The Journal of Supercomputing, 35, 227–244, 2006
© 2006 Springer Science + Business Media, Inc. Manufactured in The Netherlands.

Performance Comparison of Parallel Programming Environments for Implementing AIAC Algorithms

JACQUES M. BAHI jacques.bahi@univ-fcomte.fr
SYLVAIN CONTASSOT-VIVIER sylvain.contassot-vivier@iut-bm.univ-fcomte.fr
RAPHAËL COUTURIER raphael.couturier@iut-bm.univ-fcomte.fr
Laboratoire d'Informatique de Franche-Comté (LIFC), IUT de Belfort-Montbéliard, BP 527, 90016 Belfort, France

Abstract. AIAC algorithms (Asynchronous Iterations Asynchronous Communications) are a particular class of parallel iterative algorithms. Their asynchronous nature makes them more efficient than their synchronous counterparts in numerous cases as has already been shown in previous works. The first goal of this article is to compare several parallel programming environments in order to see if there is one of them which is best suited to efficiently implement AIAC algorithms. The main criterion for this comparison consists in the performances achieved in a global context of grid computing for two classical scientific problems. Nevertheless, we also take into account two secondary criteria which are the ease of programming and the ease of deployment. The second goal of this study is to extract from this comparison the important features that a parallel programming environment must have in order to be suited for the implementation of AIAC algorithms.

Keywords:

Introduction

Iterative algorithms are very well suited for a large class of scientific problems and, in many cases, they are the single way to solve the problem. Their principle is to achieve the solution of a problem by successive approximations. Although some conditions must be verified to ensure that this process works well, it is not actually a great obstacle since these conditions are verified for many scientific problems.

Another interesting point is that these algorithms can be modified to run in parallel rather easily. Nevertheless, the classical parallel iterative algorithms are synchronous. We have shown in [5, 6] all the interest of using asynchronism in such algorithms especially in a global context of grid computing. These works together with [4] have led to the definition of a particular class of parallel iterative algorithms which we call AIAC. This acronym stands for Asynchronous Iterations Asynchronous Communications.

The first goal of this article is to find out if there is a particular parallel programming environment which is best suited to implement our AIACs algorithms. We have chosen to compare three environments which are well-known in the parallelism community and which propose rather different conceptual views: MPI, PM2 and Corba. In this comparison, we will discuss the ease of the implementation of AIACs, the ease of deployment over the grid and the efficiency aspects. The discussion will be illustrated by several experiments on two representative kinds of scientific problems, each of them using a different communication scheme.

Using these results, our second goal is to deduce the features required for a parallel programming environment to be suited for the implementation of AIAC algorithms.

The following section recalls the principle of asynchronous iterative algorithms and replaces them in the context of parallel iterative algorithms. In Section 2, we present a brief analysis of the programming environments used for the implementations of AIACs in our previous works. A brief description of the three tested parallel programming environments is given in Section 3. Section 4 gives the description of the two scientific problems chosen to make our experiments. Then, the comparison of the three parallel programming environments is presented in Section 5 together with experimental results. Finally, the important features required for an efficient implementation of AIACs are given in Section 6.

1. Parallel iterative algorithms

1.1. Synchronous parallel iterative algorithms

Iterative algorithms have the following structure

$$x^{k+1} = g(x^k), \qquad k = 0, 1, \dots \text{with } x^0 \text{ given} \tag{1}$$

where each x^k is an n-dimensional vector, and g is some function from $I\!R^n$ into itself. If the sequence $\{x^k\}$ generated by the above iteration converges to some x^* and if g is continuous then we have $x^* = g(x^*)$, we say that x^* is a fixed point of g.

Let x^k be partitioned into m block-components X_i^k, $i \in \{1, \dots, m\}$, and g be partitioned in a compatible way into m block-components G_i, then Equation (1) can be written as

$$X_i^{k+1} = G_i\left(X_1^k, \dots, X_m^k\right) \quad i = 1, \dots, m, \text{ with } X^0 \text{ given} \tag{2}$$

and the iterative algorithm can be parallelized by letting each of the m processors update a different block-component of x according to (2) (see [18]). At each stage, the ith processor knows the value of all components of X^k on which G_i depends. It computes the new values X_i^{k+1} and communicates those on which other processors depend to make their own iterations. The communications required for the execution of iteration (2) can then be described by means of a directed graph called the dependency graph.

1.2. Asynchronous parallel iterative algorithms

Fully asynchronous networks including overlapping updating were characterized by Herz and Marcus in [12]. The model is as follows:

- the block nodes of the network may be updated in a random order and some nodes may not be updated at some times. Nevertheless, no block is permanently idle.
- at time t, each node updates its own state using the last received information from its dependencies rather than waiting for those at time $t - 1$.

This model implies two additional definitions. The first one is the set $J(t)$ of activations which contains all the nodes updated at time t. The second one is the delay of block j according to block i (noted $r_j^i(t)$) which directly intervenes in the version of data available on node i at time t coming from node j (noted $s_j^i(t) = t - r_j^i(t)$).

Then, the fully asynchronous dynamic of the n-nodes network associated to the transition function G, the activations sets $J(t)$ and with initial configuration $X^0 = (X_1^0, \ldots, X_m^0)$ is described by Algorithm 1.

Algorithm 1: Asynchronous iteration

```
Given an initial state $X^0 = (X_1^0, \ldots, X_m^0)$
for each time step $t = 0, 1, \ldots$ do
  for each block-component $i = 1, \ldots, m$ do
    if $i \in J(t)$ then
      $X_i^{t+1} = G_i(X_1^{s_1^i(t)}, \ldots, X_m^{s_m^i(t)})$
    else
      $X_i^{t+1} = X_i^t$
    end if
  end for
end for
```

It is interesting to note that this model is the most general form of parallel iterative algorithms. In this model, the residual of block i is defined by the max norm of the difference between its values from two consecutive iterations:

$$\text{residual}_i^t = \left\| X_i^t - X_i^{t-1} \right\|_\infty = \max_j \left| X_{i,j}^t - X_{i,j}^{t-1} \right|$$

where $X_{i,j}^t$ is the jth component of the block-vector X_i^t.

1.3. A categorization of parallel iterative algorithms

A more complete classification of parallel iterative algorithms can be found in [4, 6]. Here, we simply give the definitions of the synchronous and asynchronous algorithms which are directly within the scope of this study.

The SISC (Synchronous Iterations—Synchronous Communications) algorithms are characterized by the fact that all the processors begin the same iteration at the same time. Data exchanges are performed at the end of each iteration by synchronous global communications. By their synchronous nature, those algorithms present the advantage to perform exactly the same iterations as the sequential version. Thus, they keep the same convergence properties although the computations are made over several processors, speeding up the execution time. Unfortunately, the synchronous communications also have the drawback of strongly penalizing the performances of these algorithms. Hence, a lot of idle times (white spaces) may appear between the iterations (grey blocks) in the execution flow of such algorithms as shown in Figure 1.

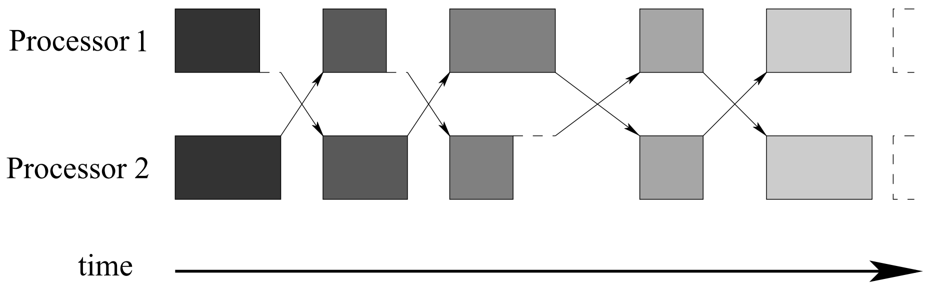

Figure 1. Execution flow of a SISC algorithm with two processors.

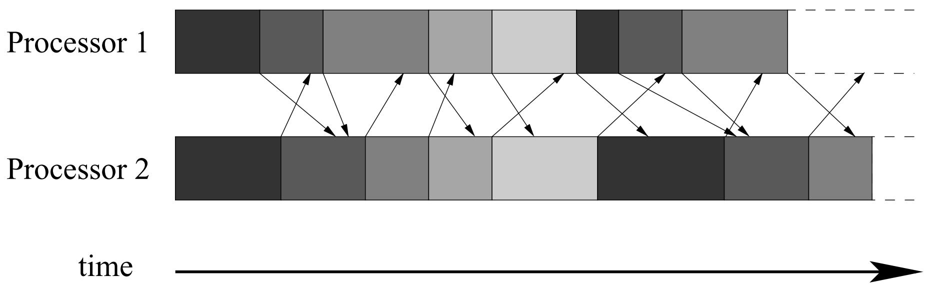

Figure 2. Execution flow of an AIAC algorithm with two processors. The arrows represent the data communications.

In AIAC (Asynchronous Iterations—Asynchronous Communications) algorithms, all processors perform their iterations without caring about the progress of the other processors. They do not wait for predetermined data to become available from other processors but they keep on computing, trying to solve the given problem with whatever data happen to be available at that time. Since the processors do not wait for communications, there is no idle time between the iterations as can be seen in Figure 2.

Although widely studied theoretically, very few implementations and experimental analyses have been carried out, especially in the context of grid computing. In the literature about this domain, there are two algorithmic models corresponding to these algorithms, the Bertsekas and Tsitsiklis model [8] and the El Tarazi's model [17]. Nevertheless, several variants can be deduced from these models depending on when the communications are performed and when the received data is incorporated in the computations, see e.g. [3, 9, 10]. Figure 2 depicts the execution flow of a general version of an AIAC. This type of algorithms has the interesting property of allowing communication delays to be substantial and unpredictable, which is a typical situation in large networks of heterogeneous machines. These algorithms are then very well suited to a global context of grid computing. Nevertheless, they require a meticulous study to ensure their convergence because even if a sequential iterative algorithm converges to the right solution, its asynchronous parallel counterpart may not converge. It is then needed to develop new converging algorithms and several problems appear like choosing the good criterion for convergence detection and the good halting procedure. There are also some implementation problems due to the asynchronous communications which imply the use of an adequate programming environment. That is why we want to compare existing parallel programming environments and deduce the general features required in such an environment to allow efficient implementations of AIACs.

2. Analysis of previous implementations

In our first tries to implement AIAC algorithms, we naturally turned towards classical parallel non-threaded libraries such as MPI [11]. Nevertheless, we rapidly found that the common version of this library was not very well suited to implement AIACs. This mainly came from the fact that in MPI, the receipts of messages must be explicitly localized in the sequence of the program whereas in AIACs we want to be able to receive messages at any time. In fact, the necessity to precisely specify *when* a message must be received is a very strong constraint according to AIACs which sharply decreases their flexibility. Hence, even if the implementation is not impossible in classical mono-threaded MPI, it is neither very convenient to write nor very efficient.

The conclusion of this first experience was the need of a multi-threaded parallel library for the implementation of AIACs.

Then, we have chosen to use the PM2 [13] library which provides a multi-threaded environment. The first goal of this library is to efficiently support irregular parallel applications on distributed architectures. It is designed to manage numerous threads on each processor and thread migration mechanisms, allowing the management of high degree of parallelism and dynamic load balancing.

Thus, the multi-threading provided in PM2 has brought us a far more convenient way to write our algorithms and also far better performances. Indeed, using threads to send and receive messages allowed us to implicitly perform an efficient overlapping of computations over communications. Thus, all the potential power of the AIAC model could be used.

These first experiences have pointed out that multi-threading is an essential feature to efficiently implement AIACs. Nevertheless, we need a more complete analysis to see if other features appear to be necessary and if a particular existing environment is best suited for the implementation of AIAC algorithms.

3. Tested environments

The three environments which have been chosen to be compared are PM2 [13], MPICH/Madeleine [1] and OmniOrb 4 [14].

The first one has been naturally selected since, as said in Section 2, it is the one we have been using up to now. It is a portable environment available on a wide range of architectures. Its implementation is built on top of two separate software components : Marcel and Madeleine. Marcel is a POSIX-compliant thread package. Madeleine is a generic communication interface which can be used on top of different communication protocols such as VIA, BIP, SBP, SCI, MPI, PVM and TCP. In our context, we used it over TCP.

The second environment is a multi-protocol version of MPICH which also uses Marcel. Thus, it provides multi-threading functionalities inside an MPI implementation. This environment has been chosen according to the wide use of MPI and the fact that it is thread safe. The comparison to PM2 is relevant since although they use the same thread manager, their communications are performed using different schemes (Explicit communication with MPI and Remote Procedure Call with PM2).

Finally, the last environment is the free Corba [15] ORB OmniOrb 4. It is a robust and high performance ORB which is certified to be in compliance with Corba 2.1. It may be surprising to use an ORB to implement parallel algorithms since this does not correspond to what they are designed for. Nevertheless, Corba in general and OmniOrb 4 in particular present both the minimal features necessary to make these implementations: a communication system between machines and a multi-threaded environment. Our choice has been to take this particular ORB since we already had to use it in another context, so we know it well, and it has one of the most efficient communication management among the free available ORBs. The comparison to the two other environments is also relevant since it uses a different thread manager and an object oriented communication scheme.

A few other environments could also have been tested. Nevertheless, they are generally not designed to take into account the performance aspects. A good example is the JACE library [2] which has been especially written to implement AIAC algorithms. The goal of the authors was to provide an easy and efficient way to implement those algorithms with the portability feature. For this purpose, the library has been written in JAVA. It is obviously not reasonable (currently) to compare an algorithm implemented with this library to the same algorithm implemented in compiled languages like C/C++.

4. Test problems

To be able to compare the different programming environments, we have chosen two representative examples of scientific problems. The first one is a sparse linear system and the second one is a two dimensional system of Partial Differential Equations (PDEs).

4.1. Sparse linear system

The problem is of the form

$$Ax = b \tag{3}$$

where A is a square sparse matrix, b is a known vector and x is the unknown vector which has to be computed.

The method chosen to solve this problem is the fixed-step gradient descent. Its principle is to use an iterative process which computes the successive approximations of x by using the inverse of the block-diagonal matrix M extracted from A and the residual between two consecutive approximations of x.

The formulation is then as follows

$$x^{k+1} = x^k + \gamma M^{-1}(b - Ax^k) \tag{4}$$

where γ is a real constant which must be conveniently chosen (around 1) to accelerate the convergence. For $\gamma = 1$, we obtain the Jacobi method. The process is initiated with an arbitrary vector x^0.

The resolution of the sparse linear system in Equation (3) is achieved by the convergence of the iterative process in Equation (4). This convergence is obtained at iteration k for a given norm *norm* and a given $\epsilon > 0 \in I\!R$ when

$$\text{norm}(x^k, x^{k-1}) < \epsilon \tag{5}$$

In our case, the *max* norm presented in Section 1.2 has been chosen. In the same formulation as Equation (5), this gives:

$$\text{norm}(x^k, x^{k-1}) = \max_i \left|x_i^k - x_i^{k-1}\right| \tag{6}$$

4.2. Non-linear chemical problem

In this problem, we want to compute the evolutions of the concentrations of two chemical species in a two dimensional domain. This problem corresponds to an advection-diffusion system with two species. It is solved by using a discretization of the space on a two-dimensional grid (x, z).

The evolutions are given in each point (x, z) of the grid by

$$\frac{\partial c^i}{\partial t} = K_h \frac{\partial^2 c^i}{\partial x^2} + V \frac{\partial c^i}{\partial x} + \frac{\partial}{\partial z} K_v(z) \frac{\partial c^i}{\partial z} + R^i(c^1, c^2, t) \tag{7}$$

where $i = 1, 2$ denotes the number of the chemical species and

$$\begin{aligned} R^1(c^1, c^2, t) &= -q_1 c^1 c^3 - q_2 c^1 c^2 + 2q_3(t) c^3 + q_4(t) c^2 \\ R^2(c^1, c^2, t) &= q_1 c^1 c^3 - q_2 c^1 c^2 + q_4(t) c^2 \end{aligned} \tag{8}$$

with

$K_h = 4.0 \times 10^{-6}$	$V = 10^{-3}$
$K_v(z) = 10^{-8} e^{\frac{z}{5}}$	$c^3 = 3.7 \times 10^{16}$
$q_1 = 1.63 \times 10^{-16}$	$q_2 = 4.66 \times 10^{-16}$
$q_j(t) = e^{-a_j/\sin(\omega t)}$	for $\sin(\omega t) > 0$
$q_j(t) = 0$	otherwise

and $j = 3, 4$, $\omega = \pi/43200$, $a_3 = 22.62$ and $a_4 = 7.601$.

The time interval is $[0, 7200s]$ and the initial conditions are the following

$$\begin{aligned} c^1(x,z,0) &= 10^6\alpha(x)\beta(z) \\ c^2(x,z,0) &= 10^{12}\alpha(x)\beta(z) \end{aligned} \tag{9}$$

with

$$\begin{aligned} \alpha(x) &= 1-(0.1x-1)^2+(0.1x-1)^4/2 \\ \beta(z) &= 1-(0.1z-1)^2+(0.1z-4)^4/2 \end{aligned} \tag{10}$$

Finally, the discretization along x and z allows us to rewrite the system of PDEs in Equation 7 in a system of ODEs (Ordinary Differential Equations) of the form

$$\frac{dy(t)}{dt} = f(y(t),t) \quad \text{with } y=(c^1,c^2) \tag{11}$$

The global solution of Equation (11) is computed using the finite differences scheme. We use the Euler implicit method to realize the discretization over the time interval and the iterative method of Newton to solve the equation for each time step.

Thus, Euler implicit gives us:

$$y(t+h)-y(t) = h.f(y(t+h),t+h) \tag{12}$$

and by posing

$$G(t) = y(t+h)-y(t)-h.f(y(t+h),t+h)$$

the solution of Equation (12) can be obtained by solving $G(t) = 0$ using the Newton iterative method. Each step of the Newton process requires the resolution of a linear system which is performed by the iterative method of GMRES.

Thus, the global solution is given by a loop over the time steps. And each time step is iteratively computed by the Newton method in which a linear system is used to yield its successive approximations.

The method presented above is formulated in the sequential case. In the parallel context, there are two main strategies to perform the Newton method. The first one consists in applying the Newton method on the entire system and using an asynchronous parallel linear solver over the global system. Unfortunately, synchronizations are necessary between two consecutive iterations of the Newton process. The second one is called multi-splitting Newton. In this approach, the set of data to compute is decomposed into several sub-sets which are locally solved by a sequential linear system solver according to data dependencies between the sub-sets. In this method, only one synchronization is needed at each time step (see for example [7] for further information about this method). In our case, the multi-splitting approach has been chosen with the GMRES method [16] as the sequential linear solver.

4.3. Implementations of the corresponding asynchronous algorithms

We consider to have a parallel machine (cluster...) of N processors numbered from 0 to $N - 1$. Each processor can send and receive messages from all others.

The general and classical idea is to decompose the set of values to compute in subsets which are distributed over the processors.

4.3.1. Sparse linear system. Concerning the first problem, the matrix and vectors are vertically decomposed and distributed over the processors. Since the matrix is sparse, each processor needs to construct the list of its data dependencies from other processors. These dependencies are the values, owned by other processors, which are required for the computation of the local data.

The first step of the algorithm then consists in computing the dependencies on each processor and communicating them to all others.

Once this is done, the iterative process can begin. Since the iterations are performed asynchronously, only the first iteration begins at the same time on all the processors. Then, on each processor the successive approximations of the local part of x are computed using the last available values of its dependencies. At each iteration, once new values of local data are computed, they are asynchronously sent to other processors which need them for their own computations. Data are actually sent only if any previous sending of the same data to the same destination is terminated. Otherwise, the sending is not performed at this iteration but is delayed to the next iteration. The receipts of data dependencies are managed in separated threads allowing to perform them at any time during the process and without blocking the computations. As soon as data are received, they are taken into account in the computations.

The last problem in the design of our algorithm is its termination. We have to stop the process only when the global convergence has been detected. We consider this is the case when all the processors have reached what we call their local convergence. The local convergence is achieved on a processor when the convergence criteria is verified for all its local data. In our case, it corresponds to the norm of the residual between two consecutive approximations. A centralized method has been chosen for the global convergence detection. It consists in charging one of the processor to gather the states of all the processors and to detect if they are all in local convergence. When this is the case, it sends a signal to all the others indicating them to stop the computations. To avoid overloading of the network, each processor sends its state only when it changes. By the continuous nature of the computations, oscillations in the residual are possible and then local convergence may be alternatively detected and canceled. This is why each processor may send several state messages. Finally, to avoid too many state messages in the network, we count a specified number of iteration under local convergence before assuming it has actually been reached. Since all these state messages are sent and received asynchronously, and the detection of global convergence is a very small computation, the overloading of the central node is negligible.

Finally, a limit is set over the number of iterations in order to avoid infinite execution when the process does not converge.

4.3.2. Non-linear chemical problem. For the second problem, the two-dimensional domain is vertically decomposed into horizontal strips. Due to the nature of the problem, the computation of the concentrations in (x, z) depends only on its direct neighbors at the top, bottom, left and right sides. They correspond to the $(x, z-1)$, $(x, z+1)$, $(x-1, y)$ and $(x+1, y)$ points in the grid. Thus, it can easily be seen that the use of a linear topology of the network implies that a given processor will have its dependencies coming only from its two direct neighbors.

As said before, there is a main loop which goes through the time interval. Then, each time step is computed by two nested iterative processes. The outer one corresponds to the iterative method of Newton and the inner one corresponds to the resolution of the linear system at each step of the Newton process, with the GMRES iterative method. To start the computation of a given time step, all the concentrations at the previous time step must be known, hence the use of a synchronization of all processors between each time step. On the contrary, the computations inside a time step can be performed asynchronously.

When the process runs, a barrier is performed to synchronize the processors for each time step. Then, each processor computes the values of its local strip of 2D data using the last available values of its dependencies. Once the iteration is performed, it asynchronously sends its new local values to its two neighbors and starts a new iteration. Here again, data are actually sent only if there is no identical communication in progress. As in the sparse linear problem, data dependencies are received asynchronously in separated threads and are immediately taken into account. The termination of each time step is performed by the same convergence detection principle as the one used in the previous algorithm for the sparse linear problem. Here also, a limit is set over the number of iterations for each of the nested iterative processes, in order to avoid infinite execution when one of these processes does not converge. Finally, when the global convergence is detected on the central node, stop messages are sent to all the nodes which leave the inner iterative process as soon as they receive the message. Afterwards, a barrier is performed to synchronize all the nodes and start the computation of the next time step and so on until the end of the time interval.

5. Comparison of the environments

Both AIAC algorithms presented in the previous section have been implemented using the three parallel programming environments: PM2, MPICH/Madeleine and OmniOrb 4.

In order to have a representative comparison of those environments, the following conditions have been respected to make the implementations:

- For each problem, keep the same algorithmic scheme between the implementations in each environment:
  - same computation scheme
  - same communication scheme (same asynchronous sending locations in the algorithm and same management of received data)
  - same convergence detection (same accuracy threshold)
  - same halting procedure

- For each environment, choose the most efficient implementation according to the technical constraints of this environment. In fact, the difference only occurs in the management of the communications: the number of threads used to perform the communications and their creation schedule.

As said in the introduction, this comparison mainly concerns the performances obtained for each of them, but it also addresses the ease of implementing those algorithms and the ease of deploying them onto the grid.

5.1. Performances

In order to make a representative comparison, three series of tests have been conducted for the non-linear problem. The first series has been performed on a heterogeneous cluster of machines scattered on three distinct sites and connected by 10 Mb Ethernet links. This configuration has been chosen since it represents the most common test environment for most of the researchers who work on grid computing. The second series has also been done on an heterogeneous cluster of machines but this time, they were scattered on four sites with some of them connected by an ADSL link (512 Kb/s in reception and 128 Kb/s in sending). These links are far slower than the Ethernet ones. Thus, this configuration is representative of a difficult case (and probably the most common one) of grid environment. Finally, the third series has been conducted on a local heterogeneous cluster with three kinds of machines: Duron 800 Mhz, Pentium IV 1.7 Ghz and Pentium IV 2.4 Ghz.

Concerning the sparse linear problem, only the first cluster has been used because it does not make sense to make this kind of computations on very slow networks since the ratio between communication and computation is rather high. In fact, the computations at each iteration are quite fast whereas the amount of data transfers are important and the communication scheme is all to all according to data dependencies.

For the non-linear problem, the amount of computations is much more important in each iteration whereas the communication scheme is only between direct neighbors and the amounts of data transfers are smaller although still important.

In order to have representative times, the chosen parameters for the problems are given in Table 1. Moreover, in order to ensure the convergence of the iterative process in the asynchronous mode, the sparse matrix is designed to have a spectral radius less than one.

The results of our experiments for the sparse linear problem are presented in Table 2 and those for the non-linear problem in the context of distant clusters are given in Table 3. Each of those results is an average of a series of ten executions.

Several interesting remarks can be made on these results. The first one is that all the asynchronous versions are always faster than the classical synchronous ones with MPI as shown by the speed ratios. This is not very surprising since we have already shown in previous studies [5, 6] that AIAC algorithms are more efficient especially in the grid context.

Concerning the asynchronous versions, although the environments obtain different results, the ranges of execution times are not too large. So, it can be said that the tested environments globally have the same behavior with AIAC algorithms.

Table 1. Chosen parameters for each problem

Parameter	Value
Sparse linear system	
matrix size	2000000 × 2000000
repartition of non-zero values	30 sub-diagonals
Non-linear problem	
discretization grid	600 × 600
time interval	2160s
time step	180s

Table 2. Execution times (in seconds) for the sparse linear problem

Version	Execution time	Speed ratio
synchronous MPI	914	1
asynchronous PM2	551	1.66
asynchronous MPI/Mad	672	1.36
asynchronous OmniOrb 4	507	1.80

Table 3. Execution times (in seconds) on each cluster for the non-linear problem

Cluster	Version	Execution time	Speed ratio
Ethernet	sync MPI	2510	1
	async PM2	563	4.46
	async MPI/Mad	565	4.44
	async OmniOrb 4	595	4.22
Ethernet and ADSL	sync MPI	3042	1
	async PM2	612	4.97
	async MPI/Mad	605	5.03
	async OmniOrb 4	664	4.58

At a more precise level, the PM2 library seems to have the steadiest one since it is always placed either second or first in the experiments. A more surprising result is that the places of MPI/Mad and OmniOrb 4 are inverted depending on the testing problem. In the sparse linear system, MPI/Mad is about 32 percent slower than OmniOrb 4 whereas in the non-linear problem, it is the OmniOrb 4 version which is between 5 and 10 percent slower than the MPI/Mad one.

Those different performances are quite difficult to be precisely explained. In fact, some of the mechanisms used, such as threads or communications management, are directly dependent on the environments. Unfortunately, it is not always possible to have clear explanations on how these mechanisms are actually implemented. Moreover, our algorithms are not exactly implemented in the same way for all the environments because some of the functionalities needed in our algorithms (threads, communications) are not usable in the same way. The main differences between the tested implementations are summarized in Table 4.

Table 4. Differences between the implementations (N is the number of processors)

Sparse linear problem	
PM2	one sending thread
	receiving threads created on demand
MPI/Mad	one sending thread
	one receiving thread
OmniOrb 4	N sending threads
	receiving threads created on demand
Non-linear problem	
PM2	two sending threads
	one receiving thread
MPI/Mad	two sending threads
	two receiving threads
OmniOrb 4	two sending threads
	receiving threads created on demand

In fact, we firstly tried to use exactly the same scheme for all the environments which consisted in using as many threads as necessary for the sendings and the receptions. Nevertheless, we have been confronted with some thread management problems in the PM2 and MPI/Mad environments. Thus, after different testings, the most efficient versions among the working ones have been chosen for each environment.

These differences may have implications for the final performances since the convergence speed of the algorithms is directly related to the sequence of data updates. So, at a given time, if a data sending or a data reception is delayed (due to the programming environment), then the entire iterative process may be slowed down. On another hand, using one thread per communication may produce overhead when the number of processors is very large.

All those remarks may seem to lead to the conclusion that the previous results are not actually comparable. Fortunately, it is not the case since our goal is not to compare the algorithms but to compare the suitability of the environments for implementing AIAC algorithms. Thus, in order to make a well-founded comparison for a given AIAC algorithm, we must use the most efficient version of this algorithm that can be implemented in each programming environment. Hence, the differences between the implementations must only be due to the differences between the programming environments. This is the case for our tested programs.

There is a last remark concerning the execution times between the Ethernet and Ethernet-ADSL contexts for the non-linear problem in Table 3. The times are not directly comparable since the sets of machines used are not the same. The slowest machine in the first set is a bit slower than the one in the second set. This is mainly why we have included the speed ratios, which are objective measures of the efficiency.

Concerning the ratios, it can be seen an important difference between the ratios of the linear problem and those of the non-linear problem. This difference mainly comes from the methods used to solve these two kinds of problems and the impact of communication delays over the global process. In the linear case, when no new data dependency is received on a processor between two consecutive iterations, the computations performed in the second iteration are the same as in the first one. Hence, the computations do not evolve between data receptions and communication delays have then an important impact over the global evolution of the process. On the opposite, in the non-linear case, even if no new data dependency is received on a processor between two iterations, the computations performed in the latter are not the same as the ones in the former due to the Newton method. Thus, the process actually continues to evolve between data receptions and is then less penalized by communication delays.

Now, if we focus on the ratios obtained in the non-linear case, those of the ADSL version are slightly better. This comes from the fact that the impact of the slower ADSL links is much more important over the synchronous version than over the asynchronous ones. As already shown in previous studies, this is due to the implicit overlapping of computations over communications performed in the asynchronous versions.

The last experiment concerns the local heterogeneous cluster. It consists in comparing the evolution of the execution times of the different environments for a given problem size (1000×1000) in function of the number of machines.

This allows us to compare the environments in a context of a quite fast network (Ethernet 100 Mb/s). Since there are three kinds of machines in the local cluster, there are merely the same number of machines of each type in order to have comparable results. In the same way, the types of machines are interleaved in the logical organization of the network in order to preserve the scalability feature. The evolution of the execution times for the three environments are presented in Figure 3.

In this figure, it can be seen that the three asynchronous versions have merely the same behavior. Nevertheless, although the curves of MPI/Mad and PM2 almost coincide, the OmniOrb 4 one is a bit higher, revealing a slightly lower efficiency. In fact, this result is

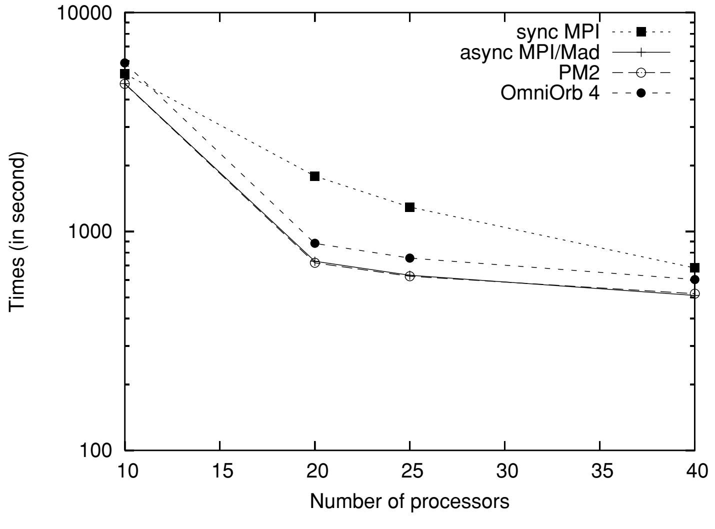

Figure 3. Execution times in function of the number of processors on the local heterogeneous cluster.

not very surprising since the two first environments have been firstly designed to work on local clusters whereas the OmniOrb 4 environment has been designed for distant client/server communications. Thus, their intrinsic communication and thread managers are not made to optimize the same aspects. Hence, the OmniOrb 4 is a bit disadvantaged in this context. Concerning the synchronous MPI version, it clearly appears that it is far less efficient even in such a local context. The convergence of all the curves for the highest number of machine only comes from the fact that the problem size becomes too small to provide enough computations for the forty machines. In other words, the limit of the parallel efficiency is reached. An important deduction which can be made according to this result is that for a given problem size, the best execution times that can be expected in parallel are reached with less processors in asynchronous than with synchronous ones. This is another way to see the advantage of asynchronism over synchronism, it is less resources demanding for the same efficiency.

5.2. Ease of programming

Obviously, the ease of programming is rather subjective to each user. Here, the comparison is based on the major programming aspects intervening in the programming of AIAC algorithms according to our personal experience.

The first remark is that all the tested environments have presented advantages and drawbacks. MPI/Mad is probably the easiest to program since all communications are done in the simple and well known MPI form and the threads are quite easily managed with the Marcel library. PM2 and OmniOrb 4 are quite similar in the way of managing the threads. Concerning the communications, both of them use a remote procedure call mechanism. In PM2, there is a system of explicit data packing before the call to the remote function whereas, in OmniOrb 4, the data to be sent are given as arguments of the called function.

There is a small additional difficulty for the OmniOrb 4 due to the particular use we made of it. Let us recall that this environment is initially made for developing client/server applications. The problem is the initialization phase of the network which consists in establishing the links between the processors. Nevertheless, this step is not so complex and is always the same. So, it can easily be placed in a small library which can be re-used for each development.

5.3. Ease of deployment

In terms of deployment, the advantage clearly goes to OmniOrb 4 due to its high flexibility of use over multiple sites and more generally over the global grid that represents the Internet. A first advantage, required for any ORB, is its wide portability which makes its use transparent on heterogeneous machines, at the programming level. Moreover, its client/server oriented architecture allows the use of a set of processors whose connection graph is not necessary complete. This may be useful to bypass visibility problems due to firewalls. Concerning the run-time environment, a naming service must run over one site of the cluster. Obviously, the environments of each site must be adequately configured to be able to localize and contact this naming service in order to redirect the method invocations to the right processors. Concerning the program launching, one instance must be launched on each processor. This can be realized by different methods (centralized or not) depending on the visibility of the processors the one to the others.

Concerning PM2, its deployment is much more restrictive since it requires a complete interconnection graph of the cluster to be used. Moreover, its portability is less important since it does not completely support the use of several systems and/or several architectures of machine in the same cluster. For example, there is no auto-conversion of data sent to a machine with a different numbers representation and the programmer has then to manage this explicitly. On the other hand, it does not need any particular run-time environment. Its configuration is quite simple since it only requires a file containing the list of the machines in the cluster. The program is launched by executing a specific PM2 command (pm2load) with the name of the program in parameter on one machine of the cluster.

Finally, concerning the MPI/Mad environment, it is quite similar to PM2 in terms of deployment and portability. Indeed, all the machines must be visible the one to the others and data representations must be taken into account by the programmer. On the other hand, an interesting feature due to the Madeleine 3 library is the possibility to use different kinds of communication protocols in the same parallel application. For example, we can use two sites communicating with the TCP-IP protocol whereas the local network of the first site uses Myrinet and the local network of the second site uses SCI. The consequence of this feature appears at the configuration level. There are two files containing a list of protocols and for each one, the list of machines communicating with this protocol. The first file contains the list of all available protocols and machines and the second one contains the list of protocols and machines actually used in the application. The launching of the program is a little bit more complicated than for PM2 but still requires only one command on one of the machines.

6. Features needed for implementation of AIACs

According to the experience acquired from all our implementations of AIACs algorithms, we have established a list of all the features required in a programming environment to allow an efficient implementation of AIACs.

There are mainly two elements. The former one is a classical communication system providing blocking point to point communications. Nevertheless, important features for a flexible deployment over the grid are the ability to use different communication protocols and the possibility to use incomplete connection graphs. Most of the communication libraries do not provide these elements.

The latter one is a multi-threading system with a fair thread scheduler. This last feature is important when managing several sending/receiving threads towards/from several destinations/sources. Indeed, if the scheduler is not fair, it is possible to have always the same threads working and the same other ones which are never activated. Thus, the communications managed by the latter are not performed. Moreover, when using a RPC-based communication system, it is important that the receptions could be done in separate threads activated on demand.

Finally, in addition to these features, a mutex system (often provided in the threads library) is very useful to efficiently manage the asynchronous communications between the processors and the data updates, especially when the algorithms use load balancing.

7. Conclusion

A comparison of three parallel programming environments according to the implementation of AIAC algorithms has been presented. The environments have been tested with two AIAC algorithms solving two representative problems, a sparse linear one and a non-linear one. For each environment, the most efficient implementation of each algorithm has been tested.

It has been pointed out that the main differences between those environments comes from the communication system and particularly from the way the threads are managed. Indeed, it has been shown that the implementation of a given algorithm cannot be exactly the same from an environment to another because of the different threads managing systems. Since the threads are used to perform the asynchronous communications, this may have a direct impact over the overall performances.

The global result of this comparison is that the three tested environments, PM2, MPI/Madeleine and OmniOrb 4 obtain merely the same performances. Moreover, this result will be strengthen when the number of data increases. Effectively, with coarse grained configurations, the ratio of communications upon computations will decrease, implying a decrease in the influence of the communications on the overall performances. Thus, the difference between the performances of the implementations of the same AIAC algorithm with the three tested environments will also tend to decrease.

Nevertheless, a distinction can be made between those environments according to the regularity of the obtained performances. Effectively, from our experiments, it can be seen that the PM2 environment has the steadiest behavior in terms of performances.

To complete the comparison, two other aspects have been discussed, the programming and deploying tasks. It appears that the MPI/Mad environment is a bit easier to program whereas it is the OmniOrb 4 environment which is the most adapted to deployment.

According to our experience acquired when implementing AIAC algorithms with all these environments, a list of features to be included in a programming environment designed to efficiently implement AIACs has also been given.

Beyond the comparison of programming environments, this study also shows that AIAC algorithms are sufficiently flexible to be easily and efficiently implemented with several general purpose parallel programming environments. This is a rather important feature since it allows everyone to use the most familiar environment more efficiently.

Finally, the experiments have confirmed that AIAC algorithms are far more efficient than synchronous algorithms. Moreover, it has been pointed out that these good performances have been obtained with environments which are not directly designed for implementing this kind of algorithms. So, despite the fact that AIAC algorithms are very flexible and do not require heavy features in the programming environments, even better results can be reasonably expected with a more complete environment efficiently providing all these features.

References

1. O. Aumage, G. Mercier, and R. Namyst. MPICH/Madeleine: a True Multi-Protocol MPI for High-Performance Networks. In *Proc. 15th International Parallel and Distributed Processing Symposium (IPDPS 2001)*, page 51, San Francisco, Apr. 2001. IEEE.
2. J. Bahi, S. Domas, and K. Mazouzi. Jace : A java environment for distributed asynchronous iterative computations. In *12-th Euromicro Conference on Parallel, Distributed and Network based Processing, PDP'04*, pages 350–357, Coruna, Spain, February 2004. IEEE computer society press.
3. J. M. Bahi. Asynchronous iterative algorithms for nonexpansive linear systems. *Journal of Parallel and Distributed Computing*, 60(1):92–112, 2000.
4. J. M. Bahi, S. Contassot-Vivier, and R. Couturier. Evaluation of the asynchronous model in iterative algorithms for global computing, 31(5):439–461, 2005.
5. J. M. Bahi, S. Contassot-Vivier, and R. Couturier. Asynchronism for iterative algorithms in a global computing environment. In *The 16th Annual International Symposium on High Performance Computing Systems and Applications (HPCS'2002)*, pp. 90–97, Moncton, Canada, June 2002.
6. J. M. Bahi, S. Contassot-Vivier, and R. Couturier. Coupling dynamic load balancing with asynchronism in iterative algorithms on the computational grid. In *International Parallel and Distributed Processing Symposium (IPDPS'2003)*, Nice, France, April 2003. Proceedings on CD-Rom.
7. J. M. Bahi, J.-C. Miellou, and K. Rhofir. Asynchronous multisplitting methods for nonlinear fixed point problems. *Numerical Algorithms*, 15(3/4):315–345, 1997.
8. D. P. Bertsekas and J. N. Tsitsiklis. *Parallel and Distributed Computation: Numerical Methods*. Prentice Hall, Englewood Cliffs NJ, 1989.
9. A. Frommer and D. Szyld. On asynchronous iterations. *J. Comp. Appl. Math.*, 123:201–216, 2000.
10. A. Frommer and D. B. Szyld. Asynchronous iterations with flexible communication for linear systems. *Calculateurs Parallèles, Réseaux et Systèmes répartis*, 10:421–429, 1998.
11. W. Gropp, E. Lusk, and A. Skjellum. *Using MPI : portable parallel programming with the message passing interface*. MIT Press, 1994.
12. A. Herz and C. Marcus. Distributed dynamics in neural networks. *Physical Review E*, 47(3):2155–2161, 1993.
13. R. Namyst and J.-F. Méhaut. PM^2: Parallel multithreaded machine. A computing environment for distributed architectures. In *Parallel Computing: State-of-the-Art and Perspectives, ParCo'95*, volume 11, pp. 279–285. Elsevier, North-Holland, 1996.
14. Omniorb web page. http://omniorb.sourceforge.net.
15. A. Pope. *The CORBA Reference Guide: Understanding the Common Object Request Broker Architecture*. Addison-Wesley, Reading, MA, USA, Dec. 1997.
16. Y. Saad. Iterative methods for sparse linear systems, second edition. *SIAM*, 2003.
17. M. E. Tarazi. Some convergence results for asynchronous algorithms. *Numer. Math.*, 39:325–340, 1982.
18. R. S. Varga. Matrix iterative analysis. *Prentice-Hall*, 1962.