical WZ algorithm assumes a $p \times p$ processor grid. Here, the object implementation of the WZ factorization lay on an equivalent object grid. For efficiency reasons, this logical grid will be mapped onto a physical processor grid of the same dimension.

Distributed WZ factorization results in several computing steps where each part of the application cooperate to the final result. Data of the algorithm are distributed onto the set of objects or processors. In the parallel algorithm, each processor works on its own data synchronously with the other processors: at each step, informations are exchanged between processors. However the object invocation model does not provide global synchronization support and this must be done at the programmer level.

In the distributed object implementation, information exchanges between objects are structured by interfaces. This means that each object, involved in the application, provides an interface which describes its public methods. This interface is declared at the application level in order to be accessible to other objects.

3.2 Data exchanges

In distributed object platforms, data exchanges lay on method invocation where data correspond to method parameters. These methods and their parameters are described in the object interfaces in which they are declared. For the grid of objects involved in the WZ factorization, the list of information exchanges is the following:

- the pivot object sends a determinant and lines to every object in the pivot column,
- every object of the pivot line, except pivot itself, sends lines to every objects in their column,
- every object of the pivot column, sends columns to every objects in their lines.

The methods provided to implement these three types of communications are described in an interface file:

```
interface iwz {

  oneway void motor();
  oneway void init();
  oneway void synchro (in long I, in long J);
  oneway void recvDet (in double det, in table lin0, in table lin1,
                                      in table lin2, in table lin3);
  oneway void recvLin (in table lin0, in table lin1,
                       in table lin2, in table lin3);
  oneway void recvCol (in table col0, in table col1,
                       in table col2, in table col3);
};
```

For instance, when an object of the pivot line sends lines to the objects in its column, it invokes one by one all the objects in the column by using the `recvLin` method. When objects are called, they start to execute the corresponding method with the received parameters.

Communication primitives usually provide by communication libraries as PVM or MPI are basically non blocking send and blocking receive. Using these primitives the sending processor does not have to wait for the receiving processor, even if this one is desynchronized with the first one. However, the receiving processor must wait for the information he needs to continue its execution.

The distributed object communication semantic implies method invocation. This means that the calling object, the client, access a method of a called object, the server, to compute results or get some service. Usually this invocation blocks the calling object until the method finish. In our case, the methods provided by the objects are just data transfer methods and we do not have to wait for results. That explain why we use `oneway` methods which are asynchronous and do not return results, as non-blocking sends. The asynchronous message exchange is handled by the system which bufferizes message exchanges and thus allows to compute and communicate in the same time.

At the system level, asynchronous invocations are possible thanks to *threads*. But this also implies to synchronize the threads when they access data: we need a data dependencies evaluation to guaranty data space-time validity.

3.3 Synchronization

At step k of the execution, for $0 \leq k \leq \frac{n}{2} - 1$ and for a logical grid of $p \times p$ objects, let $i = k\%p$:

- The pivot object (i, i) computes its determinant and sends lines to the pivot column,
- the other objects (i, c) (with $0 \leq c \leq p - 1$ and $c \neq i$) of the pivot line send lines to objects (l, c) (with $0 \leq l \leq p$ and $l \neq i$) of their column,
- the objects (l, i) of pivot column compute columns k and $n - k - 1$ of W and send them to objects (l, c) (with $0 \leq l,\ c \leq p - 1$) of their line,
- all the object update matrix Z with the received data and then start step $k + 1$.

To respect computing steps and guaranty the data validity, it's mandatory to put synchronizations inside objects and between objects at the application level. The first synchronization level, is due to objects (c, l) (with $0 \leq l$, $c \leq p - 1$) which must wait to receive lines and columns before starting to compute and update Z matrix. The second synchronization level, is due to the next $k + 1$ application step that must not be started until all the objects are finished with current step k.

Object level synchronizations Figure 3.3 shows communications and synchronizations between objects (i, i), (i, c), (l, i) and (l, c) (with $0 \leq c, l \leq p-1$

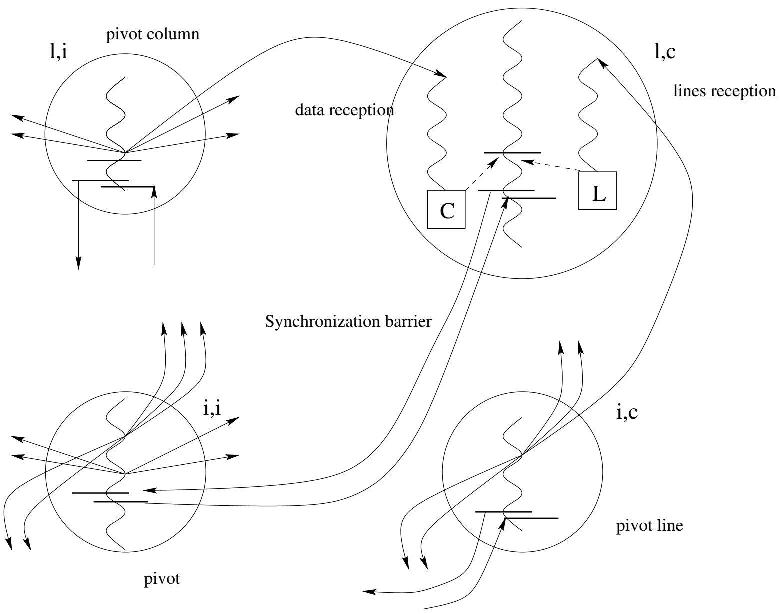

Fig. 2. Communication scheme of the CORBA WZ factorization algorithm

and $c \neq i$). Synchronizations steps, at the object level, are simple and based on flags: for each object, the flag has to be up to start computing current step. This flag is marked up when the $receiveLine$ and $receiveColumn$ methods are invoked. This synchronization is mandatory to implement blocking receive semantic.

Application level synchronization The application level synchronization, between objects, is a synchronization barrier. This means that objects execution can not continue until all the objects have reached this execution point. This global synchronization does not depend on the parallel algorithm, it has been imposed by the object execution model based on method invocation. Indeed there is no support to avoid an object executing step i to invoke an object just executing step $i-1$.

Implementing a synchronization barrier needs a lock at the object level and to gather semaphores in the pivot object. These locks are free as soon as the pivot object get them all. This synchronization is basically implemented as two invocations: one from objects to pivot and the other from pivot to objects. In the interface file, we can note the *synchro* method is there to implement this synchronization. In the pivot object this method increments a semaphore. In all the objects this method is invocated to stop them waiting.

3.4 Mono or multi-thread implementation

Even if it leads to synchronization issues, using a multi-threaded ORB is mandatory to implement parallel applications. Indeed, performing communications by object invocations means that objects have to wait for invocations to receive data. With a mono-threaded ORB, no server may be running concurrency with a receiving thread. This means that an object can not receive data while it runs. On the other hand, objects are only activated on invocations. In that case, the parallel algorithm structure must be deeply modified. Objects must realize asynchronous chained data sends to avoid idle time on other objects. This leads to very complex scheduling issues as we learned from practice.

Developing parallel algorithms under mono-threaded ORBs is very complex. We aim at providing a parallel execution environment which facilitates the implementation of parallel application on PC clusters through libraries for CORBA developer. Moreover our experiments show that the execution of such implementation leads to very fine and complex scheduling issues and thus to unstable execution times. Indeed there is no simple way to provide synchronizations on these mono-threaded ORBs.

3.5 Starting the application

Starting the application under CORBA needs an initialization step that we describe in this part. The application is started by a server object called *launcher*. This server starts basic WZ objects on each node involved in the parallel execution. As described in the CORBA specification a $FactoryFinder$ is running on each site. This server is used to find the reference of a WZ object $Factory$ on each node. Then, the *launcher* invokes the WZ $Factory$ on each node where it creates a WZ object.

After creating its interface, each object declares its object reference to the *namingservice*. This is useful to provide dynamic links between objects. In a second step, the *launcher* imports all the object references and invokes first the *init* method and then the *motor* method which will run the application.

4 Experiments

To evaluate the execution performance of our CORBA environment, we have compared its execution with the execution of a standard WZ implementation under PVM. In this part, we expose some preliminary results concerning the implementation of the WZ factorization.

We have driven tests to measure the application execution time in function of the size of the matrix to be factorized. Figure 3 presents the results of our tests. These first tests are limited to a matrix which size is less than 500 because of the platform size and memory use. Experiments have been

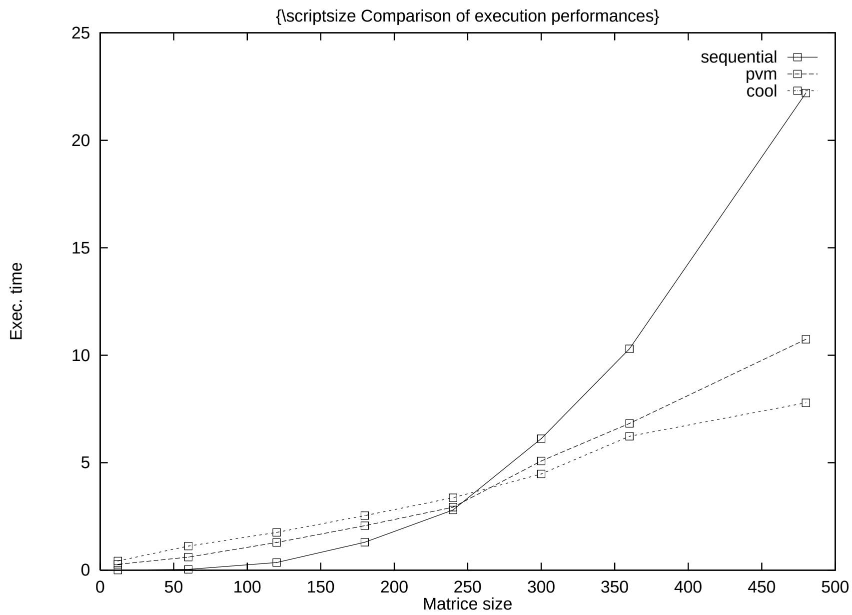

Fig. 3. Execution times

performed on a NOW composed of 4 PCs under Solaris 2.5 with 166 Mhz Pentium processors and interconnected via a 10 Mbits Ethernet network. The network has been isolated to avoid extern perturbations and during the tests the computers were only executing these applications.

Under matrix size of 300, sequential time is better than the PVM application time which is better than the CORBA application time. When the matrix size is greater than 300 the tendency is reversed: the CORBA application get better execution time than the PVM application. It seems that the application benefits from the communication layer performance above this value and for large messages. Our interpretation of these results is that the parallel object application has a important startup cost ($Factory$ search and access, creation, declaration and importation of interfaces) which is compensated by communication performances above a size of 300, but it has still to be confirmed.

Results obtained for 480 sized matrix show a efficiency of 30% for our CORBA implementation. These results are significant and allow to assert that efficient implementations of parallel programs can be realized under CORBA. However there are two limitations to these implementations: there is no tools providing a true parallel programming environment and the ORB as to be chosen with care.

5 Conclusion

The object distributed execution environment that we use presents many advantages. In addition to the development facilities of object and multi-thread programming, its performance in terms of communications are comparable to MPI and better than PVM. Moreover, the implementation of a WZ factorization shows the efficiency of a CORBA platform to execute parallel programs.

Our work shows that efficient CORBA platforms exist. By considering that object oriented development is well known to provide good properties in comparison to classical imperative programming. Thus, using a CORBA platform may be considered as an alternative when implementing parallel programs.

From our knowledge there is no CORBA implementation providing true parallel programming support. For instance, we do not have global broadcast or synchronization as barrier primitives. The CORBA standard allows the specification of domain interfaces, these domains are sets of interfaces dedicated to a particular application field. So it may be interesting to specify in a parallel domain the needed interfaces and system services to provide a comfortable parallel programming environment. We plan to investigate in that direction.

References

1. Al Geist, Jack Dongarra, Robert Manchek, and all. PVM user's guide and reference manual. Oak Ridge National Laboratory, Mai 1994.
2. M. Snir, S. Otto, S. Huss-Lederman, D. Walker, and J. Dongarra. *MPI : The complete reference*. MIT Press, 1995.
3. The OMG Consortium. Corba architecture specifications. Technical Report 97-06-15, Object Management Group, 1997.
4. Christian Jacquemot. Chorus/cool orb, programmer's guide. Technical Report CS/TR-96-2.2, Chorus Systems, 1996.
5. D.J. Evans and al. The parallel solution of banded linear equation by the new quadrant interlocking qif method. *Int. J. Comp. Math*, 9:151–162, 1981.
6. D.J. Evans. *Parallel Numerical Algorithms for Linear Systems*. Cambridge Univeristy Press, 1982.
7. I. Garcia and al. Parallel quadrant inter-locking factorization on hypercube computer. *Parallel computing*, pages 87–100, 1990.
8. D. Laiymani. *Programmation parallèle multi-phase : implantations et ordonnancements de graphes de tâches*. PhD thesis, Université de Franche-Comté, 1997.
9. A. Benaini and D. Laiymani. Parallel Block Generalized WZ Factorization on a Reconfigurable Machine. In *Proceedings of International Conference on Parallel and Distributed Systems -ICPADS'94*. IEEE Computer Society, Taiwan - December 1994.
10. A. Benaini and D. Laiymani. WZ factorization for reconfigurable architectures. Unpublished, 1996.