Interest of the asynchronism in parallel iterative algorithms on meta-clusters

Jacques M. Bahi, Sylvain Contassot-Vivier and Raphaël Couturier

Laboratoire d'Informatique de l'Université de Franche-Comté (LIFC),
IUT de Belfort-Montbéliard, BP 527, 90016 Belfort, France

Abstract

The subject of this paper is to show the high efficiency of asynchronism for parallel iterative algorithms in the context of meta-clusters, that is to say, with machines scattered on a broad geographical scale. The question is: does asynchronism help to reduce the communication penalty and the overall computation time of such a given algorithm ? The asynchronous programming model is evaluated on two test problems representing two important classes of scientific applications: a stationary linear problem and a non-stationary non-linear problem. They are implemented with a multi-threaded environment and tested on a set of distant heterogeneous machines. Several experiments have been performed allowing us to compare the performances of such asynchronous algorithms and also to analyze their behavior and extract the main possible optimizations for their use in a grid computing context.

Keywords: parallel iterative algorithms, meta-clusters

1 Introduction

In the context of scientific computations, iterative algorithms are very well suited for a large class of problems. The classical synchronous versions of parallel iterative algorithms are often substantially penalized by communications, especially in the grid computing context. Thus, introducing asynchronism in such algorithms may be an attractive solution to bypass this penalty. Starting from theoretical results about the convergence of asynchronous algorithms, we show the adaptability of the asynchronous model to grid computing.

There are two kinds of algorithms to solve numerical problems, direct and iterative ones. Direct algorithms give the exact solution after a finite number of operations; iterative algorithms give an approximation of the solution after a given number of iterations, we can say that they converge to the solution. Moreover, iterative algorithms may require smaller storage than direct ones and some non-linear systems can only be solved by iterative algorithms like, for example, the polynomial roots problem.

Iterative algorithms have the following structure:

$$x^{k+1} = f(x^k), \qquad k = 0, 1, ... \tag{1}$$

where each x^k is an n - dimensional vector, and f is some function from $\mathbb{R}^n$ into itself. Notice that if the sequence $\{x^k\}$ generated by the above iteration converges to some x^* and if f is continuous then we have $x^* = f(x^*)$, where x^* is called a fixed point of f.
Let x^k be partitioned into m block-components X_i^k

$$x^k = (x_1^k, x_2^k, ..., x_n^k) \equiv X^k = (X_1^k, X_2^k, ..., X_m^k)$$

and f be partitioned in a compatible way into m block-components F_i, then equation (1) can be reformulated as

$$X^{k+1} = F(X^k), \qquad k = 0, 1, ... \tag{2}$$

which gives in a block-components formulation

$$(X_1^{k+1}, X_2^{k+1}, ..., X_m^{k+1}) = (F_1(X^k), F_2(X^k), ..., F_m(X^k)), \qquad k = 0, 1, ... \tag{3}$$

where

$$X_i^{k+1} = F_i(X^k) = F_i\left(X_1^k, ..., X_m^k\right) \qquad i = 1, ..., m \tag{4}$$

So, the iterative algorithm can be parallelized by letting each of the m processors update a different block-component X_i according to (4). At each stage, the i^{th} processor knows the value of all the components of x^k on which F_i depends. It computes the new values X_i^{k+1} of its local data and communicates the part of them which are required on other processors for their own local computations.

In the context of a message-passing system, a distributed algorithm is considered as a collection of local algorithms. Each local algorithm is executed on a different processor and occasionally uses data generated by other local algorithms. Different cases of parallel iterative algorithms can be distinguished. A complete classification is given in [1]. Here, we briefly describe the two main cases. The first one is the synchronous case in which the local algorithms wait for needed data to become available from the other processors. This case is included in the BSP programming model [2]. It implies idle times on the processors either when they wait for another processor to be ready to communicate or during the communication itself. The second case is the asynchronous one in which local algorithms do not wait for the arrival of new updates of the non-local data required for their computations but keep on computing, using the latest available versions of these data. The direct consequence is that there are no more idle times and processors may be computing different iterations at a given time as can be seen in Figure 1. Thus, we can allow some processors to compute faster and execute more iterations than others. Moreover, the communication delays may be substantial and unpredictable. This situation corresponds typically to a large network of heterogeneous processors.

Thus, the communication penalty can often be substantially reduced by means of an asynchronous implementation. In this paper, we discuss the interest of asynchronism for iterative algorithms in the context of grid computing with message-passing. Several studies have been made of the convergence of asynchronous algorithms [3, 4, 5], their possible variants [6] and their applications [7, 8]. Nevertheless, very few experiments are reported and there is no study in a grid computing context. This paper contributes to fill this gap by proposing an experimental study of asynchronous algorithms in such a context. In this way, we perform a comparison of performances between synchronous and asynchronous versions of two iterative algorithms. These algorithms solve two representative problems: on the one hand, a classical stationary linear problem with the gradient method, and on the other hand, a non-stationary non-linear problem with a well-known relaxation algorithm called the WR algorithm. The implementation of our algorithms is performed with the PM2 (Parallel Multi-threaded Machine) environment which is very convenient for implementing asynchronous communications.

The following section briefly recalls the model of asynchronous iterative algorithms. Then, an overview of previous works related to parallel iterative algorithms is given in Section 3. The general algorithmic scheme for implementing asynchronous iterative algorithms is detailed in Section 4. Then, the two test problems and their corresponding asynchronous algorithms are described in Section 5. Finally, experimental results are given and interpreted in Section 6.

2 Algorithmic model and communication schemes of asynchronous algorithms

Let's recall that given a problem to be solved, an iterative algorithm consists in finding a fixed-point formulation

$$x^* = f(x^*) \tag{5}$$

so that algorithm (1) converges to x^*, the solution of the considered problem.

Consider the block-decomposition $X_i,\ i = 1, ..., m$ described in the introduction. At each block X_i corresponds an associated set T_i containing the series of times at which X_i is updated during the iterative process.

We suppose that each processor in the system may not have access to the most recent values of the other processors. So, in asynchronous implementation, algorithm (4) becomes

$$X_i^{k+1} = F_i\left(X_1^{s_1^i(k)}, ..., X_m^{s_m^i(k)}\right) \qquad \forall k \in T_i \tag{6}$$

where $s_j^i(k) = k - r_j^i(k)$ ($r_j^i(k)$ is the delay of node j according to node i) represents the version of data coming from node j and available on node i at time (iteration) k. The $s_j^i(k)$ satisfy

$$0 \leq s_j^i(k) \leq k, \ \forall k \in I\!N \text{ and } j = 1, ..., m$$

If $k \notin T_i$, then the value of the i^{th} block-component is unchanged

$$X_i^{k+1} = X_i^k \qquad \forall k \notin T_i \tag{7}$$

To ensure the convergence of such algorithms, two conditions, standard in the literature related to asynchronism [3, 9, 7, 6], have to be verified. The first one is that each component of the system must be updated infinitely often (sets T_i are infinite). The second one is that although the delays may not be bounded, arrival times of new versions of non-local data must follow the iterations ($\lim_{k\to\infty} s_j^i(k) = \infty$, $\forall i$ and $j \in \{1, ..., m\}$).

This model of asynchronous iterations can be split into two main variants depending on the way the communications are performed:

- rigid communications: they are performed between two iterations. On a given node, sending operations are done at the end of each iteration and the receptions which may happen at any time (also during an iteration) are taken into account only after the current iteration.

- flexible communications: communications can occur at any time during an iteration. Data which have been updated at a given time $t_s \in [t, t+1]$ are directly sent without waiting for the end of the iteration. In the same way, as soon as some data are received, they are taken into account in the current iteration.

Rigid communications are commonly used in the classical form of asynchronous iterations and a typical example of its execution flow is shown in Figure 1. Flexible communications [10, 11] are a generalization of rigid communications and have been designed to enhance the reactivity of the process and thus to speedup its convergence. Its execution flow is exhibited in Figure 2 where the dashed lines denote the communications performed during the iterations. As can be seen in this figure, this scheme sharply increases the number of communications. Hence, in most cases, the sending operations are not performed separately for each datum during an iteration but for a group of data.

A third possibility, between these two main variants, is a hybrid version which could be described as semi-flexible communications. To reduce the amount of communications, all the sending operations are done between iterations whereas the receptions are processed and taken into account as soon as they arrive on a processor. The execution flow of this scheme is depicted in Figure 3.

3 Related works

Most of parallel iterative techniques used to solve scientific problems (linear or not) have extensively been used in a context of local computing, either on supercomputers or on local clusters, where interconnection networks are quite fast and reliable. In such a case, communications between nodes are managed synchronously which implies the same behavior as the sequential algorithm. Hence, the convergence of the parallel algorithm is directly deductible from its sequential version. To our knowledge, it has never been used in a grid computing context.

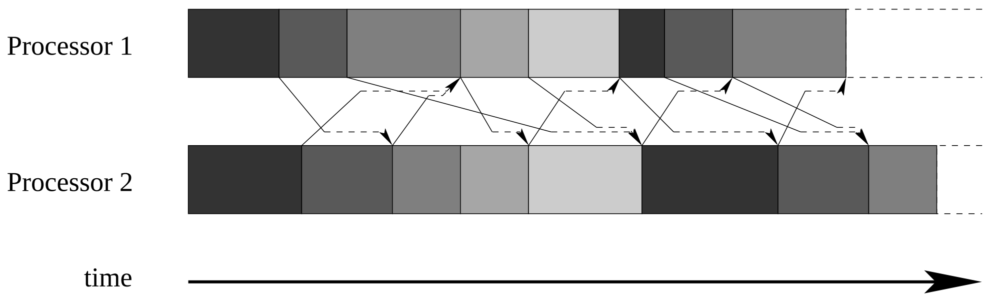

Figure 1: Execution flow of an asynchronous iterative algorithm with rigid communications.

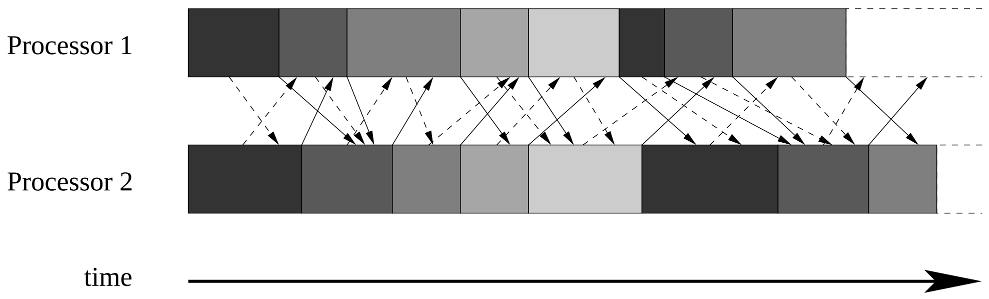

Figure 2: Execution flow of an asynchronous iterative algorithm with flexible communications.

In our study, we show that it is possible to use such computational techniques in this particular context. This purpose introduces new constraints on the communications between nodes like important delays. So, using synchronous communications becomes an important drawback since the overall computation time will be slowed down by the delays. Thus, we propose to use asynchronous algorithms. Nevertheless, since the global behavior of an asynchronous iterative algorithm is different from the one of its synchronous/sequential counterpart, its convergence cannot be directly deduced from the sequential case.

4 Asynchronous parallel iterative algorithm

In this section, we consider the use of a Network of Workstations composed of $NbProcs$ machines (processors, nodes...) numbered from 0 to $NbProcs - 1$. Each processor can send and receive data from any other one.

Since the major problems in such algorithms are the detection of the global convergence and the halting procedure, they require detailed discussions.
Hence, to be clearer, we have chosen to firstly present a simplified algorithm in which only the main iterative process and its corresponding communications are detailed. Then, the convergence detection and the halting procedure are respectively detailed in Subsections 4.2 and 4.3.

4.1 General algorithmic scheme

The general scheme of the asynchronous iterative process is given in Algorithm 1. Parallelism is classically exploited by homogeneously distributing the data over the processors. In general, the logical organization of the processors tend to follow as much as possible the topology of the problem (the data dependencies). This is done to minimize the communications in the physical interconnection network and to avoid as much as possible bottlenecks and/or collisions. Indeed, it is useless to have links between processors which do not

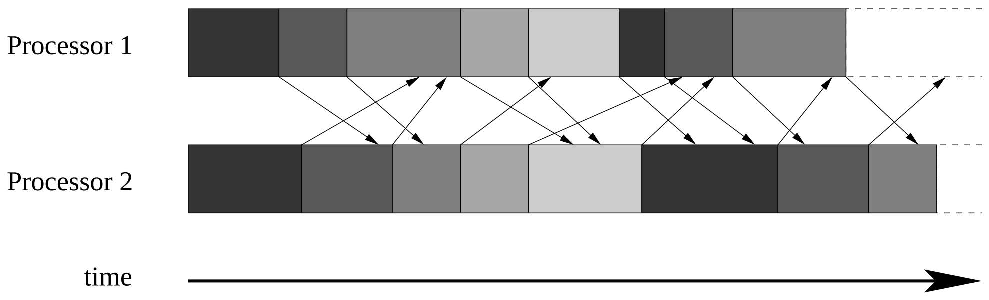

Figure 3: Execution flow of an asynchronous iterative algorithm with semi-flexible communications.

Algorithm 1 parallel asynchronous algorithm

```
Initialize the communication interface
OldLocalData = *Array of local data owned by the processor*
NewLocalData = *Array of new values of local data*
Initialization of local data in OldLocalData
**repeat**
  Compute new values of local data in NewLocalData using OldLocalData
  Asynchronously send the required local data to processors needing them
  Copy NewLocalData in OldLocalData
  Convergence detection
**until** Global convergence is detected
Display or save local components
```

exchange any information. Moreover, due to deployment problems and access constraints between machines on different sites in grid computing contexts, it is often important to minimize the number of physical links in the parallel system. Hence, for a one-dimensional problem, a simple and efficient solution is to use a linear organization of the processors. For a N-dimensional problem, processors should be organized in a way that minimizes the number of physical links.

The main scheme of the algorithm is as follows: each processor updates the values of its local data using the computational method related to the problem to solve. These updates are performed using the last values of local data and the available values of the needed data coming from other processors. Then, updated data are asynchronously sent to the processors which need them. Any of the three communication schemes presented in Section 2 can be used to perform the data exchanges. This process is repeated until the global convergence of the system is detected.

In order to facilitate and enhance the implementation of asynchronous communications, our algorithms are presented for use in a multi-threaded programming environment. This kind of environment makes it possible to carry out the sending and receiving operations in additional threads rather than in the main program. This is why the receipts of data do not directly appear in the main algorithm. In fact, they are localized in a function (Algorithm 2) called by a thread created at the beginning of the program and dealing with incoming messages. In the same way, the asynchronous sending operations appearing in our algorithms actually correspond to the activation of a communication thread calling the related sending function.

In order to avoid overloadings of the network and incoherent situations due to simultaneous uses of the

Algorithm 2 function RecvData()

```
Receive data
Put them at their corresponding position in array NewLocalData
```

same communication function (send or receive) on the same node, a system of mutex in conjunction with blocking communications has been used. It is important to notice that since we use threads to perform communication operations, their blocking feature does not synchronize the computations in our algorithm. So, when a given communication function begins (send or receive), it locks a corresponding mutex. While the mutex is locked, no other call to the function can be performed. For sending operations, the mutex are explicitly managed in the main program so that no sending function is called when it is already in use. For receiving operations, since we perform blocking sendings, the mutex will in fact only be useful to ensure there are no more receptions in progress in the halting procedure (see Section 4.3). It is not a problem to receive data during the computations, the only difficulty is then to ensure that each data writing in the local array is atomic to avoid coherency problems. Finally, at the end of the communication function, the mutex is unlocked, allowing the function to be called again. Since the use of mutex is quite simple and is inherent to the communication system, we consider they are part of the sending and receiving operations. So, to make it clear and simple, they do not appear in our algorithm.

In terms of execution flow, this version is slightly different from the communication schemes described in Section 2 since the communications are exclusive. By exclusive, we mean that on a given node, there cannot be two communication operations of the same nature (for example, two sendings of the same kind of data) at the same time. As an example, the execution flow of the exclusive semi-flexible scheme is given in Figure 4. It is merely the same as in Figure 3 except that the light gray lines correspond to communications which are no more performed. It can be seen that some sending operations between iterations are not performed as long as the previous sending is not achieved.

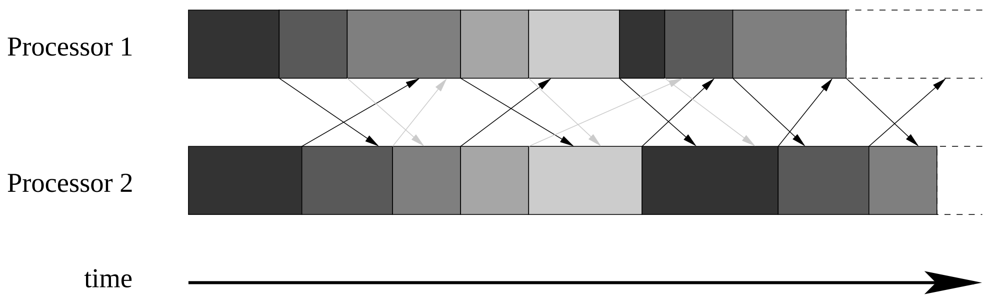

Figure 4: Execution flow of an asynchronous iterative algorithm with exclusive semi-flexible communications.

4.2 Convergence detection

Since the convergence detection in asynchronous algorithms is a difficult problem in itself, a small overview of this domain is given before our own algorithm.

4.2.1 Overview

There are two main kinds of convergence detection. The ones which substantially modify the asynchronous algorithms and the ones which do not modify them. A good example of the first category is the method of Bertsekas and Tsitsiklis [3, 12]. The induced modifications make this first category only applicable to a subset of the asynchronous iterations since the convergence is no more directly deductible from standard results. A good example of the second category is the method of Savari and Bertsekas [13]. It seems clear that these solutions are widely applicable to asynchronous algorithms. Nevertheless, they often imply a large number of additional communications. Some studies have been done to limit this number of communications, see for example [10].

Here, we define a convergence detection algorithm which enters in the second category and is thus applicable to all asynchronous algorithms. Moreover, our algorithm tends to minimize the communications and uses a robust local convergence criterion.

4.2.2 Convergence detection algorithm

To deal with convergence, the first thing to do is to define what is the convergence and how it is achieved. In the case of iterative algorithms, the convergence is commonly associated with the residual which represents the variation between the computed values of two successive iterations. It is of the form:

$$\begin{aligned} \|x^{k+1} - x^k\|_\infty &= \max_i |x_i^{k+1} - x_i^k| \\ &= \max_i |f_i(x_i^k) - x_i^k| \end{aligned}$$

As long as the error is greater than a given threshold, the system is seen as unstable and when this error becomes lower, the convergence is assumed to be reached.

The second problem of convergence detection is directly linked to the knowledge of the global state of the system since global convergence is achieved only when each node is in stable state. In order to obtain this global knowledge, we have chosen to gather the states of the nodes on one of them. This particular node will be referred to as the master node in the following. For simplicity, node 0 can be chosen to be the master node without loss of generality. In fact, this node behaves exactly like the others except that it has a small additional charge which consists in receiving the states of all other nodes and detecting the global convergence.

Since those receptions are made asynchronously and the convergence detection is not time-consuming according to the iterative computations, it can be assumed that the overhead implied by the global state concentration on the master node is negligible. Nevertheless, it could be interesting to avoid this concentration of communications towards one of the nodes. This could be avoided by using a local gathering algorithm only relying on communications between neighbors. However, this would tend to increase the number of messages in the system. The best solution would be to avoid the use of such a master node. This is an entire problem in itself which is the object of another work.

The centralized method implies that the other nodes send their local state to node 0. A first solution would be to send the state after each iteration but this is quite expensive in terms of communications. Hence, a better solution is to send the local state every time this state changes. Nevertheless, when the node is approaching a stable state, its local error may oscillate around the threshold and then the local state may also oscillate, generating a lot of useless communications. To avoid this, a small system of local stability has been designed which relies on the duration of the convergence. It consists in counting the number of consecutive iterations made while the local error is under the threshold. So, local convergence is only assumed once this number has reached a given value, then the state is sent to the master node. However, as soon as the error comes back upon the threshold, the convergence is canceled and another message is sent to the master node to notify the modification.

Finally, the last step takes place on the master node and consists in detecting the global convergence. This is done by managing an array containing the states of all nodes which is affected each time a convergence message is received. The global convergence is detected when all the cells of the array are in stable state.

The convergence detection algorithm and the additional state reception function (only used on the master node) are given in [14].

Once the global convergence is reached, the final task is to stop the iterations on all the nodes. This is detailed in the following paragraph.

4.3 Halting procedure

As seen in the previous paragraph, the global convergence is detected only on the master node. Thus, to stop the computations on all the processors, a halting message is sent to all other nodes from node 0. Such a broadcast could be quite heavy to implement in a synchronous model but it is quite straightforward in our case since it is performed asynchronously with threads.

Since all the communications must be completed before the system can exit the communication interface, each processor must ensure there is no more sending request starting from it. Hence, once a processor has received the halting message, it stops its computational loop and wait for the completion of its communications in progress. This is achieved by testing if all the mutex related to sending and receiving operations are unlocked. Nonetheless, this test is local and only ensures there is no more communication on this node but not in the entire system. Hence, a barrier is performed to synchronize all the nodes. After this barrier, it is then sure there is no more communication on every node. Then, the master can stop the communication interface and every node can save or display its results and exit the program. This halting procedure is given in [14].

The major cost in this final part lies in the barrier needed before stopping the communication interface. Nevertheless, all our experiments show that this point does not significantly degrade the overall performances of our algorithm.

Finally, we obtain a complete general scheme for the implementation of asynchronous parallel iterative algorithms which can solve a large class of scientific problems.

According to each particular problem, some adaptations of this general programming scheme are often required. In the following section, the two test problems are presented together with their asynchronous iterative algorithms.

5 Test problems

Two test problems have been considered: the first one is a stationary sparse linear system and the second one is a one-dimensional system of non-stationary Partial Differential Equations (PDEs).

5.1 The linear case

The problem is of the form

$$Ax = b \tag{8}$$

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

$$x^{k+1} = x^k + \gamma M^{-1}(b - Ax^k) \tag{9}$$

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 (8) is achieved by the convergence of the iterative process in equation (9). This convergence is obtained at iteration k according to the norm used (the *max* norm in our case) and a given $\epsilon \in \mathbb{R}^+$ when

$$\max_i |x_i^k - x_i^{k-1}| < \epsilon \tag{10}$$

5.2 Adaptation of the general programming scheme to the linear problem

Concerning this problem, the matrix and vectors are decomposed in horizontal strips and distributed to 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 its local data.

On each processor, the iterative process consists in computing the successive approximations of the local part of vector x. The last available values of the other parts of vector x are used. Once the new values of local data are computed, those which are required on other processors are asynchronously sent according to the list of data dependencies. Thus, there may be several sendings at each iteration and a processor may also receive data messages from several sources.

In order to avoid problems, data are actually sent only if any previous sending of the same data to the same destination is terminated. Otherwise, this sending is not performed at this iteration but is delayed till the next iteration. The receptions of data dependencies are managed in separated threads allowing them to be performed 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.

Finally, a limit is set on the number of iterations to avoid asymptotic convergences when the required accuracy is too high. Except this, the convergence detection and halting procedure are not modified.

5.3 The non-linear case

We have chosen a large stiff system of PDEs called the Medical Akzo Nobel (MAN) problem [15]. This is a stiff problem and as has been pointed out by Burrage [16], it requires the use of implicit methods. Thus, large systems of non-linear systems have to be solved at each iteration. Parallelism is then natural for that kind of problems.

This problem models the penetration of radio-labeled antibodies into a tissue infected by a tumor in a one dimensional reaction. The wanted results are the evolutions of the concentrations u and v of both elements (the antibodies and the tissue) along the discretized space in function of time. By using the method of lines (MOL), this problem can be reformulated as an Ordinary Differential Equation (ODE) of the form:

$$\frac{dy}{dt} = f(y,t)\ ,\quad y(0) = g \tag{11}$$

with

$$y \in I\!R^{2N}\ ,\quad 0 \le t \le 20,\ and\ \ g = (0, v_0, ..., 0, v_0) \in I\!R^{2N}$$

In the classical formulation of the problem, the vector of data is noted as y. In our context, it corresponds to the vector x of equation (1). The $y_{2j-1}, j \in$ [1,N] represent the concentrations u along the spatial dimension and the $y_{2j}, j \in$ [1,N] represent the concentrations v. N is a parameter of the problem. For further details about this problem and its formulation, the reader should refer to [15].

The important fact according to our study is that the MAN problem is a particular case of explicit Initial Value Problems (IVPs). Considerable attention has been given to solve IVP problems on parallel and distributed systems. In [17], Gear proposed two different categories for developing parallel techniques for IVPs based either on parallelism across the method or parallelism across the system. Algorithms of the first category include those which exploit concurrent function evaluations within a step such as multi-stage direct algorithms. The second category includes the waveform relaxation (WR) approach introduced by Lelarasmee [18]. WR algorithms are characterized by the fact that the state variables y_l are no longer computed as constants but as functions $y_l(t)$ within the time interval. These functions, called waveforms, are calculated according to their initial values and the waveforms of other state variables.

This is the WR algorithm which has been chosen to solve the MAN problem. Concerning its convergence in the asynchronous mode, the results given in [19, 20, 21] ensure us that our version will also converge.

It is not our objective to study the convergence of asynchronous WR algorithms, such studies have been done in the larger framework of IVPs, namely the differential-algebraic equations framework [22] and also in the particular case of ordinary differential equations [19]. For the application considered in this paper, the initial system of PDEs is converted into a system of ODEs by using the finite differences method to discretize the spatial derivative. Then, we obtain an ODE so that the function f is Lipschitz continuous with respect to y, which means that given a maximum norm $\|u\|_\infty$, there exists a constant L such as:

$$\|f(u,t) - f(v,t)\|_\infty \le L\,\|u - v\|_\infty$$

This last inequality ensures the uniform convergence of asynchronous WR algorithm in any finite time interval $[t_0, t_1]$.

For extensive study of the convergence of WR algorithms and the improvement of their speed, see [16] for the synchronous case and [19, 20] for the asynchronous case.

5.4 Adaptation of the general programming scheme to the non-linear problem

To solve equation (11), we use a two-stage iterative algorithm:

- At each iteration:
  - we use the implicit Euler algorithm to approximate $\frac{dy}{dt}$,
  - we use the Newton algorithm to solve the resulting non-linear system.

It is interesting to note that both types of WR algorithms, Jacobi or Gauss-Seidel, are usable in this context.

In order to exploit the parallelism, the y_i functions are distributed homogeneously according to the number of processors. Since these functions are in a one-dimensional space, the processors are logically organized in a linear way for the spatial components (y_i functions) to be mapped onto them. In addition, in order to be able to compute the evolution of spatial components in time, we have used a discretization of time with a precision δt given by the user.

Then, each processor applies the Newton method to its local components using the needed data from other processors involved in its computations. The residual used to detect the convergence is the maximal norm between old and new values of the local y_i functions. From the formulation of the Akzo Nobel problem, it appears that the processing of data y_p to y_q also depends on the two spatial components before y_p and the two spatial components after y_q. Hence, if we consider that each processor has at least two functions y_i, the non-local data needed by each processor to perform their iterations only come from the previous and following processors in the linear logical organization. In practical cases, the number of processors will be much smaller than the number of spatial components and each processor will then own more than two functions. However, it would not be difficult to extend our algorithm to the particular case of one data per node.

So, at each iteration, each processor asynchronously sends its first two local data to its left neighbor and its last two local data to its right neighbor. By symmetry, each processor has to receive dependency data from its left and right neighbors. Hence, data receptions are managed by two functions (one for each neighbor) executed by separate threads. Thus, when a sending operation is performed on a given processor, the function which will handle the message on the destination node must be specified.

Concerning the convergence detection and halting procedure, they are not modified.

6 Experiments

In this section, our asynchronous algorithms are respectively compared to their synchronous versions in order to evaluate the gain obtained with asynchronism. According to the scope of this article, we will focus on the grid computing context which consists of a set of heterogeneous distant machines.

In order to perform those experiments, we have used the PM2 environment [23]. Its first goal is to efficiently support irregular parallel applications on distributed architectures. It is designed to manage numerous threads on each processors and thread migration mechanisms, allowing the management of high degree of parallelism and dynamic load-balancing. Nevertheless, there is no known use of PM2 neither for asynchronous iterative algorithms nor in a grid computing context. Our study also tends to show the very good convenience of this kind of environment to implement asynchronous algorithms.

It is worth noticing that although the multi-threaded method is very well suited to the implementation of this kind of algorithms, it is not the unique solution and classical message-passing interfaces such as MPI [24] or PVM [25] could also be used. Nonetheless, another advantage of the PM2 environment is that it avoids as much as possible the use of intermediate buffers during the communications and thus does not make useless data copies. Moreover, with PM2, data can be updated as soon as they arrive on the node whereas with MPI or PVM, this can only be done after an explicit call to a receive function which then implies a delay in the data updates. For these reasons and also for the ease of programming such algorithms with PM2, we have used this environment rather than MPI or PVM.

Concerning the quality of the results obtained by our asynchronous algorithms, it is equivalent to those obtained by the synchronous versions. This is not surprising since the same convergence criterion is used in both versions. So, the iterations stop once the desired accuracy is reached. This has been confirmed in all our experiments.

Finally, in the context of our experiments, some of the machines were subject to a multi-users utilization directly influencing their load. Thus, although a special care has been taken to perform our tests when the machines were not too loaded, in order to obtain representative performances, each time given in the following results is the average of a series of executions.

The two test problems are presented in the following subsections.

6.1 Stationary linear case

Three matrix sizes ($1000000^2, 2000000^2$ and 3000000^2 elements) have been chosen to perform the tests. In each matrix, the non-zero values are localized on 30 sub-diagonals.

This problem has been tested on a heterogeneous cluster of ten machines scattered over three distinct sites with 20Mb/s Ethernet links between sites and 100Mb/s Ethernet links between machines inside each site. The repartition of the machines is as follows: one P4 1.6Ghz, one Athlon 2.1Ghz, two P4 2.4Ghz and one P4 2.6Ghz on the first site; one P3 833Mhz on the second site; four Athlon 1.7Ghz on the third site. The number of machines used is quite small due to the difficulty to find available machines without access constraints between the different sites. However, this experiment (and the following one) keeps all its comparative interest since it allows to point out the fundamental differences between the behaviors of the two versions of the algorithm. Moreover, this configuration represents a quite common test environment for most of the researchers who work on grid computing.

Finally, concerning this particular application, it would not mean anything to make this kind of computations on slower 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 transfered data is important and the communication scheme is all to all according to data dependencies.

matrix size	synchronous (MPI)	asynchronous (PM2)	ratio
1000000×1000000	300.48	203.91	1.47
2000000×2000000	609.23	398.18	1.53
3000000×3000000	1062.39	622.32	1.71

Table 1: Computational times (in seconds) for the sparse linear problem on a heterogeneous cluster of ten machines scattered on three sites.

The results of our experiment are given in Table 1. The presented times only concern the computational part of the programs and do not take into account the data loadings/savings. This allows us to precisely compare the aspect of the execution which is different in the two tested algorithms (IO parts are the same in both versions). The last column gives the ratio of the synchronous time upon the asynchronous one. A major improvement in computational times can be noticed when using the asynchronous version although this application represents a difficult case of grid computing according to the numerous data dependencies. Moreover, the evolution of the ratio in function of the problem size shows that the asynchronous version is more tolerant to an increase of the communication volumes. So, this first experiment clearly shows the potential power of asynchronism in the context of grid computing.

6.2 Non-stationary non-linear case

In this experiment, we have used a different heterogeneous system with twelve machines scattered over four sites in France interconnected with 10Mb/s Ethernet inter-site links and 100Mb/s Ethernet intra-site links. Their repartition is: two P2 350Mhz and one P2 450Mhz on the first site, four P3 450Mhz on the second site, one P3 833Mhz on the third site and four P3 733Mhz on the last site.

It has been shown in Section 5.4 that data communications related to a given machine are performed with its two neighbors in the logical organization of the cluster. Hence, the order of the machines in this logical organization is quite important. This is why two series of experiments have been done with two different logical orderings of the machines.

The first series uses an ordering by site, i.e. all the machines in the first site followed by those in the second site and so on. The second series uses an interleaved configuration where two neighboring machines are on different sites.

For both configurations the algorithms have been tested with three sizes of the problem : 8000, 18000 and 30000. The results are given in Table 2.

N=8000			
ordering	synchronous	asynchronous	ratio
by site	461.66	124.87	3.69
interleaved	656.55	152.45	4.30

N=18000			
ordering	synchronous	asynchronous	ratio
by site	845.48	257.01	3.29
interleaved	870.71	255.03	3.41

N=30000			
ordering	synchronous	asynchronous	ratio
by site	981.49	418.32	2.34
interleaved	1258.19	414.12	3.03

Table 2: Execution times (in seconds) for the non-linear problem on two configurations of a heterogeneous cluster of twelve machines scattered on four sites.

In this table, several points can be noticed. The first one is that the asynchronous version obtains far better performances than the synchronous one in all the cases. Here, we can see all the efficiency of asynchronism which allows to automatically obtain a very good overlapping of communications by computations in a very natural way. Hence, the benefits can be observed as soon as there is an imbalance in the global system, either due to the communication speeds or to the relative powers of the machines.

The second interesting point is the difference between the two orderings of the machines: by site or interleaved. It can be observed that the performances of the synchronous algorithm are more subject to fluctuations than those of the asynchronous one according to the configuration of the system. This variation of performances comes from the locality of the communications in the algorithm. As said above, the main communications are performed between neighbors in the list of machines, thus the link speed between two consecutive machines directly influences the performances on those nodes. In the context of our experiments, local links are faster than distant links between sites. Thus, when using an ordering by site, only a few links are significantly slower than others (those between two different sites), whereas in the interleaved configuration, all the links are slow. Hence, in that last context, all the nodes are penalized by slower communications whereas in the other context only a few of them are penalized. Finally, this has a direct effect on the number of iterations needed to converge and thus on the global performances of the algorithm.

6.2.1 Discussion on the speedup in heterogeneous clusters

No efficiency and speedup have been given here since it is not representative to compute efficiency with heterogeneous machines. Nevertheless, the sequential version has been tested on the fastest machine used in the system and it appears that the speedup merely follows the linear case. This result is not as natural as it seems. As the system is heterogeneous, the nodes have different powers and the system can be estimated to have the average power of the machines. Unfortunately, if we compare parallel performances to sequential ones which are obtained on a machine which corresponds to the average power of the cluster, they are super-linear.

The first idea which comes to mind is a problem of cache miss. Nonetheless, even if cache miss can play a role with large scale problems, this behavior is mainly due to the intrinsic structure of the non-linear problem.

If we consider the sequential iterative process, there is a spatial dependence between the components. At the beginning of the process, the antibodies are located on the few first spatial components and the evolution of the system comes from these components. This means that when a given spatial component is updated in the iterative process, its modification will not be significant, i.e. larger than the convergence threshold, as long as its two previous neighbors have not significantly evolved. So, in the iterative process there is what we can call a propagation phase preceding the statement that a component actually evolves when it is updated.

In the parallel version, the components are distributed to the processors. So, the propagation phase also appears at the node level in the following way: one node evolves when at least one of its spatial components evolves. Hence, at the beginning of the problem, only a certain number of the first nodes actually evolve in the system. This implies that the ordering of the machines in the system has not only a direct impact on the communications but also on the computation speed. In our previous experiment, the first machines are globally faster than the average computation speed of the system and this makes the propagation phase be processed faster than the sequential version on a machine with this average speed.

In order to exhibit this phenomenon, two series of tests have been performed on a local cluster of five machines with different powers: two Athlon 1.4Ghz, one P3 750Mhz, one P3 450 Mhz and one P2 350Mhz. In the first series, the machines are ordered from the fastest to the slowest and in the second series, the machines are in the opposite order. By the higher control we have on the performances of the local network (latency and bandwidth), we can maintain the same communication conditions between the two tests. Thus, the communications will not sharply vary from a test to the other and only the impact of the ordering of the processors will generate differences between the resulting times. As shown in Table 3, the ordering of the nodes strongly influences the performances of this kind of application.

machine speed	sequential time (s)	
on the slowest machine	7039	
on the average machine	2858	
on the fastest machine	1794	
ordering	time (s)	speedup
theoretical time on a cluster with the average speed	571,78	5
fastest $\rightarrow$ slowest	368,62	7,75
slowest $\rightarrow$ fastest	1457	1,96

Table 3: Sequential and parallel performances in function of the order of the machines.

Consequently, if we consider the execution of our algorithm with no load-balancing and a homogeneous data distribution on a heterogeneous cluster of N nodes, the speedup obtained according to the sequential execution on a machine with power P_s will be:

$$\frac{P_{min}}{P_s} \times SU(P_{min}, N) \leq Speedup \leq \frac{P_{max}}{P_s} \times SU(P_{max}, N)$$

where

$$P_{min} = \min_{i=node_1}^{node_N} Power(i) \quad \text{and} \quad P_{max} = \max_{i=node_1}^{node_N} Power(i)$$

and $SU(p,n)$ is the speedup obtained on a homogeneous cluster of n nodes with power p according to the sequential execution on a machine with the same power p. The lower bound corresponds to a no-propagation phase in the iterative process or to a propagation phase where the nodes which make the system evolve are the slowest. On the opposite, the upper bound can be reached when the entire iterative process is a propagation phase and the nodes involved in the evolution of the system are the fastest.

Even if this phenomenon is particular to our non-linear test problem, it is worth pointing it out because it may occur in numerous other non-linear applications where there is a similar propagation phase (such as IVPs). Moreover, this propagation phase can be generalized to a disordered propagation which occurs when there is at least one node whose components do not evolve when they are updated. This finally corresponds to an unbalance case since some nodes do not actively participate to the resolution of the system and can thus be considered as idle. Hence, the benefit of load-balancing in such algorithms becomes evident. It is experimentally demonstrated in the following paragraph.

6.2.2 Load-balancing

The last point of our experimental study is the interest of load-balancing in our asynchronous algorithms. It is very important to notice that asynchronism is not an alternative to load-balancing. In fact, as said in Section 6.2, asynchronism automatically performs a very good overlapping of communications by computations. Nevertheless, as pointed out at the end of this same section, this does not solve at all the problem of load-balancing, especially in a heterogeneous system.

In addition to the problem of disordered propagation, the presented version of our algorithm homogeneously distributes the data to the processors. Thus, if the algorithm is executed on N machines with one of them being much slower than the others, this machine will have to treat the same amount of data than the others and its computation time will then be greater than the others. Hence, it will sharply slow down the overall computation time. It is then important to notice here that this problem is merely independent from the communication problem. In fact, there are two distinct major improvements that can be realized on a given parallel iterative algorithm: the overlapping of communications by computations with asynchronism, and the speedup of computation time with load-balancing. Moreover, the speedup can be enhanced by two different means: a classical load-balancing which corresponds to an appropriate distribution of data according to the relative powers of the machines and an adaptive load-balancing which makes a repartition of the data which actually evolve at the considered time. This last solution is dynamic by nature and then must sometimes be performed during the iterative process. In the following experiments, the first enhancement has been used. The second one requires a more complete study and is the subject of another work [1].

Two series of experiments have been conducted with the same heterogeneous cluster as the one used for the non-linear problem but for the fourth site. The first series uses the asynchronous algorithm presented above, that is to say, with homogeneous distribution of work (data to process). The second one uses a modified version which takes into account an evaluation of the relative powers of the machines in order to distribute the work to the nodes. This second version is not dynamic since the relative powers of the machines are estimated before the execution of the algorithm and are then given as parameters. That is obviously not an optimized version but only a simple extension of our algorithm allowing us to show the interest of load-balancing. The results are presented in table 4.

N	non-balanced	balanced	gain
10000	296.85	213.73	28.0%
20000	570.86	418.79	26.6%
30000	927.93	617.05	33.5%

Table 4: Execution times (in seconds) of non-balanced and balanced asynchronous algorithms on a heterogeneous cluster of eight machines scattered on three sites.

In this table, it clearly appears that even with the very simple static load-balancing strategy used here, there is a great improvement in the execution times of our asynchronous algorithm. Thus, even better performances can be expected in a dynamically load-balanced version of our algorithm, where the relative speeds would be regularly estimated during the execution of the iterative process, as shown in [1]. It can also be observed that the gain does not directly depend on the size of the problem.

7 Conclusion

A complete description of asynchronous iterative algorithms has been given. Then, an experimental study of synchronous and asynchronous iterative algorithms has been presented in the context of grid computing, i.e. heterogeneous distant machines. The test problems considered represent two wide classes of scientific problems, namely, problems which lead to classical stationary (time-independent) linear systems and problems which lead to non-stationary (time-dependent) non-linear systems.

The results of these experiments clearly show that asynchronism is very well suited to this context since it obtains better performances than synchronism in all cases. An important conclusion of this study is that the impact of asynchronism on the performances is not just positive but of essential interest in the field of grid computing. This is mainly due to a natural and efficient overlapping of communications by computations which decreases the negative influence of the communications (by their heterogeneity between links and their unpredictable bandwidth variations) on the global performances. This positive impact also exists in the framework of dedicated parallel computers and/or local homogeneous clusters but may only appear in large configurations where communications become critical.

A complete discussion about the relevance and definition of the speedup information in the context of heterogeneous clusters has been addressed. In addition, the impact of the logical ordering of the machines on the global performances of asynchronous iterative algorithms for a large class of non-linear problems has been explained and experimentally confirmed.

Another important conclusion is that it seems judicious to combine load-balancing and asynchronism in the grid computing context whereas these two notions are quite mutually exclusive in the context of dedicated parallel computers or local homogeneous clusters. In this study, the distinction between asynchronism and load-balancing has been explicitly made. As has been pointed out, the load balancing of asynchronous algorithms is an entire problem in itself which requires a separate complete study. However, a preview of the interest of using load-balancing together with asynchronism has been experimentally given. Indeed, a great improvement in the performances has been obtained although a simple static load-balancing strategy has been used.

It is also very interesting to notice that the efficiency of such algorithms can be greatly enhanced without requiring a comparable increase in the complexity of programming. In fact, it can be even simpler with a multi-threaded environment like PM2, with which the programming of asynchronous communications is quite straightforward. The only additional difficulties are mainly the convergence detection and the halting procedure.

Finally, the asynchronous programming model can actually be recommended for high performance of parallel iterative algorithms in the grid computing context. Moreover, our last experiment shows all the interest to study the efficiency of the coupling of asynchronism with dynamic load-balancing.

References

[1] J. M. Bahi, S. Contassot-Vivier, R. Couturier, Coupling dynamic load balancing with asynchronism in iterative algorithms on the computational grid, in: International Parallel and Distributed Processing Symposium (IPDPS'2003), IEEE Computer Society Press, Nice, France, 2003, pp. 40a, 9 pages.

[2] D. B. Skillicorn, J. M. D. Hill, W. F. McColl, Questions and answers about BSP, Scientific Programming 6 (3) (1997) 249–274.

[3] D. P. Bertsekas, J. N. Tsitsiklis, Parallel and Distributed Computation: Numerical Methods, Prentice Hall, Englewood Cliffs NJ, 1989.

[4] M. E. Tarazi, Some convergence results for asynchronous algorithms, Numer. Math. 39 (1982) 325–340.

[5] K. Blathras, D. B. Szyld, Y. Shi, Timing models and local stopping criteria for asynchronous iterative algorithms, Journal of Parallel and Distributed Computing 58 (3) (1999) 446–465.

[6] A. Frommer, D. B. Szyld, Asynchronous two-stage iterative methods, Numerische Mathematik 69 (2) (1994) 141–153.

[7] J. M. Bahi, Asynchronous iterative algorithms for nonexpansive linear systems, Journal of Parallel and Distributed Computing 60 (1) (2000) 92–112.

[8] D. Szyld, Perspectives on asynchronous computations for fluid flow problems, Tech. rep., Department of Mathematics, Temple University (2000).

[9] D. Chazan, W. Miranker, Chaotic relaxation, Linear Algebra and Its Applications 2 (1969) 199–222.

[10] D. El Baz, A method of terminating asynchronous iterative algorithms on message passing systems, Parallel Algorithms and Algorithms 9 (1996) 153–158.

[11] A. Frommer, D. B. Szyld, On asynchronous iterations, J. Comp. Appl. Math. 123 (2000) 201–216.

[12] D. P. Bertsekas, J. N. Tsitsiklis, Parallel and distributed iterative algorithms: a selective survey, Automatica 25 (1991) 3–21.

[13] S. A. Savari, D. P. Bertsekas, Finite termination of asynchronous iterative algorithms, Parallel Computing 22 (1996) 39–56.

[14] J. Bahi, S. Contassot-Vivier, R. Couturier, Evaluation of the asynchronous iterative algorithms in the context of distant heterogeneous clusters., Parallel Computing 31 (5) (2005) 439–461.

[15] W. Lioen, J. De Swart, W. Van Der Ween, Test set for ivp solvers, Tech. Rep. NM-R9615, CWI Amsterdam (1996).

[16] K. Burrage, Parallel and Sequential Methods for Ordinary Differential Equations, Oxford University Press Inc., New York, 1995.

[17] C. W. Gear, The potential for parallelism in ordinary differential equations, Tech. Rep. 1246, Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana, Illinois (1986).

[18] E. Lelarasmee, A. Ruehli, A. Sangiovanni-Vincentelli, The wavefront relaxation method for time-domain analysis of large scale integrated circuits, IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems CAD-1 (1982) 131–145.

[19] M. E. Kyal, J.-C. Miellou, J. M. Bahi, Superlinear convergence of asynchronous waveform relaxation methods for nonlinear odes, in: Proceedings of the 9th international colloquium on differential equations, 1999, pp. 119–126.

[20] A. Frommer, B. Pohl, A comparison result for multisplittings and waveform relaxation methods, Numerical linear algebra with applications 2 (4) (1995) 335–346.

[21] J. M. Bahi, K. Rhofir, J.-C. Miellou, Parallel solution of linear DAEs by multisplitting waveform relaxation methods, Linear Algebra and Its Applications 3 (332–334) (2001) 181–196.

[22] J. M. Bahi, E. Griepentrog, J. C. Miellou, Parallel treatment of a class of differential-algebraic systems, SIAM Journal on Numerical Analysis 33 (5) (1996) 1969–1980.

[23] R. Namyst, 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, Vol. 11, Elsevier, North-Holland, 1996, pp. 279–285.

[24] W. Gropp, E. Lusk, A. Skjellum, Using MPI : portable parallel programming with the message passing interface, MIT Press, 1994.

[25] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, V. Sunderam, PVM: A Users' Guide and Tutorial for Networked Parallel Computing, MIT Press, 1994.