Communication Study and Implementation Analysis of Parallel Asynchronous Iterative Algorithms on Message Passing Architectures

D. El Baz

LAAS-CNRS, 7, avenue du Colonel Roche, 31077 Toulouse CEDEX 4, France,
E-mail: elbaz@laas.fr

Abstract

The implementation of parallel asynchronous iterative algorithms on message passing architectures is considered. Several issues related to communication via message passing interfaces or libraries such as MPI-1, MPI-2, PVM or SHMEM are discussed in this survey paper. Practical implementations are proposed.

1 Introduction

In the recent years, the concept of asynchronism has gained a considerable amount of attention in many domains related to computer science e. g.: circuits, processes and algorithms. In particular, parallel asynchronous iterative methods have been studied extensively (see for example [2], [3], [7], [9], [11], [13], [14], [18] and [20]). Asynchronous iterative algorithms were shown to be more efficient then their synchronous counterpart for many applications including optimization (see [9] and [13]) and numerical simulation (see for example: [18]). Today, the features of asynchronous algorithms such as: lack of synchronization, tolerance to problem data changes, fault tolerance, and flexibility make them very attractive for Grid computing, global computing and peer to peer computing. Nevertheless, asynchronous iterative algorithms remain difficult to apprehend by a large number of computer scientists mainly because several points related to the implementation of this class of methods have not yet been totally clarified. Authors who have contributed to this field generally present specific implementations of these algorithms without drawing general lessons on the implementation issue. As a result, there is no global view on the implementation topic and parallel asynchronous iterative algorithms are often considered as not easy to understand and not easy to implement methods. This issue seems particularly complex for message passing architectures, since message passing interfaces such as MPI have functionalities that may be implementation dependant.

This paper deals with the implementation of parallel asynchronous iterative algorithms on message passing architectures. We concentrate on communication aspects. The main goal of this paper is to clarify the implementation issue. We show in particular that asynchronous iterations can be implemented easily by decoupling merely computation processes. We show also that it is not mandatory to carry out specific types of communication, such as asynchronous communications or nonblocking communications, in order to implement an asynchronous iterative algorithm. We consider message passing interfaces or libraries such as MPI-1 and MPI-2 and illustrate our topic by presenting, in each case, efficient ways to implement asynchronous iterations.

Section 2 deals with parallel asynchronous iterations; various models such as classical asynchronous schemes and extensions e.g. flexible asynchronous iterations are displayed. The fundamental principles relevant to the implementation of asynchronous iterations and communication issues are discussed in Section 3. Section 4 proposes several implementations using MPI1 and MPI2.

2 Asynchronous iterative algorithms

In this section, we propose a presentation of parallel asynchronous iterative algorithms which concentrates on the basic properties and eludes cumbersome mathematical models. Classical asynchronous iterations (AI) (see [12], [1], [3] and [17]) are first considered, then, the section deals with a recent extension: flexible asynchronous iterations (FAI) (see [13] and [18]).

2.1 Classical asynchronous schemes

AI algorithms are generally used in order to compute an approximate solution to fixed point problems that can be written as follows.

$$x^\star = F(x^\star), \qquad (1)$$

where $x^\star$ is a vector in R^n and F is a given fixed point mapping from R^n into R^n. The space R^n is partitioned

into p subspaces, where p is the number of processors; the fixed point mapping and iterate vector are partitioned accordingly, i.e. $x = \{x(1), ..., x(p)\}$, where $x(i)$ is the so-called i-th block-component of the iterate vector x. Block-component $x(i)$ is assigned for example to the i-th processor. AI generate successive approximations of the block-components of x. These approximations are obtained by applying in parallel and repetitively the block-components of the fixed point mapping $F(i)$, $i = 1, ..., p$, to a given initial approximation x^0. AI, which have also been called totally asynchronous iterations (see [3]) or chaotic iterations in the bounded delay context (see [4]) have been designed for a large number of applications including solution of discretized partial differential equations, optimization and optimal control, (see for example [8], [16] and [17]); their convergence has been studied in different contexts such as contraction and partial ordering (see [2], [7], [9], [14] and [17]). A simple definition of AI can be given as follows.

Definition 1 *AI are successive approximation algorithms whereby components or block-components of the iterate vector are updated in parallel without any order nor synchronization.*

The restrictions imposed to AI are very weak: no block-component (or component) of the iterate vector is abandoned forever and more and more recent updates of the components have to be used as the computation progresses. The advantages of AI are computation flexibility, tolerance to problem data changes (the algorithm adapts itself to a modified environment) and fault tolerance (the algorithm can work well even if some data are lost or some processors fail). Since there is no synchronization overhead or idle time due to synchronization, one may also hope that AI will be more efficient than their synchronous counterpart. This last remark is particularly true in the partial ordering context where monotone sequences of updates are generated. We
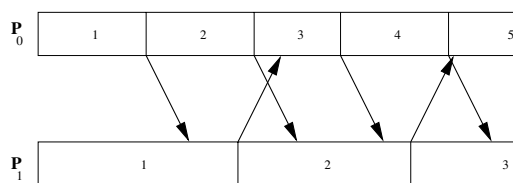

Figure 1. Asynchronous iterative algorithm

present now some simple illustrations of parallel iterative algorithms. Figure 1 displays the behavior of a typical AI in the simple case where two processors denoted by P_0 and P_1 cooperate to the same application, i.e. the solution of a given fixed point problem. Processors's updating phases are represented by boxes (the number in each box denotes the update number) and communications of updates by bold

arrows. More precisely, an arrow delimits two events occurring during a communication between two processors. The first event, which corresponds to the beginning of the arrow, is relevant to posted data (update) send at the source. The second event, which corresponds to the end of the arrow, is related to data arrival at the receiver. We note that the actual completion of a communication may occur long after the time when the new data is actually available at the receiver. We have chosen to illustrate here an important feature of AI, i.e. the possibility to overlap communication by computation. Figure 1 shows also clearly that there is no idle time in AI.

By way of comparison, Figure 2 displays a typical synchronous iterative algorithm, i.e. the parallel Jacobi scheme, $x^{k+1} = F(x^k)$, in the simple case where two processors cooperate to the solution of the same fixed point problem. A dashed box delineates here the combination of a communication phase and a synchronisation phase. More precisely, a dashed box delimits the interval of time between the beginning of a communication and the completion of a synchronisation barrier. The completion of the communication occurs clearly during this interval of time.
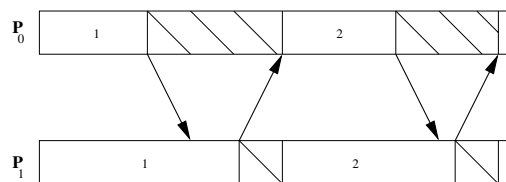

Figure 2. Synchronous iterative algorithm

2.2 Flexible asynchronous iterations

We present now an extension of AI i.e. flexible asynchronous iterations (FAI). For more details on FAI, the reader is referred to [11] to [13], see also [18].

Definition 2 *FAI are iterative algorithms whereby components or block-components of the iterate vector are updated in parallel without any order nor synchronization using the current value (which is not necessarily labelled by an update number) of each component of the iterate vector.*

Thus, FAI are algorithms whereby iterations are also carried out in parallel in arbitrary order and without any synchronization. Aside from lack of synchronization and flexibility in the order of steering components, we have also flexibility in the use of data produced by the algorithm. Indeed, the value of any component of the iterate vector used during an updating phase can correspond to the curent value of this component which is not necessarily labelled by an update

COMPUTER SOCIETY

number. This is typically the case in the inner/outer context (see [14] and [12]) where the global iteration function is such that an inner iteration has to be performed in order to approximate the value of the outer iteration function at some point. This iterative scheme allows intermediate results from the inner iteration to be used in the computations. Updating phases can also use values of components of the same block-component which are relevant to different update numbers, i.e. some components have been already updated while others not, like in the solution of a subsystem of equations with a triangular matrix; this case corresponds to the so-called partial update situation. This feature permits one to take into account data coming out from computations which are in progress; it is particularly interesting in the partial ordering context where monotonically increasing or decreasing sequences of vectors are generated iteratively. So, we may expect better performance. We note that numerical simulation have confirmed this expectation (see [13] and [18]). FAI have also been called asynchronous iterative algorihms with order intervals and asynchronous iterations with flexible communication. Convergence results for FAI have been established in different contexts: partial ordering (see [13] and [18]) and contraction (see [12]).

Figure 3 shows the behavior of a typical FAI in the simple case where two processors cooperate to the same application. Communication of updates and partial updates, respectively, are represented by bold arrows and normal arrows, respectively. Finally, we note that FAI do not necessarily lead to an important augmentation in the number of data exchanges, as Figure 3 may suggest, but rather, gives the possibility to obtain the current value of any component of the iterate vector when needed. Basically, the number of data exchanges will depend on the data exchange policy which is actually implemented.
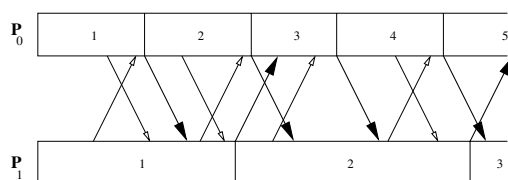

Figure 3. Flexible asynchronous iteration

3   Principles of implementation

In this Section, we bring forward the general lessons that permit one to implement AI and FAI on a message passing architecture. In particular, we point out that the implementation of parallel algorithms is not specifically related to the use of a given type of communication such as nonblocking or asynchronous communications. Merely, we bring into

evidence that the type of communication chosen must not be an obstacle to computation progress, i.e. it must allow each processor to go at his own pace. Thus, the key point of any efficient implementation will be the preservation of this important feature.

3.1   Asynchronous iterative algorithms

We state now the main principle relevant to AI implementation. In the sequel, all implementations will be derived from this basic statement. For the sake of simplicity, we will assume in what follows that updating phases are implemented according to the Single Process Multiple Data (SPMD) model using processes called computation processes and that there is only one computation process per processor.

Principle 1 In order to implement AI, one has merely to insure that the beginning of each updating phase of any processor must not be subordinated to a data exchange and that updating phases must be chained on each processor.

Remark 1 *It follows that processors will perform updating phases, i.e. iterations, at their own pace using the available updates (since no data exchange is mandatory in order to execute a new computation).*

Remark 2 *It is very important to make a clear distinction between parallel iterative schemes of computation on the one hand and available types of communication that can be used in order to implement them on the other hand. For example, one must not make a confusion between asynchronous iterative algorithms and asynchronous communications in MPI-1 as we shall see in Section 4. One must simply choose a type of communication which is in agreement with the chosen implementation of the parallel algorithm. For example, one must organize computations and communications so that the latter do not block the former. As a consequence, all direct communication operations between two computation processes based on rendez-vous semantics must be banished a priori. It is also clear that all computation processes must not wait here for the completion of any communication; so that, computation processes will not be synchronized by any means. It is also important to note that there are simple ways to desynchronize computation processes Several mechanisms can be implemented in order to carry out AI from the mere dire use of nonblocking communications between computation processes to the design of specific processes that handle communications or computations on each processor (see [8]). In the later case, only communication processes communicate thus, computation processes running on the distributed memory architecture are naturally desynchronized; as a consequence, we can make use of any type of communication between communication processes; even rendez-vous techniques can be used.*

3.2 Flexible asynchronous iterations

Principle 2 In order to implement FAI, one has on the one hand to insure that the beginning of any updating phase of a given processor must not be subordinated to a data exchange and on the other hand to insure that all processors can have access, when needed, to the current value of the components of the iterate vector which are updated by other processors.

Remark 3 *It follows that each processor performs updating phases at his own pace, using the current value of each component of the iterate vector which is not necessarily labelled by a given iteration number.*

Remark 4 *Very few things are supposed on the way data exchanges are performed in the flexible asynchronous context. It is only assumed that the current value of the components of the iterate vector can be obtained when needed.*

It follows from Principle 2 and the above remark that there are several ways to implement flexible asynchronous iterative algorithms. One can, on the one hand, perform a data acquisition of the current value of the components of the iterate vector. We will see in detail, in the sequel, that we can use for this purpose Remote Memory Access (RMA) functions such as, for example, the MPI-2 function MPI_GET(). On the other hand, one can also decide that each process will send the current value of its assigned components of the iterate vector according to a given policy; this is typically the case in the inner/outer context (see [13]). Flexible asynchronous iterations allow intermediate results from the inner iteration in a given processor to be used by other processors. This case can be found typically in twostage iterative methods i.e. iterative methods with two embedded loops (see [13]); for example, intermediate values can be sent every q steps of the interior loop. This case is relevant to the situation illustrated by Figure 3. We will see, in the sequel, that we can use for this purpose RMA functions such as the MPI-2 function MPI_PUT(). If we implement an inner/outer iterative methods (see [13]), then, data aquisition or request for new data, can be made directly in the inner iteration loop when an accuracy very close to the requested accuracy is reached. This anticipation mechanism permits one to get new data available just before the very beginning of a new updating phase.

4 Examples of implementation

4.1 MPI-1 implementation

We illustrate now the basic principles of implementation of parallel algorithms via the Message Passing Interface (MPI-1) library (see [19]). The MPI-1 interface permits one to use a network of processors of a given parallel architecture as a unique resource of calculus. MPI-1 and the new MPI-2 are available on all main machines such as IBM SP series and various clusters. MPI-1 is well suited to massive parallelism. In this situation, it is expected to be faster than Parallel Virtual Machine (PVM). We note also that MPI-1 has more communication options than PVM. This last point is particularly important for the implementation of asynchronous iterative algorithms.

Asynchronous iterative algorithms

An elegant way to implement AI is to use point to point communications between computation processes. More precisely, on the one hand, nonblocking send can be used, namely the MPI_ISEND() posting send operation; the arguments of MPI_ISEND() being the address of data and their size. On the other hand, messages can be received by using persistent communication request since communication with the same argument list is repeatedly executed. Persistent communication request can be thought of as a communication port or a half-channel; the construct allows reduction of the communication overhead between the process and communication controller. We note that each call of a MPI_RECV() operation implies an additional latency time. When there are several occurences of messages in the receive buffer, it is more efficient to use persistent communication since the receive process is then performed only once using MPI_RECV_INIT(). Initialisation and activation, respectively, of the reception are made with MPI_RECV_INIT() and MPI_START(), respectively. Message detection in the buffer is made with the MPI_TEST() test function. The following parts of Fortran code show how nonblocking send and persistent communications can be implemented.

```
C       Initialisation of the reception
        CALL MPI_RECV_INIT(DATA,...)
        CALL MPI_START(REQ_P(SOURCE),...)
C       In the COMPUTATION()  process:
        SUBROUTINE COMPUTATION()
C       Variables
        ......
  10    CONTINUE
C       Reception test of data
        CALL MPI_TEST(REQ_P(SOURCE),...)
        WHILE (LOG_P)
          CALL MPI_START(REQ_P(SOURCE),.)
          CALL MPI_TEST(REQ_P(SOURCE),..)
C       Computation of a new update
        CALL UPDATING()
C       Send the data to other processes
        CALL MPI_ISEND(DATA,...)
C       Stopping test - >  test
        IF(test.GT.EPSILON) GOTO 10
        RETURN
        END
```

We note that neither the computation process which transmits, nor the computation process which receives are waiting.

Remark 5 *If no reception loop was implemented, then messages could be received long after their emission and the implementation might be inefficient. The reception loop using MPI_TEST() and MPI_START() permits the processor to take into account new messages which have been received before beginning a new updating phase. This implementation is more performant, particularly in the case where some computation processes are updating components more rapidly than others.*

Persistant communication can also be implemented with the MPI_IPROBE() function which is a nonblocking operation that returns flag = true if there is a message that can be received and that matches the message envelope specified by source, tag and com. The receive process is activated only one time with the MPI_INIT_RECV() function and the buffer is read via the MPI_START() function. Message detection in the buffer is made with the MPI_IPROBE() function. The Implementation is as follows.

```
MPI_INIT_RECV(list->buf_recv,...)
MPI_DOUBLE, list->num,...)
MPI_IPROBE(list->num,...)
WHILE(flag1)
  MPI_START(&request)
  MPI_IPROBE(list->num,...)
```

This solution is very attactive because if no message can be received, then no reception is performed and the process does not wait. Reception is also performed while messages can be received. So, processes have always access to recent updates, since MPI preserves the order of emission.

Remark 6 *The above solutions are better than the one which consists in using the simple nonblocking posting send and receive operations MPI_ISEND() and MPI_IRECV(), respectively, of the MPI-1 library since data may then be received and used long after they have been sent.*

Remark 7 *The program may write in the space memory buffer just after a send. Thus, we could also use MPI_IBSEND(), which is a bufferized nonblocking posting send in order to respect MPI safe programming rules. However, we point out that the use of MPI_ISEND() or MPI_IBSEND() operations without using completion of communications goes against the same safe programming rules of MPI. In fact, it is not false to say that the implementation of asynchronous iterative algorithms does not correspond exactly to the spirit of MPI. Another solution could consist in testing if all the send operations have been performed and then cancel via a CANCEL() those operations who have not been made; however, the use CANCEL() is not recommended since its overhead is nonnegligeable.*

Flexible asynchronous iterative algorithms

Flexible asynchronous iterations can be implemented in a way quite similar to the one presented in the very beginning of this subsection. The only difference is that the MPI_ISEND() posting send operation is now used to send intermediate values or partial updates. For example, these data can correspond to values of the iterate vector delivered in the inner loop of a inner/outer iterative algorithm. Thus, the MPI_ISEND() posting send operation must be placed now in the inner loop. An example of implementation in Fortran is shown below.

```
      SUBROUTINE COMPUTATION(...)
C     Variables
      ......
 10   CONTINUE
C     Reception test of data
      CALL MPI_TEST(REQ_P(SOURCE),...)
      WHILE (LOG_P)
        CALL MPI_START(REQ_P(SOURCE),..)
        CALL MPI_TEST(REQ_P(SOURCE),...)
C     Updating
      ...
C     Beginning of the inner loop
      WHILE condition not satisfied
        update the iterate vector
C       Send current value to processes
        CALL MPI_ISEND(CURRENT,...)
C     Send update to processes
      CALL MPI_ISEND(DATA,...)
C     Stopping test - >  test
      IF(test.GT.EPSILON) GOTO 10
      RETURN
      END
```

In [15] and [16], we have proposed an extension of the above approach for carrying out flexible asynchronous iterative algorithms on clusters of symetric multiprocessors (clusters of SMP). This extension combines the library MPICH-gmm (Myrinet) with multithreading aspects. For details on the implementation reference is made to [15]. The above implementation can also be extended to meta-computing or grid computing via MPICH-G an MPI version adapted to Globus.

4.2 MPI-2 implementation

We illustrate now the basic principles of AI and FAI implementation in the MPI-2 context. An important feature of MPI-2 is remote memory access (RMA) via one sided communications (OSC). This feature is a major extension of the communication model of MPI. This type of communication is particularly well suited to distributed memory programming model and permits one to write with the

MPI_PUT() function or read via the MPI_GET() function directly in the memory of a distant target processor. The arguments of the functions are: the identity of the target, the addresses of the data in the source and in the target, the size of the data. In this case, the target processor does not interfere in data transmission. OSC are nonblocking and permit the programmer to uncouple, i.e. desynchronize, naturally computation processes. Moreover, this type of communication may be more performant on machines which have material support for shared memory operations. Among major features of MPI-2, we can quote also the dynamic control of processes, i.e. the possibility to create or suppress processes during the application. We note that a call to the MPI_GET() nonblocking function does not guarantee that the data will be actually available for upcoming computations. In fact, with OSC, we don't know exactly when a data reception is complete and when data are really available. Completion of data exchange could be obtained via a synchronization phase which is not really desirable in the general context of our study. Synchronization could be done, for example, via an active target operation, whereby all processes communicating throughout the same memory window participate to the synchronization with a MPI_WIN_FENCE() operation. Another possibility, is passive target synchronization, whereby the process which is at the origin of data exchange calls the functions of the synchronization subprogram via MPI_WIN_LOCK() and MPI_WIN_UNLOCK(). This approach is one of the most interesting feature of MPI-2. All the necessary calls for data transfer: transmission, initialization and synchronization need only the origin process, this is the feature of an actual OSC. We note that the implementation of asynchronous iterations departs in some sense to the RMA rules of good use since we can ignore the synchronization phase.

Asynchronous iterative algorithms

Asynchronous iterations can simply be implemented via the MPI_PUT() function. The following algorithm, where the variable $nbtarp(i)$ represents the number of target processors for processor P_i and $P(i, j)$ the identity of the jth-target processor for P_i, gives an illustration of a simple implementation.

```
Do in parallel i = 1, ..., p
  Until convergence
    compute a new update of x(i)
    Do in parallel j = 1, ..., nbtarp(i)
      MPI_PUT(x(i),P(i,j),size)
    End do
End do
```

Here data are transferred via RMA, thus, each processor reads directly in his local memory the data required for its computations. This straightforward phase is skipped in the above algorithm. This solution is simple and permits one to overlap naturally communication by computation.

Flexible asynchronous iterative algorithms

Flexible asynchronous iterations can be implemented easily with the MPI_GET() function that reads the current value of a block-component of the iterate vector. The following algorithm displays how easily flexible asynchronous iterations can be implemented using MPI_GET().

```
Do in parallel i = 1, ..., p
  Until convergence
    Do in parallel j = 1, ..., nbtarp(i)
      MPI_GET(DATA(j),P(i,j),size)
    End do
    compute a new update of x(i)
End do
```

This solution presents the advantage to require less data exchanges than the one using the MPI_PUT() function, which consists in writing directly in the memory of a distant target processor from time to time the curent value of the block-components of the iterate vector, as in the inner loop of a inner/outer iterative algorithm; it is also more natural and performant (see [11] and [13]). We note that the afore-mentioned two solutions are typical of the emission/reception duality.

5 Conclusions

In this paper, we have tried to clarify issues related to the implementation of parallel asynchronous iterative algorithms in message passing architectures. In particular, we have presented the main principles upon which must be based implementations of parallel aynchronous iterative algorithms and flexible parallel aynchronous iterative algorithms. We have also dealt with the important role of communications in the implementation issue. Finally, we have presented several implementations in the case of MPI-1, MPI-2.

We have seen that it is always possible to implement easily asynchronous iterative algorithms and flexible asynchronous iterative algorithms with the above quoted libraries. However, it is important to note that efficient implementation of asynchronous iterative algorithm relies on important features of the libraries such as the possibility to superpose new data on old one or the existence of a wide variety of communication protocols such as nonblocking data transfer or RMA.

One important question remains, when one studies the

performance of a parallel algorithm: what part of efficiency is due to the algorithm on the one hand and what part is due to the communication library, its implementation and finally to the system, on the other hand. In future work, we plan to deal with these points.

References

[1] G. M. Baudet, *Asynchronous iterative methods for multiprocessors*, J. Assoc. Comput. Mach., 2 (1978), 226-244.

[2] D. P. Bertsekas and D. El Baz, *Distributed asynchronous relaxation methods for convex network flow problems*, SIAM J. on Control and Optimization, 25 (1987), 74-85.

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

[4] D. Chazan and W. Miranker, *Chaotic relaxation,* Linear Algebra Appl., 2 (1969), 199-222.

[5] D. Conforti, L. Grandinetti, R. Musmano, M. Cannataro, G. Spezzano and D. Talia, *A model of efficient asynchronous parallel algorithms on multicomputer systems,* Parallel Computing, 18 (1992), 31-45.

[6] D. El Baz, *Mise en œuvre d'algorithmes itératifs distribués asynchrones sur un réseau de transputers*, Calculateurs Parallèles, Réseaux et Systèmes Répartis, 3 (1989), 31-40.

[7] D. El Baz, *M-functions and parallel asynchronous algorithms*, SIAM Journal on Numerical Analysis, 27 (1990), 136-140.

[8] D. El Baz, *Asynchronous implementation of relaxation and gradient algorithms for convex network flow problems*, Parallel Computing, 19 (1993), 1019-1028.

[9] D. El Baz, *Asynchronous gradient algorithms for a class of convex separable network flow problems*, Computational Optimization and Applications, 5 (1996), 187-205.

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

[11] D. El Baz, *Contribution à l'algorithmique parallèle le concept d'asynchronisme : étude théorique, mise en œuvre, et application*, Habilitation à Diriger des Recherches de l'Institut National Polytechnique de Toulouse, 6 Octobre 1998, LAAS report 98428.

[12] D. El Baz, A. Frommer and P. Spiteri, *Asynchronous iterations with flexible communication: contracting operators*, Journal of Computational and Applied Mathematics, Vol. 176, Issue 1, April 2005, p. 91-103.

[13] D. El 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, 38, (1996), 1-15.

[14] A. Frommer and D. Szyld, *On asynchronous iterations*, Journal of Computational and Applied Mathematics, (2000), 777-783.

[15] M. Jarraya and El Baz, D., *A new implementation of asynchronous iterations with flexible communication on a network of symmetric multiprocessors*, Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications, Las Vegas, USA, 2, (2000), 777-783.

[16] M. Jarraya and El Baz, D, *Implementation of distributed iterative algorithm for optimal control problems on several parallel architectures*, Journal of Systems and Software, 60, (2002), 141-148.

[17] J. C. Miellou, *Algorithmes de relaxation chaotique à retards*, RAIRO, R1 (1975), 55-82.

[18] J.C. Miellou, D. El Baz, P. Spiteri, *A new class of asynchronous iterative algorithms with order intervals*, Mathematics of Computation, 67, 221, (1998), 237-255.

[19] M. Snir et al. *MPI: the Complete Reference*, The MIT Press, Cambridge, Mass. (1996).

[20] P. Spiteri, J.C. Miellou et D. El Baz, *Perturbation of parallel asynchronous linear iterations by floating point errors*, Electronic transactions on Numerical Analysis, 13, (2002), 38-55

IEEE
COMPUTER
SOCIETY