Dynamic Load Balancing and Efficient Load Estimators for Asynchronous Iterative Algorithms

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

Dynamic Load Balancing and Efficient Load Estimators for Asynchronous Iterative Algorithms

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

***Abstract*— In a previous paper [1], we have shown the very high power of asynchronism for parallel iterative algorithms in a global context of grid computing. In this article, we study the interest of coupling load balancing with asynchronism in such algorithms. After proposing a non-centralized version of dynamic load balancing which is best suited to asynchronism, we verify its efficiency by some experiments on a general Partial Differential Equation (PDE) problem. Finally, we give some general conditions for the use of load balancing to obtain good results with this kind of algorithms and discuss the choice of the residual as an efficient load estimator.**

***Index Terms*— Parallel iterative algorithms, asynchronism, load-balancing.**

I. Introduction

In the context of scientific computations, iterative algorithms are very well suited to a large class of problems (see for example [2]–[7]). In many cases, they are preferred to direct methods and sometimes they are even the single way to solve the problem (e.g. root polynomial problems). Direct algorithms give the exact solution of a problem within a finite number of operations whereas iterative algorithms provide an approximation of it. We say that they converge (asymptotically) towards the solution. When dealing with very large-sized problems, iterative algorithms are preferred especially if they give a good approximation within a small number of iterations.

The latter properties have led to a good expansion of parallel iterative algorithms (PIAs). Nevertheless, most of those parallel versions are synchronous. We have shown in [1] all the interest of using asynchronism in such parallel iterative algorithms especially in a global context of grid computing. Moreover, we have also shown in [8] that static load balancing can sharply improve the performances of our algorithms.

In this article, we discuss the general interest of using dynamic load balancing in asynchronous iterative algorithms and we empirically show its major efficiency in the global context of grid computing.

Due to the nature of PIAs, a centralized version of load balancing would not be well suited in a global context of grid computing. Hence, the technique used in this study works locally between neighboring processors. In our case, the neighborhood is determined by the communications between processors. Two nodes are defined as neighbors if they have to exchange data to perform their job. We evaluate the gain of the load balancing in some experiments on a representative chemical test problem which is described by a Partial Differential Equation (PDE). Finally, we study the impact of the load estimator on the performances in the particular case of PIAs and propose to use the residual instead of the classical amount of data to evaluate the load. The residual is defined by the max norm of the difference between data values from two consecutive iterations. In fact, choosing such a load estimator takes into account the actual progress of the iterative process.

The following section recalls the principles of asynchronous iterative algorithms and replaces them in the context of PIAs. Then, Section III presents a small discussion about the motivations of using load balancing in such algorithms. A brief overview of related works concerning non-centralized load balancing techniques is given in Section IV. An example of application is exhibited with a chemical reaction problem detailed in Section V. The corresponding algorithm and the insertion of load balancing are then detailed in Section VI. Experimental results are given and interpreted together with a discussion about the best conditions of use in Section VII. Finally, we study the impact of the load estimator on the performances of load balancing in Section VII-B.

II. What are asynchronous iterative algorithms ?

A. Iterative algorithms: background

Iterative algorithms have the following structure

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

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

Let x^k be partitioned into m block-components $X_i^k,\ i \in \{1, ..., m\}$, and f be partitioned in a compatible way into m block-components F_i, then equation (1) can be written as

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

and the iterative algorithm can be parallelized by letting each of the m processors updates a different block-component of x according to (2) (see [9], [10]). At each stage, the i^{th} processor knows the value of all components of X^k on which F_i depends, computes the new values X_i^{k+1}, and communicates

This research was supported by the STIC Department of the CNRS

The authors are with LIFC (Laboratoire d'Informatique de l'Université de Franche-Comté), FRE CNRS 2661, IUT de Belfort-Montbéliard, BP 527, 90016 Belfort, France

those on which other processors depend to make their own iterations.

Considering fully asynchronous iterative algorithms, the model is as follows:

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

Fully asynchronous networks including overlapping updating were characterized by Herz and Marcus in [11].

In the classical definition of fully asynchronous networks, we denote by $J(t)$ the set of nodes updated at time t and by $X_j^{s_j^i(t)}$ the state of the group j of nodes available for the group i at time t. $s_j^i(t)$ is the iteration number of the data from group j available on group i at time t. It is defined by $s_j^i(t) = t - r_j^i(t) \leq t$, where $r_j^i(t)$ denotes the delay of the group of nodes j with respect to the group i. Moreover $\lim_{t\to\infty} s_j^i(t) = \infty$, which means that although the delays are unbounded, they follow the evolution of the system. Finally, it can be noticed that the groups X_l may be reduced to a single node x_{i_l}.

Then, the fully asynchronous dynamic of the n-nodes network associated to the given transition function F and to the activation set J, and with the initial configuration $X^0 = \left(X_1^0, ..., X_m^0\right)$, is described by Algorithm 1.

Algorithm 1 Asynchronous iteration

```
Given an initial state X^0 = (X_1^0, ..., X_m^0)
for each time step t = 0, 1, ... do
  for each block-components i = 1, ..., m do
    if i ∈ J(t) then
      X_i^{t+1} = F_i(X_1^{s_1^i(t)}, ..., 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 PIA. This implies that if an algorithm converges in this context, it will also converge in more synchronous ones.

In this model, the residual of block i is defined by the max norm of the difference between its values from two consecutive iterations:

$$residual_i^t = ||X_i^t - X_i^{t-1}||_\infty = max_j|X_{i,j}^t - X_{i,j}^{t-1}|$$

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

B. A categorization of parallel iterative algorithms

Since this article deals with what we commonly call asynchronous iterative algorithms, it appears necessary, to make it clear, to detail the class of parallel iterative algorithms. In this part, we present classes of algorithms which can actually be implemented and used. The main difference with fully asynchronous algorithms lies in the delays which are bounded in the practical case. So, this class can be decomposed into three main parts:

Synchronous Iterations - Synchronous Communications (SISC) algorithms: all processors begin the same iteration at the same time since data exchanges are performed at the end of each iteration by synchronous global communications. After parallelization of the problem, these algorithms have exactly the same behavior as the sequential version in terms of the iterations performed. Hence, their convergence is directly deducible from the initial algorithm. Unfortunately, the synchronous communications strongly penalize the performances of these algorithms. As can be seen in Figure 1, there may be a lot of idle times (white spaces) between iterations (grey blocks) depending on the speed of communications.

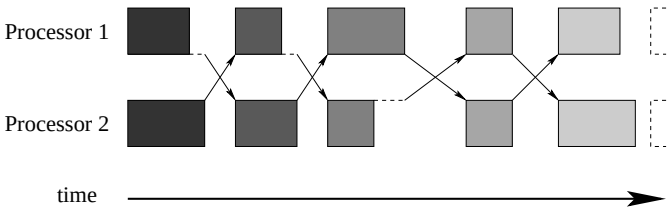

Fig. 1. Execution flow of a SISC algorithm with two processors.

Synchronous Iterations - Asynchronous Communications (SIAC) algorithms: all processors also wait for the receptions of needed data updated at the previous iteration to begin the next one. Nevertheless, each data (or group of data) required on another processor is sent asynchronously as soon as it has been updated, so that the remaining computations of the current iteration overlap its communication. This scheme lies on the probability that data will be received on the destination processor before the end of its current iteration, and then will be directly available for the next iteration. Hence, this partial overlapping of communications with computations during each iteration implies shorter idle times and thus better performances. Since each processor begins its next iteration as soon as it has received all the needed data updated from the previous iteration, all the processors may not begin their iterations at the same time. Nonetheless, in terms of iterations, the notion of synchronism still holds in this scheme since at any time t, it is not possible to have two processors performing different iterations. In fact, at each time t, the processors are either computing the same iteration or idle (waiting for data). Hence, from the algorithmic point of view, this category of algorithms, like the SISC one, performs the same iterations as the sequential version. Thus, they have the same convergence properties. Unfortunately, this scheme does not completely eliminate idle times between iterations, as shown in Figure 2. In fact, some communications may be longer than the computation of the current iteration and the sending of the last updated data on the latest processor can not be overlapped with computations. It can also be seen on

that figure that the order of the communications may not be respected.

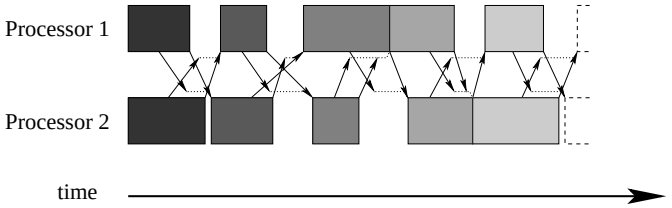

Fig. 2. Execution flow of a SIAC algorithm with two processors. In this example, the first half of data is sent as soon as updated and the second half is sent at the end of the iteration.

Asynchronous Iterations - Asynchronous Communication (AIAC) algorithms: all processors perform their iterations without taking care of the progression of the other processors. They do not wait for any reception of needed data coming from other processors but they keep on computing, trying to solve the given problem with the current version of data available at that time. Since the processors do not wait for communications, there are no more idle times between the iterations as can be seen in Figure 3. Although widely theoretically studied (see for example [9], [12]–[14]), very few implementations and experimental analyses have been carried out, especially in the context of grid computing. In the literature, there is a major algorithmic model corresponding to these algorithms expressed in two main theoretical results, the Bertsekas and Tsitsiklis theorem [13] and the El Tarazi's theorem [12]. The former is based on nested sets whereas the latter uses contraction properties. Nevertheless, several variants can be deduced from these models depending on when the communications are performed and when the received data are incorporated into the computations, see e.g. [3], [15]. Figure 3 depicts a general version of an AIAC algorithm with a data decomposition in two halves for the asynchronous sendings. This type of algorithms requires 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 such as choosing the right criterion for convergence detection and the right halting procedure. There are also some implementation problems due to the asynchronous communications which imply the use of an adequate programming environment. Nevertheless, despite all these obstacles, these algorithms are quite convenient to implement and are the most efficient ones especially in a global context of grid computing as we have already shown in [1]. This comes from the fact that they allow communication delays to be substantial and unpredictable which is a typical situation in large networks of heterogeneous machines.

III. Why using load balancing in the AIAC model ?

The scope of this paper is to study the interest of using dynamic load balancing in the AIAC model. One of our goals is to show that, contrary to a generally accepted idea, asynchronism does not exempt from distributing the workload

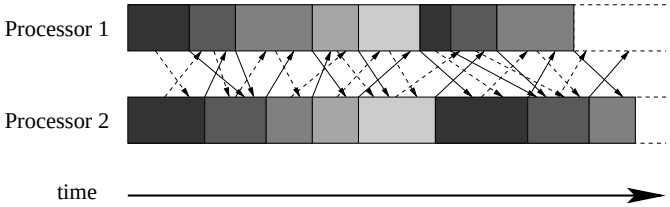

Fig. 3. Execution flow of an AIAC algorithm with two processors. Dashed lines represent the communications of the first half of data, and solid lines are for the second half.

efficiently. Indeed, load balancing can efficiently take into account the heterogeneity of the machines involved in the parallel iterative computation. This heterogeneity can be found at the hardware level when using machines with different speeds but also at the user level if the machines are used in multi-users or multi-tasks contexts. All these cases are especially encountered when dealing with grid computing.

In numerous problems resolved by iterative algorithms, the progression towards the solution is not the same for all the components of the system and some of them reach their partial fixed point faster than others. By performing an appropriate load balancing with some criteria based on this progression (the residual for example) instead of the classical load estimator (amount of data), it is then possible to enhance the distribution of the actually evolving computations over the processors. Thus, even in a homogeneous context, this coupling has the great advantage to deal with the evolution of the computation during the iterative process.

Hence, there are two main ideas motivating the coupling of load balancing and AIAC algorithms:

- when the workload is efficiently distributed on the system, asynchronism allows us to efficiently overlap communications with computations, especially on networks with very fluctuating latencies and/or bandwidths.
- even if AIAC algorithms are potentially more efficient than the other models, they do not take into account the workload distribution over the processors. If this is well managed, it can reasonably make us expect yet better performances.

The great advantage of AIAC algorithms in this context is that they are far more flexible than synchronous ones. Indeed, it is less imperative to have at all times exactly the same amount of work on each processor. The goal here is thus to avoid too large differences of progression between processors. A non-centralized strategy of load balancing appears to be necessary since it avoids global communications which would synchronize the processors.

IV. Existing non-centralized load balancing models and related works

The load balancing problem has been widely studied from different perspectives and in different contexts [16]. A categorization of the various techniques for load balancing can be found in [17] based on criteria like centralized/distributed, static/dynamic, and synchronous/asynchronous. To be concise, we present here the few techniques which are the most suited to AIAC algorithms.

In the context of parallel iterative computations, the load-balancing scheme must be non-centralized and iterative by nature. Local iterative load balancing algorithms were first proposed by Cybenko in [18]. These algorithms iteratively balance the load of a node with its neighbors until the whole network is globally balanced. There are mainly two iterative load balancing algorithms: diffusion algorithms [18]–[20] and dimension exchange algorithms [17]–[19], [21]. Diffusion algorithms assume that a processor simultaneously exchanges load with its neighbors, whereas dimension exchange algorithms assume that a processor exchanges load with only one neighbor (along each dimension or link) at each time step. All these works took place in the context of a homogeneous system. The problem of load balancing in a heterogeneous system has been addressed by Elsasser et al in [22].

Unfortunately, these techniques are all synchronous which is not convenient for the AIAC class of algorithms. Bertsekas and Tsitsiklis have proposed in [13] an asynchronous model for iterative non-centralized load balancing. The principle is that each processor has an evaluation of its load and those of all its neighbors. Then, at some given times, this processor looks for its neighbors which are less loaded than itself. Finally, it distributes a part of its load to all these processors. Nevertheless, the authors have focused their work on proving that this iterative load balancing asymptotically leads to a homogeneous distribution of the work. In our work, the asynchronism occurs both in the numerical application and in the load balancing. In addition, we describe how to efficiently perform the coupling of asynchronous load balancing and asynchronous iterative algorithms.

A variant evoked by the authors is to send a part of the work only to the lightest-loaded neighbor. This last variant has been chosen for implementation in our AIAC algorithms since it has the most suited properties: it maintains the asynchronism in the system with only local communications between two neighboring nodes.

In the following section, we describe a typical problem of Partial Differential Equations (PDEs) which has been chosen for our experimentations.

V. Test problem

To perform our experiments, a classical example of non-linear problem has been chosen since iterative algorithms are mostly used for this kind of problems.

Our test problem is known as the Brusselator problem. It models a chemical reaction mechanism which leads to an oscillating reaction. It deals with the conversion of two elements A and B into two others C and D by the following series of steps:

$$\begin{array}{rcl} A & \rightarrow & X \\ 2X+Y & \rightarrow & 3Y \\ B+X & \rightarrow & Y+C \\ X & \rightarrow & D \end{array} \quad (3)$$

There is an autocatalysis and when the concentrations of A and B are maintained constant, the concentrations of X and Y oscillate with time. For any initial concentrations of X and Y, the reaction converges towards what is called the limit cycle of the reaction. This is the graph representing the concentration of X against those of Y and it corresponds in this case to a closed loop.

The desired results are the evolutions of the concentrations *u* and *v* of both elements X and Y along the discretized space in function of time. If the discretization is made with N points, the evolution of the u_i and v_i for $i = 1, ..., N$ is given by the following differential system:

$$\begin{array}{l} u_i' = 1 + u_i^2 v_i - 4u_i + \alpha(N+1)^2(u_{i-1} - 2u_i + u_{i+1}) \\ v_i' = 3u_i - u_i^2 v_i + \alpha(N+1)^2(v_{i-1} - 2v_i + v_{i+1}) \end{array} \quad (4)$$

The boundary conditions are:

$$\begin{array}{rcl} u_0(t) & = & u_{N+1}(t) = \alpha(N+1)^2 \\ v_0(t) & = & v_{N+1}(t) = 3 \end{array}$$

and the initial conditions are, for each $i \in \{1, ..., N\}$:

$$\begin{array}{l} u_i(0) = 1 + sin(2\pi x_i) \text{ with } x_i = \dfrac{i}{N+1} \\ v_i(0) = 3 \end{array}$$

Here, we fix the time interval to $[0, 10]$ and $\alpha = \frac{1}{50}$. N is a parameter of the problem.

This problem corresponds to a large stiff system of PDEs formulated as an IVP (Initial Value Problem) which is very common in many scientific domains. It is well-known (see for example [4], [23], [24]) that the use of implicit methods is required and then, large systems of nonlinear equations have to be solved at each iteration. Further information about this problem and its formulation can be found in [5].

VI. AIAC algorithm and load balancing

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.

It must be noticed that the principle of AIAC algorithms is generic and can be adapted to every iterative process under convergence hypotheses which are satisfied for a large class of problems. In most cases, the adaptation comes from the data dependencies, the function to approximate and the methods used for intermediate computations. By this way, these algorithms can be used to solve either linear or non-linear systems which can be stationary or not.

In the case of our non-linear problem, the u_i and v_i of the system are represented in a single vector as follows:

$$y = (u_1, v_1, ..., u_N, v_N)$$

with $u_i = y_{2i-1}$ and $v_i = y_{2i}$, $i \in \{1, ..., N\}$. The denomination y, used in the classical formulation of the Brusselator problem, is equivalent in our context to the x vector in equation (1).

The y_j functions, $j \in \{1, ..., 2N\}$ thereby defined will also be referred to as spatial components in the remaining of the article.

A. Unbalanced AIAC algorithm

To solve the system (4), we use a two-stage iterative algorithm:

- At each iteration:
  - use of the implicit Euler algorithm to approximate the derivative,
  - use of the Newton algorithm to solve the resulting nonlinear system.

The inner procedure will be called `Solve` in our algorithm. In order to exploit the parallelism, the y_j functions are initially homogeneously distributed over the processors. Since these functions are represented in a one-dimensional space (the state vector y), we have chosen to logically organize our processors in a linear way and map the spatial components (y_j functions) on them. This organization is directly deduced from the data dependencies of the problem in order to exploit as much parallelism as possible. Hence, each processor applies the Newton method to its local components using the needed data from other processors involved in its computations. From the problem formulation given in Section V, it arises that the processing of components 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 owns at least two functions y_j, the non-local data needed by each processor to perform its iterations only come from the previous processor and the following one in the logical organization. In practical cases, there will be much more than two functions on each node.

In Algorithm 2, the core of the AIAC algorithm without load balancing is presented. Since the convergence detection and halting procedure are not directly involved in the modifications brought by the load balancing, only the iterative computations and corresponding communications are detailed.

In this algorithm, the arrays `Ynew` and `Yold` have always the same organization which consists in the following ordered contents: the last two components of the left neighbor, the local components of the current node and the first two components of the right neighbor. This organization is depicted in Figure 4 where data (in grey levels) have been drawn on two separated lines (to be clearer) whereas they should all be represented on the same line. Hence, two vertical instances of data with the same abscissa actually represent the same data.

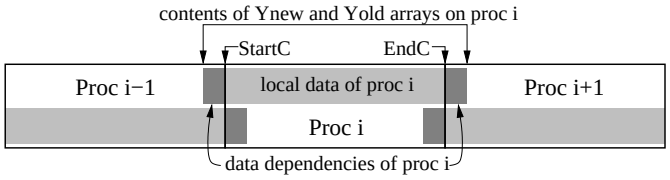

Fig. 4. Contents of data arrays Ynew and Yold on processor i.

This structure will have to be maintained even when performing load balancing. The `StartC` and `EndC` variables are used to indicate the beginning and the end of the local components actually computed by the node as shown in Figure 4. Finally, the δt variable represents the precision of the time discretization needed to compute the evolution of spatial components in time.

In order to facilitate and enhance the implementation of asynchronous communications, we have chosen to use the PM2 multi-threaded programming environment [25]. This kind of environment allows us to make the sending and receiving operations in additional threads rather than in the main program. This is why the receptions of data do not directly appear in our algorithms. In fact, they are localized in functions called by a thread created at the beginning of the program and dealing with incoming messages. Thus, when a sending operation is performed on a given processor, the function which will manage the message on the destination node must be specified. In the same way, the asynchronous sending operations appearing in our algorithms actually correspond to the creation of a communication thread calling the related sending function. According to the linear organization of the processors explained above, each node has a left and a right neighbor, except for the first node which only has a right neighbor and the last one which only has a left neighbor. Hence, sending operations towards the left (/right) neighbor are referred to as left (/right) communications in Algorithm 2.

Algorithm 2 Unbalanced AIAC algorithm

```
Initialize the communication interface
NbProcs = Number of processors
MyRank = Rank of the processor
Yold, Ynew = Arrays of local spatial components
StartC, EndC = Indices of the first and last local spatial
               components
ReT = Range of evolution time of the spatial components
StartT, EndT = First (0) and last (ReT/δt) values of time

Initialization of local data
repeat
  for j=StartC to EndC do
    for t=StartT to EndT do
      Ynew[j,t] = Solve(Yold[j,t])
    end for
    if j=StartC+2 and MyRank > 0 then
      if there is no left communication in progress then
        Send asynchronously the first two local compo-
        nents to left processor
      end if
    end if
  end for
  if MyRank < NbProcs-1 then
    if there is no right communication in progress then
      Send asynchronously the last two local components
      to right processor
    end if
  end if
  Copy Ynew in Yold
until Global convergence is achieved
Display or save local components
Halt the communication system
```

Data reception functions only consist in receiving two components from the corresponding neighbor (left or right) and in putting them at the right place, before or after the

local components, in array Ynew. It can be noticed that all the variables in Algorithm 2 can be directly accessed by the reception functions since they are in threads which share the same memory space.

For each communication function (sending or receiving), a mutual exclusion system is used to avoid simultaneous threads to perform the same kind of communication with different data which could lead to incoherent situations and also to a useless overloading of the network. This also has the advantage to generate fewer communications. Hence, the AIAC variant used here and detailed in Figure 5 is slightly different from the general case given in Figure 3.

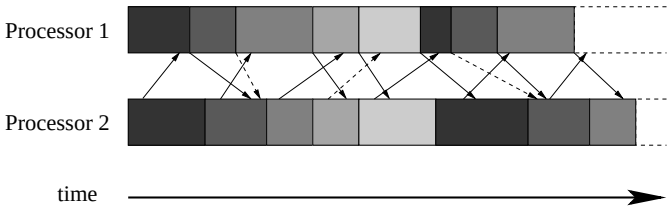

Fig. 5. Execution flow of our AIAC variant with two processors. Dashed lines represent communications which are not actually performed due to mutual exclusion. Solid lines starting during iterations correspond to left sendings whereas those at the end of iterations are for right ones.

B. Load-balanced AIAC algorithm

As evoked in Section IV, Bertsekas and Tsitsiklis [13] have proposed a theoretical algorithm to perform load balancing asynchronously and have proved its convergence. We have used this model to design our load balancing algorithm adapted to parallel iterative algorithms and particularly to AIAC algorithms on the grid.

An additional contribution of our study is to explicitly describe the implementation of the load balancing scheme and to give very efficient load estimators to be used with AIACs.

Our load balancing scheme of the AIAC algorithm is given in Figure 6. In order to provide a general and uniform version of this scheme, all the sendings in this Figure are performed at the end of the iterations. According to the treated problem, these sendings may appear sooner in the iterations (as shown in Figure 5 for the Brusselator problem). However, these possible variants do not affect the load balancing scheme.

In this scheme, each processor periodically tests if it has to balance its load with one of its neighbors, the left or the right one here. If needed, it sends a given amount of data to its lightest loaded neighbor. This step corresponds to the balloon (1) in Figure 6.

Concerning the load balancing process itself, most of the additional parts take place at the beginning of the main loop of the iterative algorithm. At each iteration, we test if a load balancing process has been performed. This may be a load reception (as indicated by balloon (2) in the figure) or a load sending. In these cases, data arrays have to be resized at the end of the iteration during which the load balancing was performed, in order to contain just the local components affected to the node. Hence, a second test is performed to find the nature of the load balancing on the node (load reception or sending). In the former case, the arrays have to be enlarged in order to receive the additional data which then have to be copied in this new array. This step is indicated by balloon (3a) in the figure. In the latter case, arrays have to be reduced and no data copying is necessary. This is indicated by the balloon (3b) of the figure. Once the arrays have been correctly updated, the computations can be performed and the overall iterative process resumes as if nothing special had happened. The only difference being the data distribution (which has changed) between the two processors.

If no load balancing has been performed, two tests have to be done to eventually perform a load balancing towards the left or right processor. The first one allows us to try load balancing periodically every k iterations. This is useful to tune the frequency of load balancing during the iterative process which directly depends on the considered problem. In some cases, a high frequency will be efficient whereas in other cases lower frequencies will be recommended since too much load balancing could take most of the computation time of the process according to the iterations, especially with low bandwidth networks.

The second test detects if a communication from a previous load balancing is not finished yet. In this case, the trial is delayed till the next iteration and so on until the previous communication is achieved. In the other case, the corresponding function is called.

It can be noticed that according to the current organization of these tests, the left load balancing is tested before the right one, which could seem to give an advantage to it. In fact, this is not actually the case and this does not alter the generality of our algorithm. This has only been done to avoid simultaneous load balancings of a processor with its two neighbors, which would not correspond to the model used.

Finally, the last point in the main algorithm concerns the data sendings performed at each iteration. Since the arrays may change from an iteration to another, we have to ensure that the received data correspond to the local data before (/after) the current arrays and that they can thus be safely put before (/after) them. This is why the global position of the first (/last) two components are joined to the data. Moreover, in order to decide whether or not to balance the load, the local load evaluations are used and then sent together with the components.

In Algorithm 3 is presented the load-balanced version of the AIAC algorithm given in Section VI-A. To be clearer, implementation details which are related to the programming environment used are not shown.

In Algorithm 4 is detailed the function to balance the load with the left neighbor. Obviously, this function has its symmetrical version for the right neighbor. Its first step is to test if a balancing is actually needed by computing the ratio of the workloads on the two processors and comparing it to a given threshold. If satisfied, the number of data to be sent is then computed and another test is done to verify that the number of remaining data on the processor will be large enough. This is done to avoid the famine phenomenon on slowest processors. Finally, the first (/last) data whose number has been previously computed are asynchronously sent with two more components which will represent the dependencies

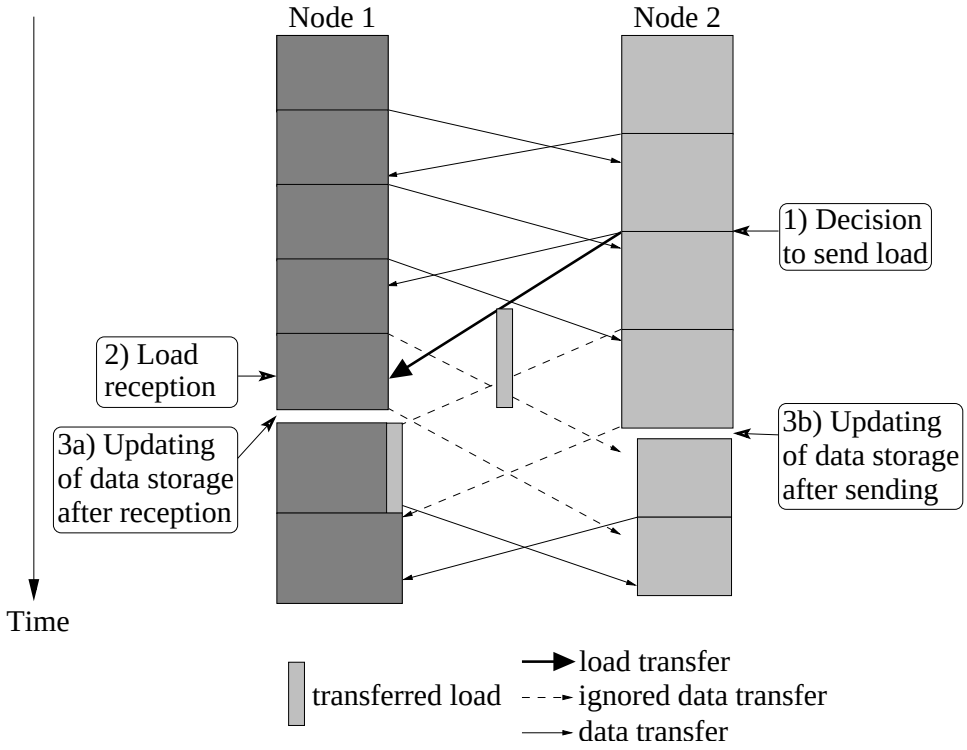

Fig. 6. Execution flow of our load balancing scheme

of the left (/right) processor. These two additional data will continue to be computed by the current processor but their values will be sent to the left (/right) processor to allow it to perform its own computations with updated values of its data dependencies. In the same way, the two components before (/after) those two ones will be kept on the current processor and become its new data dependencies related to the left (/right) neighbor.

Concerning the reception functions, the first type, exhibited in Algorithm 5, is related to the load balancing whereas the second type, given in Algorithm 6, deals with the classical data exchanges induced by dependencies. The former function consists in placing additional data into a temporary array until they are copied in the resized array Yold, then the temporary array is destroyed. Once the reception is done, the flags indicating the completion of a load balancing communication and its nature are set. The latter function corresponds to the data reception function used in Algorithm 2. Nonetheless, some modifications appear in this version, the global position of the received data must be confronted to the expected one before stocking them in the array. Besides, another additional information to be received is the load evaluation obtained on the source node.

Concerning the behavior of this load-balanced version, since the iterative process is not modified in itself, we can guarantee that the load-balancing will not affect the convergence property. Thus, any converging non-balanced asynchronous algorithm will also converge with our load-balancing scheme and will provide a similar result according to the accuracy threshold.

It can be noticed that under the specified accuracy threshold, it is not possible to ensure exactly the same values for the result. This is not properly due to the load-balancing scheme but mainly to the non-deterministic nature of asynchronous algorithms. Indeed, this fact even holds for asynchronous algorithms without load-balancing since the delays may be different from an execution to another. It comes that the local computations on each processor (the series of iterates) may not exactly be the same and thus the final result too. The only thing which is ensured with our initial assumptions is that the algorithm will converge in a small space around the exact solution according to the specified accuracy.

VII. Experiments

In order to perform our experiments, we have used the PM2 (Parallel Multi-threaded Machine) environment [25]. Its first goal is to efficiently support irregular parallel applications on distributed architectures. We have already shown in [1] how convenient this kind of environment is to program asynchronous iterative algorithms in a global context of grid computing.

The context of our experiments is as follows: the space is discretized in 60000 points, the time interval is from 0 to 10 by steps of 0.05 and the required accuracy is $7e-4$.

A. Balancing vs not balancing

The evaluation of the gain obtained by coupling load balancing with asynchronism is obtained by comparing the balanced and non-balanced versions of our AIAC algorithm in two different contexts. The first is a local homogeneous cluster of PIII-733Mhz with a 100Mb/s network and the second is a collection of heterogeneous machines scattered on distant

Algorithm 3 Load-balanced AIAC algorithm

```
Initialize the communication interface
Variables from Algorithm 2
LBDone = boolean indicating if LB has just been performed
LBReception = boolean indicating if additional data from
              LB have been received
OkToTryLB = integer allowing to periodically test for
            performing LB. Initially set to 20
Initialization of local data
repeat
  if LBDone=true then
    if LBReception=true then
      Resize Ynew,Yold arrays after reception of additional data
      Complete new Yold array with additional data from temporary array
      LBReception=false
    else
      Resize Ynew,Yold arrays after the sending of transferred data
    end if
    LBDone=false
  else
    if OkToTryLB=0 then
      if there is no left LB communication in progress then
        TryLeftLB()
      else
        if there is no right LB communication in progress then
          TryRightLB()
        end if
      end if
    else
      OkToTryLB=OkToTryLB-1
    end if
  end if
  for j=StartC to EndC do
    ... /* The ... indicate a same part as in Algorithm 2 */
      Send asynchronously the first two local components
      together with their global position and the load
      evaluation of previous iteration to left processor
    ...
  end for
  ...
    Send asynchronously the last two local components
    together with their global position and the load
    evaluation of current iteration to right processor
  ...
until Global convergence is achieved
...
```

Algorithm 4 function TryLeftLB()
/* symmetrical for TryRightLB() */

```
Ratio = Ratio of load evaluations between local node and
        its left neighbor
NbLocal = Number of local data
NbToSend = Number of data to send to perform the
           load-balancing
Ratio=local load evaluation / left load evaluation
if Ratio>ThresholdRatio then
  Compute the number of data to send NbToSend
  if NbLocal-NbToSend>ThresholdData then
    Send asynchronously the first NbToSend+2 data to left
    processor /* +2 is added for data dependencies */
    OkToTryLB=20
    LBDone=true
  end if
end if
```

Algorithm 5 function RecvDataFromLeftLB()
/* symmetrical for RecvDataFromRightLB() */

```
Receive the number of additional data sent
Receive these data and put them in a temporary array
LBReception=true
LBDone=true
```

sites linked together with a 10Mb/s network. Since different configurations have been used, the heterogeneous context is described for each experiment. In all our experiments, the given results correspond to an average of a series of 20 executions.

Figure 7 shows the evolution of execution times in function of the number of processors on a local homogeneous cluster. The residual is used as the load estimator in the balanced version. It can be seen that both versions have a very good scalability. This is quite an important point since load balancing usually introduces sensitive overheads in parallel algorithms leading to quite moderate scalabilities. This good result mainly comes from the non-centralized nature of the balancing used in our algorithm. Nevertheless, the most interesting point is the large vertical offset between the curves

Algorithm 6 function RecvDataFromLeft()
/* symmetrical for RecvDataFromRight() */

```
if not accessing data array then
  Receive the global position and the two components from
  left node
  if global position corresponds to the two left data needed
  on local node then
    Put these data before local components in array Yold
  else
    Do not stock these data in array Yold
    /* array Yold is being resized */
  end if
  Receive the load evaluation obtained on the left node
end if
```

which denotes a high gain in performances. In fact, the ratio of execution times between the non-balanced and balanced versions varies from 6.2 to 7.4 with an average of 6.8. These results show all the efficiency of coupling load balancing with AIAC algorithms on a local cluster of homogeneous machines.

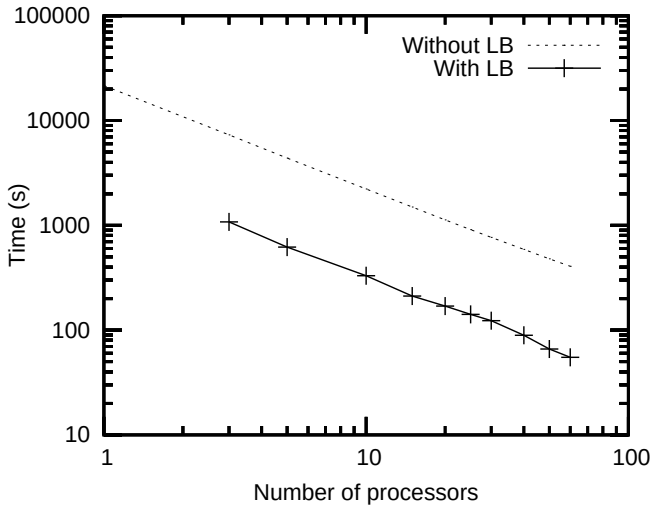

Fig. 7. Execution times (in seconds) on a homogeneous cluster

Concerning the heterogeneous cluster, fifteen machines have been used over three sites in France: Belfort, Montbéliard and Grenoble, between which the speed of the network may sharply vary. The organization of the system has been chosen irregular in order to get a grid computing context which is not favorable to load balancing. The machine types vary from a PII 400Mhz to an Athlon 1.6Ghz. In this cluster, no direct neighbors are similar in terms of power. Again, the load estimator used in the balanced version is the residual.

The results obtained in this context are given in Table I. Here also, the balancing brings a potential enhancement of the performances of the initial AIAC algorithm. In this case, the ratio is smaller than in the local case because of the larger cost of communications and thus of data migrations. Although this ratio remains very satisfying, this remark would imply a closer study concerning the tuning of the load balancing frequency during the iterative process. This is not within the scope of this article but it will probably be the subject of a future work.

version	not-balanced	balanced	ratio
execution time	515.3	105.5	4.88

TABLE I

EXECUTION TIMES (IN SECONDS) ON A HETEROGENEOUS SYSTEM

Despite this, the load balancing is more interesting in this context than in local clustering. This comes from the fact that in the homogeneous context, as was shown in [1], the synchronous and asynchronous iterative algorithms almost have the same behavior and performances whereas in the global context of grid computing, the asynchronous version reveals all its interest by providing far better results. Hence, we can reasonably deduce that load balancing AIAC algorithms with a load estimator based on the amount of data in a local homogeneous context would only produce slightly better results than their SISC counterparts. On the opposite, in the global context, the difference between SISC and AIAC load-balanced versions will be much larger. In fact, this last version will obtain the very best performances.

As explained in Section III and pointed out by these experiments, load balancing and asynchronism are thus not incompatible and can actually lead to very efficient parallel iterative algorithms.

The attainment of this efficiency lies on the way this coupling is performed and the context in which it is used. The first point has already been discussed and the important role played by the non-centralized nature of the balancing technique has been shown. Concerning the second point, there are also some conditions which should be verified on the treated problem to ensure good performances.

According to our experiments, at least four conditions required to get an efficient load balancing on asynchronous iterative algorithms have arisen. The first one concerns the number of iterations which must be large enough to make it worth performing load balancing. In the same way, the average time to perform one iteration must be long enough to have a reasonable ratio of computations over communications. In the opposite case, the load balancing will not sensibly influence the performances and will have the drawback to overload the network. Another important point is the frequency of load balancing operations which must neither be too high (to avoid an overloading of the system) nor too low (to avoid too large an imbalance in the system). It is then important to design a good measure of the need to load balance, that is to say a measure which gives a quite precise idea of the unbalance of the system. It is important to perform just the right number of load balancings. Finally, the last point is the accuracy of the load balancing which depends on the network load. If the network is heavily loaded (or slow) it may be preferable to perform a coarse load balancing with less data migration. On the other hand, an accurate load balancing will tend to speed up the global convergence. The tricky work is then to find the good trade-off between those two constraints.

B. Residual vs classical load evaluation

In this section, we focus our attention on two different load estimators which can be chosen to perform load balancing in AIAC algorithms.

The most classical estimator consists in computing the processing time of a given amount of data (a component of the problem here) and then in distributing the data in order to obtain merely the same computation times for an iteration on all the processors in the system. Unfortunately, this definition of the load is not very efficient in our context of PIAs and more particularly of AIAC algorithms. This is due to the fact that in these algorithms, the critical point for efficiency is that all the processors reach local convergence merely at the same time. However, using this estimator only tends to equalize the processing times of each iteration on all the processors.

The load estimator we propose to use is thus the residual on each processor which allows to take into account the relative

progression of the processors during the iterative process. It may seem surprising to use the residual as a load estimator but this choice is very well adapted to this kind of computation as was briefly exposed in Section III. If a processor has a low residual, all its components are not evolving so far and its computations are not so useful for the overall progression of the algorithm. Hence, it can receive more components to treat in order to potentially increase its usefulness and also to allow its neighbor to progress faster.

In this context, several variants could be used. For example, we could use one residual per data and move those data independently. However, this strategy may break the data order and then the dependencies between processors too, leading to a sharp increase of the number of communications and to a modification of the communication graph (all-to-all in the worst case). Thus, our choice is to maintain the logical organization of the data in order to always use the same communication graph. Hence, each processor only has one residual which consists of the maximum norm of all its local components. Moreover, the data are moved while maintaining their global logical organization.

Since the residual on each processor is computed for all the components on this processor, it is not possible to precisely identify data which have the highest residuals. Thus, during a load balancing, nothing can ensure us that the moved data are actually the ones with the highest residuals. Nevertheless, our method has the great advantage of enhancing the progression in all cases. If the moved data have a high residual, then the processor with a lower residual will become in charge of a larger residual as initially expected. If the moved data do not have the highest residual (see Figure 8), then the residual of the receiving processor may not change much but the processor which has sent the data will have fewer components to manage. So, it will perform its iterations faster and then will tend to evolve faster, its residual will decrease faster and it will eventually overtake its initial lag.

Moreover, although in most cases the error will not follow a monotonous decrease, convergence conditions ensure us that it will globally decrease during the entire process until reaching the desired accuracy. Hence, this estimator actually gives a good indication of the distribution of the remaining work in the iterative process.

This scheme is exhibited in Figure 8 where a load transfer takes place between two processors according to their relative residuals. This figure is a detailed version of Figure 6 with computational errors of the components and residuals exhibited. The grey rectangles on both sides of the figure represent the flow of consecutive iterations on each processor with their length proportional to their duration. At the first iteration shown at the top of the figure, both processors have merely the same amount of data but their residuals are sufficiently different to detect the necessity of a load balancing. Hence, at the following iteration, the left processor sends a given part of its components to its right neighbor. We recall that the transfered components are always chosen to maintain the global order of the components which, in turn, maintains the global organization of the processors and thus the communication graph between them. Once the load transfer is completed, the

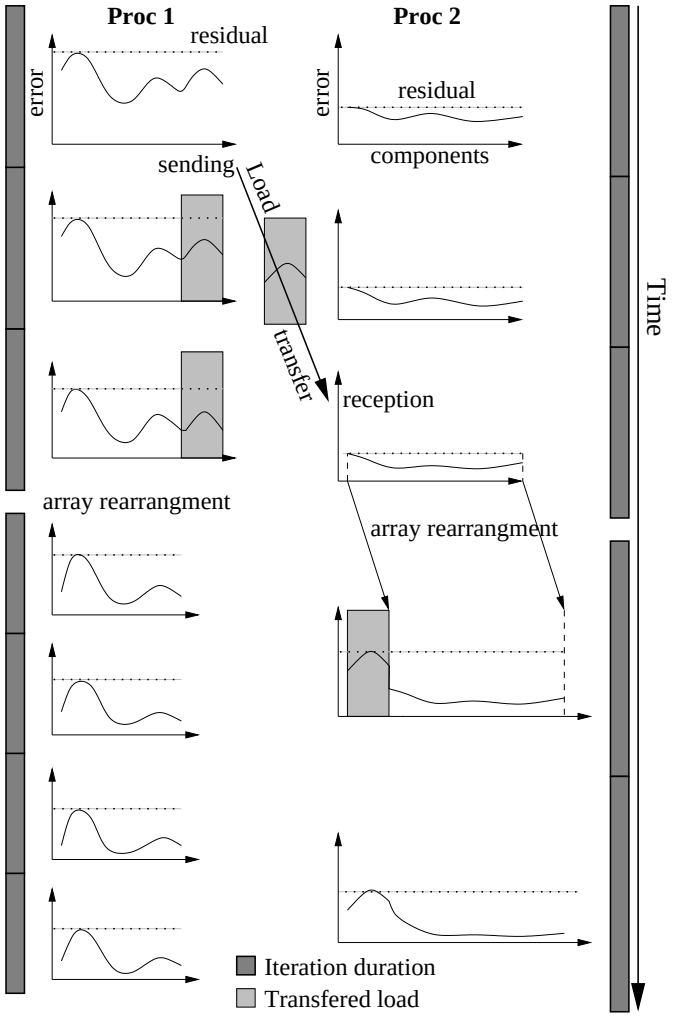

Fig. 8. Load transfer between two processors based on the residual

array rearrangements can be performed on both processors at the end of their current iteration. Then, both processors continue their computations with the new load distribution. A small break can be seen in the error curve of the right processor after the rearrangement. This is due to the fact that during the load transfer, components already on the right processor continue to evolve and their residual decreases whereas it is obviously not the case for the transferred components whose error stays the same as at their sending time. After the balancing, the left processor computes its iterations faster since it has fewer components whereas the right processor is slowed down by its increased number of components. Hence, the left processor tends to catch up with the right one and after some iterations, both processors have approximately the same residuals. The process can be repeated as soon as the difference between the residuals becomes too large again.

To show the impact of the load estimator on the overall performances of the algorithm, we have compared the execution times of our load-balanced AIAC with those two estimators in function of the final accuracy. This experiment has been realized using ten heterogeneous machines (from a PII 450Mhz to a PIV 2.4Ghz) scattered over three geographical

sites (Belfort, Besançon and Montbéliard). The results are given in Figure 9.

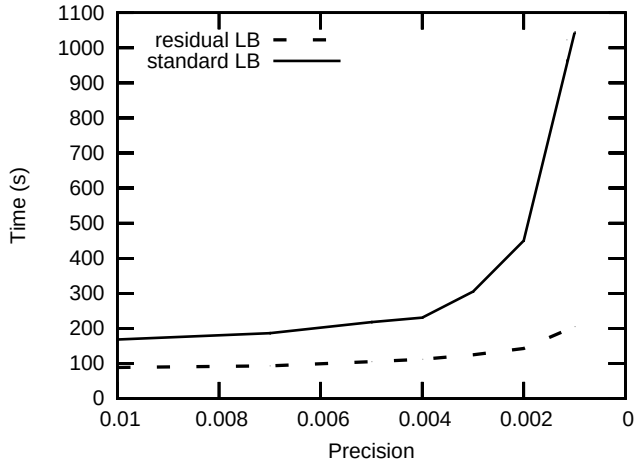

Fig. 9. Execution times (in seconds) with two different load estimators

Obviously, it can be seen that using the residual leads to far better performances especially for higher accuracies where a larger number of iterations is necessary. This comes from the fact that with the classical load estimator, the probability to have some processors which do not perform actually useful iterations drastically increases with the number of iterations. This is not the case with the residual with which there is a continuous attention to have all processors performing useful iterations.

VIII. Conclusion

The general interest of load balancing parallel iterative algorithms has been discussed and its major efficiency in the context of grid computing has been experimentally shown.

A comparison has been presented between a non-balanced and a balanced asynchronous iterative algorithm. The complete load balancing scheme has been detailed. Experiments have been done on a PDE problem using the PM2 multi-threaded environment. It has been tested in two representative contexts of grid computing. The first one is a local homogeneous cluster and the second one corresponds to a global context of grid computing.

The results of these experiments clearly show that the coupling of load balancing and asynchronism is fully justified since it gives far better performances than asynchronism alone which is itself better than synchronous algorithms. The efficiency of this coupling comes from the fact that those two techniques individually optimize two different aspects of parallel iterative algorithms. Asynchronism brings a natural and automatic overlapping of communications with computations and load balancing, as its name implies, provides a better distribution of the work over the processors. Moreover, the advantage induced by the non-centralized nature of the balancing technique has also been pointed out. Avoiding global synchronizations leads to less overheads and thus to a better scalability.

Finally, the required conditions for an efficient use of this coupling have been discussed as well as the choice of an efficient load estimator. Among the most interesting possibilities, the residual seems to be the most convenient load estimator as it does not try to balance the computation time at the level of one iteration but at the global level of the complete set of iterations needed in the process. Hence, it tends to make all the processors reach local convergence at the same time. Moreover, although we have obtained good results with our residual based load balancing algorithm, we think that some optimizations could be brought leading to yet better performances. This will probably be the subject of a future work.

In conclusion, a residual based load balancing whose frequency is tuned in function of the network speed is recommended in AIAC algorithms to obtain the best performances in both local and global contexts of grid computing.

References

[1] 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)*, Moncton, Canada, June 2002, pp. 90–97.

[2] J. M. Bahi, K. Rhofir, and J.-C. Miellou, "Parallel solution of linear DAEs by multisplitting waveform relaxation methods," *Linear Algebra and Its Applications*, vol. 3, no. 332–334, pp. 181–196, 2001.

[3] D. E. Baz, P. Spiteri, J. C. Miellou, and D. Gazen, "Asynchronous iterative algorithms with flexible communication for nonlinear network flow problems," *Journal of Parallel and Distributed Computing*, vol. 38, no. 1, pp. 1–15, 10 Oct. 1996.

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

[5] E. Hairer and G. Wanner, *Solving ordinary differential equations II: Stiff and differential-algebraic problems*, ser. Springer series in computational mathematics. Berlin: Springer-Verlag, 1991, vol. 14, pp. 5–8.

[6] Y. Saad, *Iterative Methods for Sparse Linear Systems*. New York: PWS Publishing, 1996.

[7] D. Szyld, "Perspectives on asynchronous computations for fluid flow problems," Department of Mathematics, Temple University, Tech. Rep., 2000. [Online]. Available: http://www.math.temple.edu/∼szyld

[8] J. Bahi, S. Contassot-Vivier, and R. Couturier, "On the interest of load balancing asynchronous parallel iterative algorithms," LIFC, AND Team, Tech. Rep., 2003.

[9] A. Frommer and D. Szyld, "On asynchronous iterations," *J. of computational and applied mathematics*, vol. 23, pp. 201–216, 2000.

[10] R. S. Varga, "Matrix iterative analysis," *Prentice-Hall*, 1962.

[11] A. Herz and C. Marcus, "Distributed dynamics in neural networks," *Physical Review E*, vol. 47, no. 3, pp. 2155–2161, 1993.

[12] M. E. Tarazi, "Some convergence results for asynchronous algorithms," *Numer. Math.*, vol. 39, pp. 325–340, 1982.

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

[14] J. C. Miellou, D. E. Baz, and P. Spitéri, "A new class of asynchronous iterative algorithms with order intervals," *Math. of computation*, vol. 221, no. 67, pp. 237–255, 1998.

[15] J. M. Bahi, "Asynchronous iterative algorithms for nonexpansive linear systems," *Journal of Parallel and Distributed Computing*, vol. 60, no. 1, pp. 92–112, Jan. 2000.

[16] C. Xu and F. Lau, *Load Balancing in Parallel Computers: Theory and Practice*, 1996.

[17] S. H. Hosseini, B. Litow, M. Malkawi, J. McPherson, and K. Vairavan, "Analysis of a graph coloring based distributed load balancing algorithm," *Journal of Parallel and Distributed Computing*, vol. 10, no. 2, pp. 160–166, Oct. 1990.

[18] G. Cybenko, "Dynamic load balancing for distributed memory multiprocessors," *Journal of Parallel and Distributed Computing*, vol. 7, no. 2, pp. 279–301, Oct. 1989.

[19] B. Ghosh, F. T. Leighton, B. M. Maggs, S. Muthukrishnan, C. G. Plaxton, R. Rajaraman, A. W. Richa, R. E. Tarjan, and D. Zuckerman, "Tight analyses of two local load balancing algorithms," in *Proceedings of the Twenty-Seventh Annual ACM Symposium on the Theory of Computing*, Las Vegas, Nevada, 29 May–1 June 1995, pp. 548–558.
[20] G. Karagiorgos and N. M. Missirlis, "Accelerated diffusion algorithms for dynamic load balancing," *Information Processing Letters*, vol. 84, no. 2, pp. 61–67, Oct. 2002.
[21] H. Rim, J.-W. Jang, and S. Kim, "An efficient dynamic load balancing using the dimension exchange method for balancing of quantized loads on hypercube multiprocessors," in *IPPS/SPDP 1999*, 1999, pp. 708–713.
[22] R. Elsasser, B. Monien, and R. Preis, "Diffusion schemes for load balancing on heterogeneous networks," *Theory of Computing Systems*, vol. 35, pp. 305–320, 2002.
[23] C. W. Gear, "Massive parallelism across space in ODEs," *Applied Numerical Mathematics: Transactions of IMACS*, vol. 11, no. 1–3, pp. 27–43, Jan. 1993, parallel methods for ordinary differential equations (Grado, 1991).
[24] R. E. White, "Multisplitting of a symmetric positive definite matrix," *SIAM Journal on Matrix Analysis and Applications*, vol. 11, pp. 69–82, 1990.
[25] 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*, vol. 11. Elsevier, North-Holland, 1996, pp. 279–285.

Jacques M. Bahi is a full professor of computer science at the university of Franche-Comté (France). He received a PhD and a HDR in applied mathematics at the same university. His research interests include asynchronism, distributed computing, load balancing and massive parallelism. He is a member of IEEE and the IEEE Computer Society.

Sylvain Contassot-Vivier received the PhD degree from École Normale Supérieure de Lyon in 1998. He joined the image processing group of the ERIC Laboratory at the University Lyon 2 in 1997. He obtained a postdoctoral position at the FUNDP of Namur (Belgium) in 1998. Since 1999, he is an Assistant Professor in the Computer Science Laboratory LIFC at the University of Franche-Comté (France). His main research interests lie in the parallel computing domain such as massively parallel systems, asynchronism, grid-computing and also in the domain of image processing and vision. He is a member of IEEE and the IEEE Computer Society.

Raphaël Couturier received the PhD degree in computer science from Henri Poincaré University, Nancy, France, in January 2000. In September 2000, he joined the computer science laboratory of the University of Franche-Comté where he is an assistant professor. His research interests include parallel and distributed computation, numerical algorithms and data mining. He is a member of the IEEE and the IEEE Computer society.