Asynchronous iterative algorithms on computational grid

St. Maruster

Institute e-Austria Timisoara

Abstract. The paper deals with asynchronous parallel iterative algorithms and with some of their applications to certain nonlinear problems, especially to systems of nonlinear equations. We analyse the properties of these methods such as convergence, rate of convergence, soundness, etc. Original preconditioners and a load balancing algorithm for this special case are suggested. Various numerical experiments and tests are also presented.

Keywords: Parallel computing, asynchronous algorithms, nonlinear systems.

1 Introduction

We consider the problem of using the asynchronous iterative algorithms to the solution of a large, sparse nonlinear problems. The focus of this paper is on asynchronous algorithms for systems of nonlinear equations (case study), algorithms that admit a high degree of parallelization. The block-Jacobi methods is taken under consideration and some particular implementations using classical serial algorithms, Newton, Gradient (Fridmann variant), conjugate gradient, quasi-Newton. The sparsity of the systems will plays an important roll in our development and it is motivated by a number of practical applications which have this characteristics. We point out some common problems that involve large, sparse nonlinear systems: ordinary differential equation (for example, implicit difference scheme for stiff equation), partial differential equations (different discretizations), but also nonlinear systems resulting directly from practical problems, such as steal building design (in domain of elasticity).

The main purpose of our work is the analysis of the block Jacobi algorithms from the programming view of point. Besides of this, we partly deal with some mathematical aspects of the parallel solving nonlinear systems, such that convergence, rate of convergence and soundness. More close to our general intention, are the following three subjects:

(1) Preconditioning of a large, sparse nonlinear systems;

(2) Stoping criteria;

(3) Load balancing and the possibility of using different algorithms in our case.

We briefly point out each of these issues.

Concerning the partitioning, we must emphasize that, due of the necessity of no lose solutions, the unique possibilities of processing the system are the reordering both variables and function components. As we will see below, such of preconditioning is always possible and it have a major influence of the computing performance.

The stoping criteria have the following meaning in our case. Every block algorithm considered here involves an outer iteration consisting in a **Repeat** cycle which must be performed until global convergence is achieved. Also there exists an inner iteration in which a serial algorithm is applied for every block. The standard pseudocod is as follow:

```
Repeat until convergence
    For $i \in \{1, 2, ..., m\}$ Do
        While some precision is achieve Do
            Serial algorithm applied to a specific block;
        Eventually setting the new block component;
    End for
    Eventually setting the new block component;
End repeat
```

The number m represents the dimension of the considered block.

The precision in the **While** cycle should be prescribed in such a way that extra-computation to be avoid. Notice that the problem ia quite similar with the stoping criteria problem in inexact Newton serial algorithm ([**?**])

Load balancing is important mostly when the number of equation is very large, much larger than the number of available processors. In this case the computing strategy, which involves the assignment of different blocks to each processor, may be done dynamically such that equally distributes the work among processors. The general idea is to check periodically the computing stage for every processor (for example, by checking the errors of the all blocks assigned to that processor) and modify the assignment by transferring some load (blocks) from the highly loaded processors to the less loaded ones.

The general aspects of synchronous and asynchronous algorithms for systems of nonlinear equations, such as: convergence, implementation, applications and so on, were considered by many authors; see the survey work of Bertsekas and Tsitsiklis [**?**] and the reviewer paper of Formmer and Szyld [**?**].

The performance of parallel algorithms, both in synchronous and asynchronous case, depends in a great extent of the serial method used for numerical solve the block equations. The Newton block method which uses the classical Newton method and its variant were intensively studied in the literature. For instance, in the recent paper [**?**] the conditions that guarantee the local convergence of asynchronous Newton block method are established and the connection with the sequential and synchronous Newton block method are considered. The synchronous Newton block method was studied earlier in [?]. The Newton method is very attractive, because of its high rate of convergence; on the other hand, this method involves the solution of a nonlinear system at every iteration step, that may be often a difficult task. In [?] the quasi-Newton method is used for solving the the partial systems. The overlapping block parallel Newton method was considered in [?]. A particular case of a mildly nonlinear system, resulting from a discretization of certain differential equation was treated in [?]. In the paper [?] the authors consider the inexact Newton method and the quasi-Newton method, both combined with a block iterative row-projection linear solvers of Cimmino-type.

2 General Mathematical models

Throughout this paper we consider the following system of nonlinear equations

$$F(x) = 0, \quad F : D \subseteq I\!R^N \to I\!R^N, \tag{1}$$

where $F = (f_1, ..., f_N)^T$ is a large, sparse nonlinear function. Recall that large and sparse function means that N is a big number, which means that we have a large number of equations and unknowns, but every equation depends of a small number of variables, less than 20 percent from total variables.

Usually, the equation (1) is solved by an iterative method that is described by a *iteration* function $G : I\!R^N \to I\!R^N$ which depend of F. A well known example is the Newton method described by the iteration function G of the form $G(x) := x - F'(x)^{-1}F(x)$.

Suppose $X_1, X_2, ..., X_n$ are subsets of Euclidean spaces $I\!R^{d_1}, ..., I\!R^{d_n}$ and let $X = X_1 \times X_2 \times ... \times X_n$ be the Cartesian product of these subsets. The s-tandard assumption is that $X = \Re^N$, that is $\sum_{i=1}^n Dim(X_i) = N$. The case $\sum_{i=1}^n Dim(X_i) < N$ will be trivial, since in this case we should have less equations and less unknown, but it is possible that $\sum_{i=1}^n Dim(X_i) > N$, which means that the subspaces X_i can have overlaps. Suppose also that $x \in X$ is conformally decomposed in n partitions (blocks) as

$$x = (x_1, ..., x_n),$$

where $x_i = (x_{i_1}, ..., x_{i_{d_i}})^T$ is an nd_i-dimensional vector. The vector x_i is called a block component. Let $g_i : X \to X_i$ be given functions and let $G : X \to X$ be the function defined by $G(x) = (f_1(x), ..., f_n(x))$. We look at G as an iteration function defining an iterative method for solving nonlinear systems.

A partition (a block component) is defined by the set of indices $S_i = (i_1, ..., i_{d_i})$. In term of the sets S_i the standard conditions are

$$\begin{array}{ll} (1) & \cup_{i=1}^n S_i = \{1, 2, ..., n\}, \\ (2) & S_i \cap S_{i+1} \neq \emptyset, \quad i = 1, ..., n-1. \end{array}$$

The first condition implies that it doesn't exists gaps between blocks, that is every simple component belongs at least one block component. The second condition means that the decomposition may have neighboring overlaps. Usually, we will suppose that a partition consists of a continuous sequence of indices, that is $i_{k+1} = i_k + 1$, $k = 1, ..., d_i - 1$.

In particular, we consider the trivial decomposition in which every partition (block component) have exact one element and then $S_i = (x_i)$ and x_i is interpreted as being the simple component of x.

An example of decomposition is:

$$\underbrace{x_1, x_2, x_3, x_4, x_5}_{x_1}\!\underbrace{x_6, x_7, x_8}_{x_2}, \underbrace{x_9, x_{10}, x_{11}, x_{12}}_{x_3}\!\!\!\!\!\!\!\!\underbrace{\phantom{x_{12},}x_{13}, x_{14}, x_{15}}_{x_4}$$

In this example, $S_1 = (1, 2, 3, 4, 5)$, $S_2 = (4, 5, 6, 7, 8)$, $S_3 = (9, 10, 11, 12)$ and $S_4 = (12, 13, 14, 15)$.

There are two main models of parallel algorithms for solving equation (1):

Block Jacobi model:

```
**Initialization:** $t = 0$, $x(0) = x0$;
**Repeat until convergence**
    **For** $i \in \{1, 2, ..., n\}$ **Do**
        Solve $f_i(x_1(t), ..., x_{i-1}(t), y_i, x_{i+1}(t), ..., x_n(t)) = 0$;
    **End For**
    **Set** $x(t + 1) := (y_1, ..., y_n)$;
    t:=t+1;
**End Repeat**
```

Remark.

The cycle **For** can be covered in arbitrary order, provided that the cycle index i will covers all values 1,2,...,n.

Block Gauss-Seidel model:

```
**Initialization:** $t = 0$, $x(0) = x0$;
**Repeat until convergence**
    **For** $i = 1, 2, ..., n$ **Do**
        Solve $f_i(x_1(t + 1), ..., x_{i-1}(t + 1), y, x_{i+1}(t), ..., x_n(t)) = 0$;
        **Set** $x_i(t + 1) := y$;
    **End For**
    t:=t+1;
**End Repeat**
```

The Gauss-Seidel algorithm generally converge faster then corresponding Jacobi algorithm, because the Gauss-Seidel algorithm incorporate the newest available computed components in the cycle **For**.

The both models involve as a main computation the solution of the system $f_i(x_1, ..., x_{i-1}, y, x_{i+1}, ..., x_n) = 0$, y being the unknown block component. In practice, this step can be done approximatively, using the iteration function $G : X \to X$, $G = (g_1, ..., g_n)$, and $g_i : X \to X_i$. Different type of such function can be used, the Newton type being the most common and known. Thus, the two models become:

Block Jacobi model:

```
**Initialization:** $t = 0$, $x(0) = x0$;
**Repeat until convergence**
    **For** $i \in \{1, 2, ..., n\}$ **Do**
        $y_i := g_i(x_1(t), ..., x_n(t))$
    **End For**
    **Set** $x(t + 1) := (y_1, ..., y_n)$;
    t:=t+1;
**End Repeat**
```

Remark.

We have used the index t for distinguish the new value of x. In fact, this index is actualized in the cycle **Repeat** and it can be viewed as the time of an external clock.

Block Gauss-Seidel model:

```
**Initialization:** $t = 0$, $x(0) = x0$;
**Repeat until convergence**
    **For** $i = 1, 2, ..., n$ **Do**
        $y = g_i(x_1(t + 1), ..., x_{i-1}(t + 1), x_i(t), x_{i+1}(t), ..., x_n(t)) = 0$;
        **Set** $x_i(t + 1) := y$;
    **End For**
    t:=t+1;
**End Repeat**
```

3 Synchronous Algorithms

The computation required by the two mathematical model are divided among P processors of a computing system and are performed in parallel. In general, we may suppose that each processor execute the computation required of the main step of the algorithm, the computation of $x_i(t + 1)$ for a given set of indices. Roughly speaking, if processors would wait each other to complete the computation in the cycle **Repeat**, that is all the computation are ready when the index t is updated, we would get a parallel synchronous algorithm. If processors do not wait for each other, we get an asynchronous implementation of the successive approximation given in the two models. In this case, each processor would perform its own number of iteration in **Repeat** loop, which may differs in a great extent from a processor to other, due of the characteristics of the generation function involved. More detailed aspects of concrete implementation of synchronous and asynchronous algorithms will be given below.

3.1 Synchronous implementation of block Jacobi model

If $P = n$ each processor P_i performs the computation $y_i = g_i(x(t))$. The cycle **For** is ready when each processor have finished its task and the value x at the time $t + 1$ is setting outside of the cycle. Note that we can always dispose of the decomposition such that $n = P$, but in the case of a very large system and a moderate number of processors, such a decomposition is not reasonable. Hence, in the case when $P < n$, we can implement the block Jacobi model by assigning to each processor P_k a subset of indices J_k such that $\cup_{k=1}^{P} J_k = \{1, ..., n\}$. In the cycle **For** each processor performs the computation $y_i = g_i(x(t))$, for $i \in J_k$ in a sequential mod. A particular choice of the subsets J_k gives the strategy in block Jacobi model. In this case, the the cycle **For** of the block Jacobi model for each processor P_i become:

```
For $i \in J_k$ (in sequential mod) Do
    $y_i := g_i(x_1(t), ..., x_n(t))$
End For
```

Remark. In the synchronous implementation, each processor k execute the computation corresponding to the set J_k and put the values $\{y_i\}$, $i \in J_k$ in a local (vector) variables Y_k. When all processor are ready, then $x(t + 1)$ take its new value, $x(t + 1) = (Y_1, ..., Y_p)$ (note that the components of each Y_k are not necessarily in a continuous sequence). Of course, the results of the computation are exactly as in a sequential implementation and so we can simulate the parallel processing by the programm:

```
Initialization: $t = 0$, $x(0) = x0$;
Repeat until convergence
    For $k = 1, ..., p$ Do
        For $i \in J_k$ (in sequential mod) Do
            $y_i := g_i(x_1(t), ..., x_n(t))$
        End For
        For $i \in J_k$ Do
            Set $x_i(t + 1) := y_i$;
        End For
    End For
    t:=t+1;
End Repeat
```

3.2 Synchronous implementation of block Gauss-Seidel model

The Gauss-Seidel iteration may be sometimes non-parallelizable. For instance, if every generation function g_i depends of all block components x_i, then only one of them can be updated at a time. Nevertheless, in some particular cases, certain blocks can be updates in parallel. We notice that there are several alternative of Gauss-Seidel algorithm corresponding to the same generation function G, because there is freedom in choosing the order in which the block components are to be updated. Also, if $P < n$, the computation strategy given by the subsets J_k, k=1,...,P, is free. The speed of convergence corresponding to different updating order and different strategy, is not exchanged in a great extent. Thus, it is natural to choose an ordering and a strategy for which the parallelization is maximized.

We present below two ways of choosing the ordering and the strategy.

Using the structure matrices.

Let B be the *structure matrix* corresponding to a function $G : X \to X$, i.e., a matrix which shows what variables appear in every function component. B is a boolean matrix $B = (b_{ij}), \quad i, j = 1, ..., N$, where $b_{ij} = 1$ if and only if the component g_i depends of the variable x_j. Note that both g_i and x_j mean simple components (not block) and consequently the indices i and j take their values between 1 and N. Suppose now that B has a block diagonal structure, $B = (B_1, ..., B_n)$, where B_i are square matrix with the following two properties:

(1) The successive two matrices $B_i, B_{i+1}, \quad i = 1, ..., n-1$, have partial overlap, that is if $B_i = (b_{kj}), \quad k, j = s, s+1, ..., s_i$ and $B_{i+1} = (b_{kj}), \quad k, j = t, t+1, ..., t_{i+1}$, then $t \leq s_i$. It must be emphasize that the case $t > s_i$ is a trivial, because the problem can be decomposed in two or more total distinct problems in such a case.

(2) Every other two matrices $B_i, B_{i+j}, \quad i = 1, ..., n-2; \; j = i+2, ..., n$, have no overlaps, that is, if $B_i = (b_{kj}), \quad k, j = s, s+1, ..., s_i$ and $B_{i+2} = (b_{kj}), \quad k, j = t, t+1, ..., t_{i+2}$, then $s_i < t$.

Briefly, we will say that $B_i \cap B_{i+1} \neq \emptyset$ and that $B_i \cap B_{i+j} = \emptyset \; , j \geq i+2$.

Note that the structure matrix corresponding to a function, defines also the decomposition that is required in parallel computation.

An example is given in fig. 1.

Assume for now that we are working with P processors, then the parallel processing of Gauss-Seidel model can be implemented as follow:

Starts with P blocks $B_{i_1}, ..., B_{i_P}$ for which $B_{i_k} \cap B_{i_j} = \emptyset$; in the above block diagonal structure depicted in the fig. 1, $P = 3$ and the starting diagonal blocks are B_1, B_3, B_5. When a processor P_j have finished its task corresponding to the block B_k, then that processor stars the processing corresponding to an other block B_j, such that $B_k \cap B_j = \emptyset$. Of course, all blocks must be covered in one wave. This process can be organized in different ways. For example, in the case of fig. 1 and if we are working with a system having three processors, the process is starting with the blocks B_1, B_3, B_5 and involved computation is defined as a semi-phase. The next semi-phase starts only when all processors were finished their tasks and consists in processing the blocks B_2, B_4, B_6. This will end a phase. The next phase begins with the blocks B_7, B_1, B_3, and so on. It must be pointed out that we have a pure block Gauss-Seidel iteration, except the order in which computation is done. As we have already mentioned, this order have not a major influence on the speed of convergence.

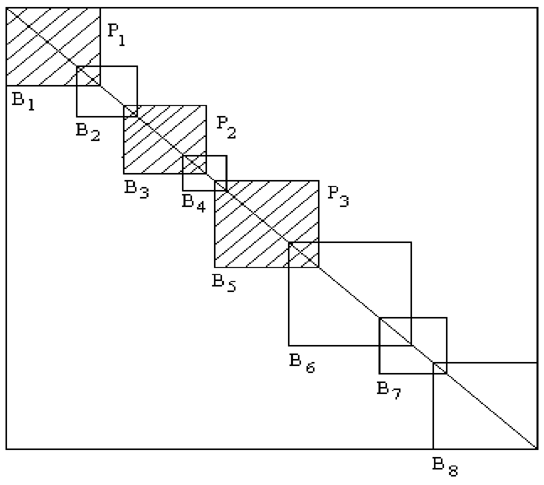


Fig. 1

Remark.

One of the most important element of such implementation, it seems to be the block diagonal structure of the matrix B. In the case of a sparse system, we always can put B in such block diagonal form, by reordering both variables and equations (see section 6)

Using the Directed Acyclic Graph[**?**].

A Directed Acyclic Graph (DAG) is a directed graph $G = (N, \Gamma)$ that has no positive cycles, where the set of nodes N is { 1,2,...,m}, corresponding to the operations performed by an algorithm and the set of arcs are used to represent data dependencies. Between two nodes i and j there exists an arc (i, j) if and only if the operation corresponding to node j uses the results of the operation corresponding to node i. A DAG represents the evolution of the computation in time. It specifies what operations are to be performed, on what operands, and imposes certain precedence constraints on the order that these operations are to be performed. In our case, the operations corresponding to each node i is the computation of $g_i(x(t))$ at a given time t.

For example, the DAG associated with the iteration of the form

$$\begin{aligned}
x_1(t+1) &= g_1(x_1(t), x_3(t))\\
x_2(t+1) &= g_2(x_1(t), x_2(t))\\
x_3(t+1) &= g_3(x_2(t), x_3(t), x_4(t))\\
x_4(t+1) &= g_4(x_2(t), x_4(t))
\end{aligned}$$

corresponding to two iterations is given in fig. 2.

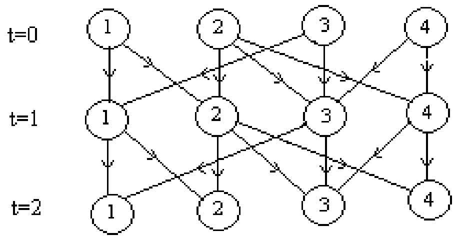


Fig 2

Such a DAG correspond to a Jacobi-type iterations, in which all the components of x are simultaneously updated, assuming that we are working with n processors.

The Gauss-Seidel iteration for the system above is of the form

$$\begin{aligned}x_1(t+1) &= g_1(x_1(t), x_3(t))\\x_2(t+1) &= g_2(x_1(t+1), x_2(t))\\x_3(t+1) &= g_3(x_2(t+1), x_3(t), x_4(t))\\x_4(t+1) &= g_4(x_2(t+1), x_4(t))\end{aligned}$$

The corresponding DAG is given in fig. 3.

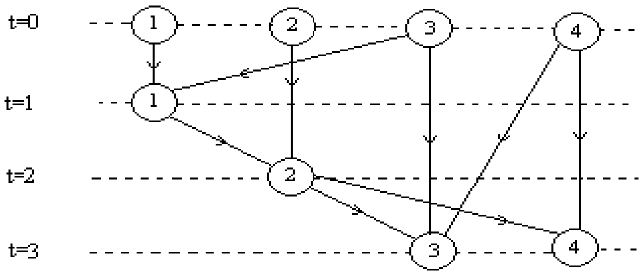


Fig. 3

At the time 1 are performed the operation associated to the node 1, that is the computation $x_1 = g_1(x_1, x_3)$; at the time 2 are performed the computation $x_2 = g_2(x_1, x_2)$, where x_1 was already updated, and so on. The operations associated to the nodes on the same level can be performed in parallel mod, because every level corresponds to a certain time and all data that come form the nodes on the superior levels are available. Does not exists a given order in which the operation must be performed, that is an aprioric arrangement of nodes on levels (supposing, of course, that the implicit precedence conditions are respected).

For example, an other DAG for the above Gauss-Seidel iteration is given in fig. 4.

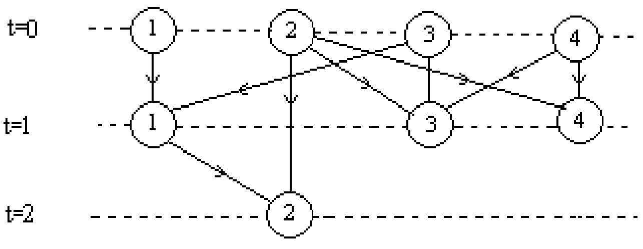


Fig. 4

It can see that x_1, x_3, x_4 may be update in parallel.

Remark.

The problem of finding a DAG that maximizes the parallelism is equivalent to an "optimal coloring" problem. Some results concerning this subject are given in [?].

4 Asynchronous algorithms

In what follows, we assume that there exist a discrete set of time $T = \{0, 1, ...\}$ at which one or more block components of x are update by some processor of a multi-processor system. In fact, this time is a general index that takes the values $0, 1, ...$ and it is increment with one every time when one block component of x is updated.

4.1 Time model for asynchronous algorithms [?]

. An important feature of the time model is that there exits a certain times (certain values of the general index), T^i, at which the component x_i is updated. We assume that the processor updating x_i may not have access to the most recent values of the components of x. So we suppose that

$$x_i(t+1) = g_i(x_1(\tau_1^i(t)), ..., x_n(\tau_n^i(t))), \quad \forall t \in T^i, \tag{2}$$

where $\tau_j^i(t)$ are times satisfying

$$0 \leq \tau_j^i(t) \leq t.$$

At all times $t \notin T^i$, x_i is left unchanged:

$$x_i(t+1) = x_i(t), \quad \forall t \notin T^i.$$

The differences $t - \tau_j^i(t)$ between the current time and the times $\tau_j^i(t)$ corresponding to the jth component available at the processor updating $x_i(t)$ can be viewed as the communications delays.

One of the most common asynchronous model is the shared memory system, that is described below.

The multiprocessor system have a shared memory which contains $x = (x_1, ..., x_n)$; every processor read x from this memory, performs its operation (more precisely, computes the new value of a certain component) and writes the new value in the corresponding place of the memory. The activities of processors are quite asynchronous, the processors do not wait each other for perform their task. Note that a processor i updates the corresponding component every time when $t \in T^i$.

The processing can be simulated by setting an infinite sequence of cells for every component; the cells will contain the successive values of that component with respect to the time. An example is given in the fig. 5.

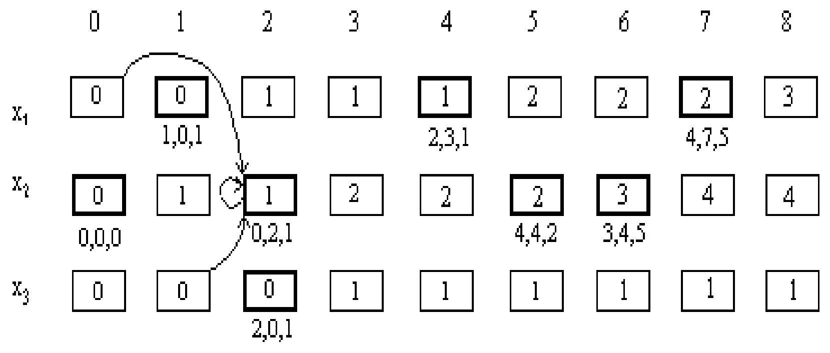


Fig. 5

In this example, The sets of times associated to every processors are: $T^1 = \{1, 4, 6, ...\}$, $T^2 = \{0, 2, 5, 6, ...\}$, $T^3 = \{2, ...\}$ (in the fig.5 the boxes corresponding to these times are draw with bulky lines). Each processor updates its block component at these times using the values of other components eventually outdated (the outdated times $\tau_1^i(t), ..., \tau_n^i(t)$) are written below of each updating time. For example, the component x_2 is updates at time 2 using the values $x_1(0)$, $x_2(2)$, $x_3(1)$.

4.2 Convergence analysis

This subsection was written following [?].

In the next, we make the following assumptions:

Assumption 1. The sets T^i are infinite, and if $\{t_k\} \subset T^i, t_k \to \infty$ then $\tau_j^i(t_k) \to \infty$ for every j.

This assumption guarantees that each component is updated infinitely often and that the old values of components will not be used in updating process. More precisely, given any time t_1, there exists a time $t_2 > t_1$ such that

$$t > t_2 \Rightarrow \tau_j^i(t) \geq t_1, \quad \forall i, j.$$

Assumption 2. There exists a sequence of descending sets $\{X(k)\}, \quad X(k) \subset X, \quad X(k+1) \subset X(k), \quad k = 1, 2, ...,$ satisfying

(a)(*Synchronous convergence condition*):

$$x \in X(k) \Rightarrow G(x) \in X(k+1), \quad \forall k.$$

Furthermore, if $\{y^k\}$ is a sequence such that $y^k \in X(k)$ for every k, then every limit point of $\{y^k\}$ is a fixed point of G.

(b)(*Box condition*): For every k, there exists sets $X_i(k)$ such that

$$X(k) = X_1(k) \times X_2(k) \times ... \times X_n(k).$$

Condition (a) implies that the limit points of a sequence generated by iteration $x := G(x)$, that is by the synchronous algorithms, are fixed points of G. The condition (b) implies that combining components of vectors in $X(k)$, we obtain vetcors in $X(k)$.

Theorem 1. *(Asynchronous Convergence Theorem) Suppose that assumptions 1 and 2 are fulfilled and that initial iteration belong to the set $X(0)$, that is $x(0) \in X(0)$. Then every limit points of $\{x(t)\}$ is a fixed point of G.*

Proof. We show that for each $k \geq 0$, there is a time t_k such that

$$(a)\ x(t) \in X(k), \forall t \geq t_k;$$
$$(b)\ for\ all\ i\ and\ t \in T^i\ with\ t \geq t_k,\ we\ have\ x^i(t) \in X_k,$$

where

$$x^i(t) = (x_1(\tau_1^i(t)), ..., x_n(\tau_n^i(t))), \forall t \in T^i.$$

We proceed by induction.

For $k = 0$ the induction hypotheses is true, since the initial estimate is assumed to be in $X(0)$.

Assuming it is true for a given k, we show that there exists a time t_{k+1} with the required properties. Let t^i the first element of T^i such that $t^i \geq t_k$. Then by condition (a) we have $f(x^i(t^i)) \in X(k+1)$ and further (in view of Box Condition):

$$x_i(t^i+1) = g_i(x^i(t^i)) \in X_i(k+1).$$

Similarly, for every $t \in T^i,\ t \geq t^i$, we have $x_i(t+1) \in X_i(k+1)$. Between elements of T^i, does not change. Thus

$$x_i(t) \in X_i(k+1),\ \forall t \geq t^i+1.$$

Let $t'_k = \max_i\{t^i\}+1$. Then, using the Box Condition we have

$$x(t) \in X(k+1),\ \forall t \geq t'_k.$$

Finally, since by assumption 1, we have $\tau^i_j(t) \to \infty,\ as\ t \to \infty,\ t \in T^i$, we can chose a time $t_{k+1} \geq t'_k$ that is sufficiently large so that $\tau^i_j(t) \geq t'_k$, for all i, j and $t \in T^I$ with $t \geq t_{k+1}$. We then have $x_j(\tau^i_l(t)) \in X_j(k+1)$ with $t \geq t_{k+1}$ and $j = 1, ..., n$, which implies that

$$x^i(t) = (x_1(\tau^i_1(t)), ..., x_n(\tau^i_n(t))) \in X(k+1).$$

The induction is complete. □

Example.

Suppose $G : I\!R^n \to I\!R^n$ is a contraction mapping with respect to the weighted maximum norm $\|.\|^w_\infty$. Recall that this norm is defined by

$$\|x\|^w_\infty = \max_i \frac{|x_i|}{w_i},$$

where $w \in \Re^n$ is a vector with positive coordinates. Suppose further that $X_i = \Re,\ i = 1, ..., n$. So $X = \Re^n$. The sets $X(k)$ are defined by

$$X(k) = \{x \in \Re^n |\ \|x - x*\|^w_\infty \leq \alpha^k \|x(0) - x^*\|^w_\infty\},$$

where x^* is the unique fixed point of G and $\alpha < 1$.

The assumption 2 is satisfied. Indeed, $X(k+1) \subset X(k)$, since

$$x \in X(k+1) \Rightarrow \|x - x*\|^w_\infty \leq \alpha\alpha^k \|x(0) - x^*\|^w_\infty < \alpha^k \|x(0) - x^*\|^w_\infty \Rightarrow x \in X(k).$$

If $x \in X(k)$ then $\|G(x) - x^*\|^w_\infty \leq \alpha \|x - x^*\|^w_\infty \leq \alpha^{k+1} \|x(o) - x^8\|^w_\infty$ which implies that $g(x) \in X(k+1)$. Now let $\{x^k\}$ a sequence defined by $x^{k+1} = G(x^k)$. Obviously, if $x^0 \in X(0)$ then $x_k \in X(k)$ and so $x^k \to x^*,\ k \to \infty$.

The Box Condition is also satisfied, because every set $X(k)$ is a sphere in $\Re^n$ centered at at x^* and of radius $\alpha^k \|x(o) - x^*\|^w_\infty$.

4.3 Counting set model for asynchronous algorithms

This subsection was written following [?].

We are keeping the sane concepts and notation as in previous model. Assume also that we are working with a shared memory parallel computer with P processors, $P \leq n$. Let us associate certain block components to each processor k, association that is defined by a set of indices $J_k \subseteq \{1,2,...,P\}$. We recall these sets as *association sets.* For example, if $n = 8$ and we are working with two processors, the association may be: $J_1 = \{1,3,4\}$ and $J_2 = \{2,5,6,7,8\}$. The sets J_k will be called the *counting sets* and we impose some natural conditions on them, such as, $J_{k_1} \cap J_{k_2} = \emptyset$ and $\cup J_k = \{1,...,n\}$.

The pseudocode for processor k will be:

```
Until convergence do
    Read $x$ from common memery
    Compute $y = g_i(x)$ for $i \in J_k$
    Overwrite $x_i$ in common memory with $y$
End until
```

As in the time model, we have a general time (counter) t which is incremented by one each time x is read from common memory by some processor k. Thus, x is made up of components each of which has been written back to memory as results of the computation belonging to some earlier iteration. At a given general time t, each component x_i has been iterated by its own number of times. So, we have $x(t) = (x_1^{s_1(t)},...,x_n^{s_n(t)})$. The model is complete if the sets J_k are setting for each time t, which we will denote by $J_k(t)$. Note that in particular $J_k(t)$ may be constant par rapport of t, which means that the association of the components to processors remain constant in time.

Concerning the association sets $J_k(t)$ and the iteration counters $s_1(t),...s_n(t)$ we makes the following assumptions:

Assumption 4.1. For each time $t = 0,1,...,$ and for all $i \in \{1,...,n\}$ the association sets and the iteration counters, satisfies:

$$\begin{array}{l} (a)\ s_i(t) \leq t-1; \\ (b)\ \lim_{t\to\infty} s_i(t) = \infty; \\ (c)\ Card\{t\ for\ which\ i \in J_k(t)\} = \infty. \end{array}$$

The condition (a) indicates that only components computed earlier (and not the future ones) are used in the current approximation. The second condition indicates that as the computation proceeds, every component is iterated infinitely many times. The condition (c) indicates that no component fails to be updates as time goes on.

Given an initial guess $x(0) = (x_1(0),...,x_n(0))$, the asynchronous iteration with associated sets $J_k(t)$ and iteration counters $(s_1(t),...s_n(t))$, is given by:

$$x_i(t) = \begin{cases} x_i(t-1), & i \notin J_k(t), \\ g_i(x_1^{s_1(t)},...,x_n^{s_n(t)}), & i \in J_k(t). \end{cases}$$

4.4 Partial asynchronous algorithms

Partial asynchronous algorithms refer to some restrictions on the sets of time T^i and on the outdated times $\tau^i_j(t)$.

Assumption 4.2.*(Partial Asynchronism)* There exists a positive integer B such that

(a) For every i and for every $t \geq 0$ at least one of the elements of the set $\{t, t+1, ..., t+B-1\}$ belongs to T^i;

(b) There holds

$$t - B < \tau^i_j(t) \leq t,$$

for all$i$ and j, and all $t \geq 0$ belonging to T^i.

(c) There holds $\tau^i_i(t) = t$ for all i and $t \in T^i$.

A consequence of this assumption is that the value of $x(t+1)$ depends only on $z(t) := x(t), x(t-1), ..., x(t-B+1)$.

We have the following theorem on the convergence properties of quasi-nonexpansive mappings.

Theorem 2. *Assume that $G : I\!R^N \to I\!R^N$ satisfies the conditions of partial asynchronous algorithms (assumption 4.2). Then the sequence $\{x(t)\}$ generated by the asynchronous algorithm converges to an element of an fixed point of G.*

5 Some classical algorithms

In the next sections we will develop some concrete synchronous and asynchronous algorithms for systems of nonlinear equations. Most of them will use the Jacobian of a nonlinear mapping from a finite dimensional space into itself, and we will suppose that the Jacobian will be decomposed conformally with x and $F(x)$.

5.1 Block-Newton algorithm

Suppose that $x \in I\!R^n$ and that the function $F : D \subset I\!R^n \to I\!R^n$ are conformally decomposed in n partitions (blocks) as in section 2. If F' is the Jacobian of F we let F' be decomposed conformally wit x as is suggested in fig. 6.

The block diagonal matrices is denoted by $\frac{\partial F_i(x)}{\partial x_i}$, $i = 1, ..., n$, that is

$$Diag(F'(x)) = (\frac{\partial F_1(x)}{\partial x_1}, ..., \frac{\partial F_n(x)}{\partial x_n}).$$

The Block-Newton (Jacobi) algorithm is

Initialization: $t = 0$ and $x(0) = x0$ (initial iteration);
Repeat until convergence

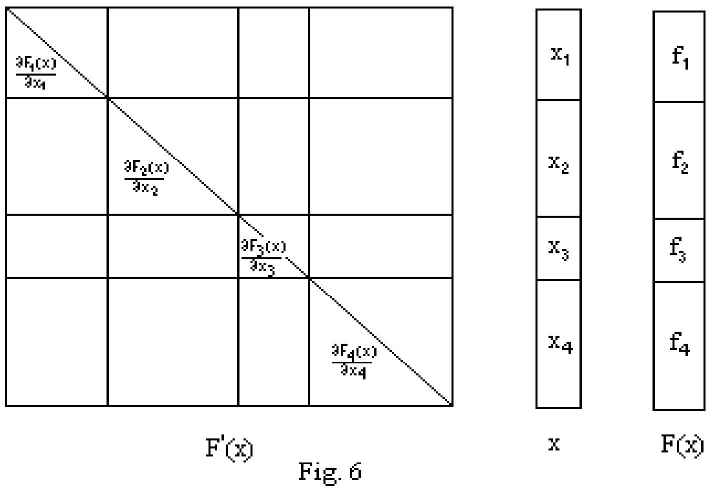

$\frac{\partial F_1(x)}{\partial x_1}$
$\frac{\partial F_2(x)}{\partial x_2}$
$\frac{\partial F_3(x)}{\partial x_3}$
$\frac{\partial F_4(x)}{\partial x_4}$
x_1
x_2
x_3
x_4
f_1
f_2
f_3
f_4
$F'(x)$
x
$F(x)$

Fig. 6

```
        For $i \in \{1,2,...,n\}$ Do
            $y_i = x_i(t) - (\frac{\partial F_i(x(t))}{\partial x_i})^{-1} f_i(x(t))$;
        End For
        Set $x(t+1) = (y_1,...,y_n)$;
        t:=t+1;
    End Until
```

The Block-Newton (Gauss-Seidel) algorithm is

```
    Initialization: $t = 0$ and $x(0) = x0$ (initial iteration);
    Repeat until convergence
        For $i \in \{1,2,...,n\}$ Do
            $x_i(t+1) = x_i(t) - (\frac{\partial F_i(x(t))}{\partial x_i})^{-1} f_i(x(t))$;
        End For
        t:=t+1;
    End Until
```

Notice that only the diagonal blocks are needed from the whole Jacobian. A suitable routine for the computation the necessarily partial derivatives can be implemented and so the computational effort should be widely reduced. Notice also that the Gauss-Seidel variant is in a great extent more performant, as results from numerical experiments below.

The two models (Jacobi and Gauss-Seidel) can be implemented both in synchronous or asynchronous mod, although the Gauss-Seidel algorithm seems to be more suitable for asynchronous implementation, and this case was largely studied in the literature. Concerning the convergence analysis of asynchronous Newton algorithm, we present in that follows the results of I. Lazar [?].

The standard assumptions on F are:

(C1): Equation $F(x)$ has a solution x^*;

(C2): $F' : D \subset I\!R^N \rightarrow I\!R^N$ is Lipschitz continuous on D, with constant γ , i.e., $\|F'(y) - F'(x)\| \leq \gamma \|y - x\|$, for all $x, y \in D$.
(C3): $F'(x^*)$ is nonsingular.

These conditions guarantee the convergence of the classical Newton method as well as some variant of classical Newton method, like, for example, the Newton-SOR method. In the case of asynchronous Newton method, the condition C3 will be replaced by the following sufficient conditions which guarantee the existence of solutions of the subsystems and local convergence of the asynchronous method:

(C3'): All the matrices $\frac{\partial F_i(x^*)}{\partial x_i}$, $i = 1, ..., n$ are nonsingular, and

$$\rho(D(x^*)^{-1}(F'(x^*) - D(x^*)) < 1.$$

Theorem 3. *Suppose that the conditions 4.1 of the total asynchronous algorithm are fulfilled and also that the conditions C1, C2 and C3' hold. Then there exists* $\delta > 0$ *such that if* $x^0 \in S(x^*, \delta)$ *then the sequence generated by the asynchronous Newton block method converges to* x^*.

The Newton method was adjusted for parallel computing in various ways. For example, in [?] the inexact Newton method is combining with block Cimmino method as the second iteration, for solving the linear system that arise in the main Newton iteration. It is also used the block Cimmino method as an inner solver in an inexact Quasi-Newton method, with Broyden secant update.

5.2 Block-Gradient method

The gradient-like method that we will consider here, have the following iteration function

$$G(x) = x - cF'(x)^T F(x),$$

where the descent direction is the gradient of the functional $\|F(x)\|^2/2$ at the point x (in Euclidean norm this is the vector $F'(x)^T F(x)$) and the steplength c satisfies the minimum conditions $\|x_{k+1} - x^*\| \approx min$. It is routine to see that $c = \|F(x)\|^2 / \|F'(x)^T F(x)\|^2$ [?], [?], [?]. In general, the convergence is linear and only in particular cases the convergence become superlinear, for example, if the condition number of $F'(x^*)$ is one.

The gradient Block-Gradient (Jacobi) algorithm (Fridman variant) is:

```
Initialization: $t = 0$ and $x(0) = x0$ (initial iteration);
Repeat until convergence
    For $i \in \{1, 2, ..., n\}$ Do
        $y_i = x_i(t) - (\frac{\partial F_i(x(t))}{\partial x_i})^T f_i(x(t))$;
    End For
```

```
        **Set** $x(t+1) = (y_1, ..., y_n)$;
        t:=t+1;
    **End Until**
```

The gradient Block-Gradient (Gauss-Seidel) algorithm is similarly.

5.3 Block-Gonjugate-Gradient method

We make first some remarks on the Nonlinear Conjugate Gradient method (briefly the NCG method) for the sequential case. Notice that the methods of this class preserve the good properties of gradient-like methods, such as the low effort of computation per iteration, a large possibility of the use the sparsness of the Jacobian, etc., and in the same time they have improved convergence. The NCG method was first introduced and studied by J.W.Daniel [?] for nonlinear operators with symetric Jacobian. Variants of this method were given in [?], [?], [?]. Our development is based on the paper [?] in which the weight factor b_k in the second iteration $p_{k+1} = F(x_{k+1} + b_k p_k$ (that improves the descent direction) is computed form conditions $\|x_{k+2} - x^*\| = min$. It is shows that the asymptotic behavior of this weight factor is like the Fletcher-Reeves factor for unconstrained optimization problems, that is, for k sufficiently large, $b_k \approx \|F(x_{k+1})\|^2/\|F(x_k)\|^2$. So, the NCG method is given by

$$\begin{aligned} x_{k+1} &= x_k - c_k F'(x_k)^T p_k, \\ p_{k+1} &= F(x_{k+1} + b_k p_k, \end{aligned}$$

where $c_k = \langle F(x_k), p_k\rangle/\|F'(x_k)^T F(x_k)\|^2$ and b_k is defined above. As usual, the first descent direction is $P_0 = F(x_0)$ and x_0 is a given starting point.

The Block-Conjugate-Gradient (Jacobi) method is:

```
**Initialization:** $t = 0$, $x(0) = x0$ (initial iteration), $p_0 = F(x_0)$;
**Repeat until convergence**
    **For** $i \in \{1, 2, ..., n\}$ **Do**
        $c_k = \langle F(x_i), p_i\rangle/\|F'(x_i)^T F(x_i)\|^2$;
        $y_i = x_i(t) - (\frac{\partial F_i(x(t))}{\partial x_i})^T p_i$;
        $b_i = \|F(x_{i+1})\|^2/\|F(x_i)\|^2$;
        $p_{i+1} = F(x_{i+1} + b_i p_i$;
    **End For**
    **Set** $x(t+1) = (y_1, ..., y_n)$;
    t:=t+1;
**End Until**
```

6 Preconditioning

The precondition of a nonlinear system involves, in the large, to compute preconditioners at every step of iteration. A global perconditioner, which is appropriate for entire iteration process, is of course also desirable, but in the nonlinear case the computational cost for such a preconditioner is highly expensive (for example, the approximation of the inverse of the Jacobian in the solution).

The general idea of preconditioning nonlinear systems consists in substitute the original function $F : D \subseteq R^n \to R^n$ with an equivalent one $G : D' \subseteq R^n \to R^n$ which have better properties concerning the computation of a solution (generally by a certain iterative method). The two functions are equivalent in the sense that they have the same solution. Other than having the same solution, the two functions may have completely different forms.

A very simple preconditioners of this type are the left and the right matrix preconditioners. The left preconditioner is a nonsingular matrix M and the preconditioned function is defined by $G := MF$. If Newton-like methods are used for solving the equation $G(x) = 0$, then it would be reasonable that $M = G(x^*)^{-1}$, where x^* is the solution of the equation $F(x) = 0$. The right preconditioned function is defined by $G(y) = F(Mx)$ and the proper unknown x is recovered from $x = My$. In the both cases the matrix M works as a global preconditioner.

More sophisticated preconditioners of this type are the single-level and multi-level nonlinear additive Schwarz preconditioners, [**?**], [**?**], [**?**]. For constructing single-level preconditioner, let $S = (1, 2, ..., n)$ be an set of index; i.e., one integer for each unknown x_i and F_i. Consider $S_1, S_2, ..., S_N$ a partition of S such that $\cup_{i=1}^{N} = S$ and $s_i \subset S$. It is allowed the subsets to have overlap, that is, if n_i is the dimension of S_i, then, in general, $\sum_{i=1}^{N} \geq n$. For each S_i it is defined $V_i \subset R^n$ as

$$V_i = \{v | v(v_1, ..., v_n)^T \in R^n, \quad v_k = 0, \quad if \quad k \notin S_i\}$$

and an $n \times n$ matrix I_{S_i} whose kth column is either the kth column of the identity $n \times n$ matrix if $k \in S_i$ or zero if $k \notin S_i$. Let $F_{S_i} = I_{S_i}$ and define the function $T_i : R^n \to V_i$ as the solution of the subspace nonlinear equation $F_{s_i}(v - T_i(v)) = 0$, for $i = 1, ..., N$. The single-level nonlinear additive Schwarz preconditioner is defined by

$$G(x) = \sum_{I=1}^{n} T_i(x).$$

The common way for preconditioning nonlinear systems is to use some linearization of a nonlinear system and preconditioning the resulting linear system at each nonlinear (outer) iteration. In fact, such a preconditioning attempts to improve the iterative process (inner iteration) for solving the corresponding linear systems. The inexact Newton method is probably the most popular algorithm in this class. The outer iteration involves the solution of the linear system $J(x_k)s_k = -F(x_k)$ and sets $x_{k+1} = x_k + s_k$; $J(x_k)$ is the Jacobian of F computed in the current iteration x_k. When n is large and the nonzero structure of $J(x_k)$ does not help, the classical direct methods (Gauss elimination, LU factorization, etc.) produce a large amount of fill-in and, so, they cannot be used for practical computation. It is used an iterative method (inner iteration) for solve the linear system at each step of outer iteration. Obviously, the linear systems must not be solved with the same accuracy during the whole process of outer iteration. The following problem arise: what must be the accuracy in solving linear systems? A natural idea is that this must be smaller as the outer iteration progress. In [?] the following stopping criteria is suggested: an increment s_k is accepted as an approximate solution if

$$\|J(x_k)s_k + F(x_k)\| \leq \theta_k \|F(x_k)\|,$$

where $0 < \theta_k \leq \theta < 1$ for all $k \in N$. Under suitable conditions, this algorithm has local linear convergence. If $\theta_k \to 0$ the convergence is superlinear.

In [?], some approximation of the Jacobian matrix is used in the linear iteration and GMRES method is applied for solve the linear system. Then Klylov subspace information generated by GMRES is recycled in order to compute step by step preconditioners. The general idea is either to deflate or to shift to one the eigenvalues of the matrix. The successive preconditioners are given by recursive formula of the form $M_k^{-1} := M_{k-1}^{-1} M_{(k-1)}^{-1}$ and by certain explicit formula for $M_{(k-1)}^{-1}$.

The Conjugate gradient (CG) method for inner iteration in inexact Newton method is proposed in [?], [?]. A least-change secant-update procedures is proposed to generate the matrix preconditioners B_k at each iteration of the inexact Newton method and then CG method is applied to the equivalent preconditioned system $B_k^{-1} J(x_k)s = -B_k^{-1} F(x_k)$. The preconditioners are generate using Broyden "good formula" so that they are easy to compute. An other type of preconditioning are incomplete LL^T factorizations for positive definite matrices, developed in [?].

The effect of the permutation algorithms on the performance of certain methods for linear systems is explored in some papers [?], [?], [?]. For example, in [?], a symmetric permutation of the matrix A of a system $Ax = b$ of the form $P^T AP$ is considered and then it is solved the equivalent system $P^T APy = P^T b$ and $x = Py$. The permutation algorithm produces a permuted matrix with dense diagonal blocks, while the entries outside the diagonal have magnitude below a prescribed threshold. The algorithm works with the graph of the matrix, choosing one node at a time, adding it to a group of nodes which would form the blocks along the diagonal, if the new node satisfies certain criteria. The amount of work is controlled by not searching through all the available nodes, but through a subset of eligible nodes, those which are adjacent to some nodes in the current set, i.e., in the group of nodes of the block along the diagonal being formed.

The structure of the paper is as follow. A brief description of our algorithm is given in section 2. The performance of this algorithm is illustrated in section 3, using spars matrices of moderate dimensions. In the last section the effective MathCad code is given together with necessary comments. Note that the additional analysis and experiments concerning the effect of this preconditioning on the certain Jacobi block method, will be reported in a forthcoming paper.

6.1 The algorithm

Let A be the "structure matrix" of a nonlinear system $F(x) = 0$, where $F : D \subseteq R^n \to R^n$, i.e., a matrix which shows what unknowns appear in every equation.

A is a boolean matrix $A = (a_{ij}), i, j = 1, ..., n$, where $a_{ij} = 1$ if and only if the component x_j appears in the ith equation. The main idea of the algorithm is to process the matrix A in such a way that the nonzero elements of A be gather around the main diagonal.

Let i_l, j_l and i_u, j_u be index in A which satisfy the following properties:

$(1) a_{i_l j_l} \neq 0,\ a_{i_u j_u} \neq 0;$
$(2) i_l \geq j_l,\ j_u \geq i_u;$
$(3) a_{i+k,k+1} = 0,\ k = 0, 1, ..., n - i + 1,\ \forall i > i_l - j_l + 1;$
$a_{k+1,j+k} = 0,\ k = 0, 1, ..., n - j + 1,\ \forall j > j_u - i_u + 1.$

These index define the diagonal band of the matrix, a_{i_l,j_l} being a nonzero element on the lowest diagonal and $a_{i_u j_u}$ being a nonzero element of the most up diagonal.

Remark. The condition (2) stipulates that the lowest diagonal is under the main diagonal and that the most up diagonal is over the main diagonal. If no, then a special case arise; for instance, if $i_l < j_l$ then either the solution of the system is trivial if $a_{ii} = 1$ for $i = 1, ..., n$, or some un-determinism are taking place.

Definition 1. The width of the diagonal band is defined by

$$w(A) = i_l - j_l + j_u - i_u + 1.$$

Remark. $w(A)$ represents the number of diagonals which compose the band of A.

Definition 2. A permutation matrix P is a boolean $n \times n$ matrix with the property that in every row and in every column of P there is exact one element hose value is one.

It is easy to see that the left product of the matrix A with P produces a permutation of rows, while the right product of the matrix A with P produces a permutation of columns. We will denote by $\mathbf{P}_{n \times n}$ the set of all the $n \times n$ permutation matrices.

The problem which we are interested in, consists in find a $P_l, P_r \in \mathbf{P}_{n \times n}$ for which

$$w(P_l A P_r) = \min_{P_L, P_R \in \mathbf{P}_{n \times n}} w(P_L A P_R)$$

The goal of these permutations is to produce matrices with dense diagonal band (hence, a nonlinear system with dense diagonal band of the Jacobian) and presumable with improved convergence properties for parallel algorithms which uses the Jacobian, like block Newton-Jacobi, block CG-Jacobi, etc. The eigenvalue of the Jacobian (usually computed in solution), rest unchanged and so a global iterative method couldn't be improved. Nevertheless, the diagonal blocks of the Jacobian could have essential diminished condition numbers and therefore the above mentioned methods should works better. We also mention that the matrices P_l, P_r may be interpreted as left and right preconditioners. So, in Newton method, the preconditioning linear system will have the form

$$P_l J(x) P_r y = P_l F(x).$$

Note that the solution is preserved.

The solution of a nonlinear system involves the following steps:

(1) Computing the structure matrix A;
(2) Computing the preconditioners P_l and P_r satisfying (1);
(3) Reordering the equations and the unknowns;
(4) Splitting of the systems in blocks with or without overlap;
(5) Applying a block Jacobi method.

In fact, the preconditioning is achieved in the step (2). In the following, we will give a simple algorithm for computing the two matrices P_l, P_r.

Let $c_i = (a_{1i}, ..., a_{ni})$ be the ith column of A. Let $l(c_i)$ denote the first element of c_i hose value is equal to one (i.e., if $l(c_i) = a_{j_l i} = 1$, then $a_{ji} = 0$ for all $j < j_l$, and similar, let $r(c_i)$ denote the last element of c_i hose value is equal to one, that is $r(c_i) = a_{j_r i} = 1$. Now, define the *average index of the column* c_i of the possible nonzero elements of c_i, by

$$mc_i = \left[\frac{j_l + j_r}{2}\right],$$

where [.] means the integer part.

Similar definition have the average index of the row j of the matrix, which we will denote by mr_j.

The algorithm.

```
Repeat until convergence
  Step 1: Compute $mc_i$ for $i = 1, ..., n$;
  Step 2: Transform the sequence $(mc_1, ..., mc_n)$ in a permutation
          of $(1, ..., n)$; $p_c$ denote this permutation;
  Step 3: Preform the permutation $p_c$ of columns in the
          matrix $A$;
  Step 4: Compute $mr_j$ for $j = 1, ..., n$;
  Step 5: Transform the sequence $(mr_1, ..., mr_n)$ in a permutation
          of $(1, ..., n)$; $p_r$ denote this permutation;
  Step 6: Preform the permutation $p_r$ of rows in the matrix $A$.
```

Remarks.

The step 3 involves the following processing: the maximum value of mc_i is forced on n (if more elements have this maximum, say k elements, then these elements are forced on $n, n-1, ..., n-k+1$) respectively; next, the second element of mc_i in magnitude is forced on $n-1$ (or on $n-k$); etc. Analogous for step 5. Obviously, the product of permutation matrices obtained successively in the steps 3 and respectively 5, give the searching preconditioners. More precisely, if $l_1, ..., l_m$ denote the permutation matrices obtained in the step 3 an if $r_1, ..., r_m$

denote the permutation matrices obtained in the step 5, then $P_l = l_m l_{m-1} ... l_1$ and $P_r = r_1 r_2 ... r_m$.

The algorithm will stops when no change turn up after step 3 or 6 or if a cycle arises during of the process. The convergence analysis and the complexity of this algorithm will be done in a future work. In the annex of this paper a detailed MathCad programm is given.

In a practical application, it is not necessarily to effective computing the preconditioners (the matrices P_l and P_r); the reordering of equations and the unknowns result directly from permutation matrices or even from the permutations computed in routines vr.

6.2 Numerical experiments

The reordering algorithm was tested on a significant number of spars matrices, from 8×8 until 20×20 dimension, and a special experiment on a 80×80 matrix. The results were encouraging, for all considered cases the algorithm works well, the reordered matrices having dense diagonal bands and the computational effort being reasonable, the number of iteration having the average value 18 in the cases of small matrices and the value 74 for 80×80 matrix.

To illustrate the effectiveness of the approach described in section 2, we show below the results for a spars 8×8 matrix. The nonzero elements of the matrix were taking as 1,2,...,17, in order to see the new positions of the elements after reordering.

$$A := \begin{pmatrix} 0 & 0 & 1 & 0 & 2 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 3 & 0 \\ 0 & 0 & 0 & 4 & 5 & 0 & 0 & 6 \\ 7 & 0 & 0 & 0 & 0 & 8 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 9 & 10 \\ 0 & 11 & 12 & 0 & 0 & 0 & 0 & 0 \\ 13 & 0 & 0 & 14 & 0 & 0 & 15 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 17 \end{pmatrix} \qquad B := \begin{pmatrix} 8 & 7 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 3 & 0 & 0 & 0 & 0 & 0 \\ 0 & 13 & 15 & 14 & 0 & 0 & 0 & 0 \\ 0 & 0 & 9 & 0 & 10 & 0 & 0 & 0 \\ 0 & 0 & 0 & 4 & 6 & 5 & 0 & 0 \\ 0 & 0 & 0 & 0 & 17 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 2 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 12 & 11 \end{pmatrix}$$

Fig. 7

It can see that the width of the diagonal band was reduced from 12 to 3, that is a 75% reduction in percents.

```
minc(a, j) :=
    i ← 0
    while  $a_{i,j} = 0$
        i ← i + 1
    return i
```

```
maxc(a, j) :=
    i ← n
    while  $a_{i,j} = 0$
        i ← i − 1
    return i
```

```
vc(a) :=
    for  j ∈ 0 .. n
        $vc_{0,j}$ ← j
        m ← minc(a, j) + maxc(a, j)
        mi ← $\frac{m}{2} - \frac{mod(m,2)}{2}$
        $vc_{1,j}$ ← mi
    return vc
```

```
maxv(vc) :=
    max ← −2
    for  j ∈ 0 .. n
        if  $vc_{1,j}$ > max
            max ← $vc_{1,j}$
            j0 ← j
    return j0
```

Fig.8

Comments.

Routine **minc(a,j)** and **max(a,j)**: Compute the first and the last nonzero elements of the column j of the matrix a;

Routine **vc(a)**: Compute *average index of the columns* of the matrix a. For example:

$$vc(A) = \begin{pmatrix} 0\,1\,2\,3\,4\,5\,6\,7 \\ 4\,6\,2\,4\,1\,3\,3\,4 \end{pmatrix}$$

Routine **maxv(vc)**: Compute the maximum of the second row of the matrix vc (usually, vc=vc(A)). For example, maxv(vc(A))=1.

Routine **vr(vv)**: Transforms the matrix vv (this matrix must have two rows, the first being 0,1,...,n) into a proper permutation. For example:

$$vc(A) = \begin{pmatrix} 0\,1\,2\,3\,4\,5\,6\,7 \\ 6\,7\,1\,5\,0\,3\,2\,4 \end{pmatrix}$$

Routine **vc(a)**: Performs the permutation given in routine **vr** on the columns of the matrix a.

The routines **minl, maxl, vl, sl** are similar with **minc, maxc, vc, sc** but they work on the rows (lines) of a matrix.

The matrices s and d are the preconditioners computed after 7 iterations, that is $s = s_7 s_6 ... s_1$ and $d = d_1 d_2 ... d_7$. It can see that sAd is the preconditioned matrix and that $w(sAd) = 3$.

```
vr(vv) :=  for  k ∈ 0..n
               j ← maxv(vv)
               $vr_{0,k} \leftarrow k$
               $vr_{1,j} \leftarrow n - k$
               $vv_{1,j} \leftarrow -1$
           return vr
```

```
sc(a) :=  vf ← vr(vc(a))
          for  j ∈ 0..n
              $jn \leftarrow vf_{1,j}$
              for  i ∈ 0..n
                  $a1_{i,jn} \leftarrow a_{i,j}$
          return a1
```

```
minl(a,i) :=  j ← 0
              while  $a_{i,j} = 0$
                  j ← j + 1
              return j
```

```
maxl(a,i) :=  j ← n
              while  $a_{i,j} = 0$
                  j ← j − 1
              return j
```

```
vl(a) :=  for  j ∈ 0..n
              $vl_{0,j} \leftarrow j$
              m ← minl(a,j) + maxl(a,j)
              $mi \leftarrow \frac{m}{2} - \frac{mod(m,2)}{2}$
              $vl_{1,j} \leftarrow mi$
          return vl
```

```
maxv(vl) :=  max ← −2
             for  j ∈ 0..n
                 if  $vl_{1,j} > max$
                     $max \leftarrow vl_{1,j}$
                     j0 ← j
             return j0
```

```
sl(a) :=  vf ← vr(vl(a))
          for  i ∈ 0..n
              $jn \leftarrow vf_{1,i}$
              for  j ∈ 0..n
                  $a1_{jn,j} \leftarrow a_{i,j}$
          return a1
```

```
vr(vv) :=  for  k ∈ 0..n
               j ← maxv(vv)
               $vr_{0,k} \leftarrow k$
               $vr_{1,j} \leftarrow n - k$
               $vv_{1,j} \leftarrow -1$
           return vr
```

Fig. 9

$$s := \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \end{pmatrix} \qquad d := \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \end{pmatrix}$$

$$s \cdot A \cdot d = \begin{pmatrix} 12 & 11 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 2 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 17 & 0 & 0 & 0 & 0 \\ 0 & 0 & 5 & 6 & 4 & 0 & 0 & 0 \\ 0 & 0 & 0 & 10 & 0 & 9 & 0 & 0 \\ 0 & 0 & 0 & 0 & 14 & 15 & 13 & 0 \\ 0 & 0 & 0 & 0 & 0 & 3 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 7 & 8 \end{pmatrix}$$

```
p(b, itm) :=  for  it ∈ 0 .. itm
                  b ← sc(b)
                  b ← sl(b)
              return  b
```

$$p(A, 30) = \begin{pmatrix} 8 & 7 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 3 & 0 & 0 & 0 & 0 & 0 \\ 0 & 13 & 15 & 14 & 0 & 0 & 0 & 0 \\ 0 & 0 & 9 & 0 & 10 & 0 & 0 & 0 \\ 0 & 0 & 0 & 4 & 6 & 5 & 0 & 0 \\ 0 & 0 & 0 & 0 & 17 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 2 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 12 & 11 \end{pmatrix}$$

Fig. 10

p(b,itm) is the general program; itm represent the iteration number which must be given by the user. For example, after 30 iteration the predonditioned matrix is given above.

Remark. The precondition matrix is not unique, in other words, the solution of (2) is not unique. In our example, $sAd \neq p(A, 30$, but $w(sAd) = w(p(A, 30))$.

The next experiments show the efficiency of preconditioning for nonlinear equations.

Nonlinear systems like:

$$
\begin{aligned}
&x_4^2 - 2x_8 + 1 = 0\\
&x_2 + x_6x_9 - 2 = 0\\
&x_9^2 - 1 = 0\\
&x_0 + 2x_9^3 - 3 = 0\\
&-x_1 + 3x_5 - 2 = 0\\
&x_5^2 - x_8 = 0\\
&2x_0x_3 - 2 = 0\\
&x_0 + x_1 + x_7 - 3 = 0\\
&x_5^2 - x_8^2 = 0\\
&x_0 - x_5 = 0
\end{aligned}
$$

$Ex1, 10 \times 10$	N	G	$FNCG$
$(0,4)(5,4)$	$*\,(sing)$	$*\,(500,0)$	$*\,(5000,0)$
$(0,2)(3,2)(6,3)$	$*\,(sing)$	$*\,(sing)$	$*\,(sing)$
$(0,3)(4,2)(7,2)$	$*\,(sing)$	$*\,(dep)$	$*\,(dep)$
$(0,4)(4,3)(7,2)$	$*\,(sing)$	$*\,(1000,0)$	$*\,(1000,0)$

$Ex1 - pre, 10 \times 10$	N	G	$FNCG$
$(0,4)(5,4)$	$*\,(sing)$	$*\,(500,0)$	$*\,(5000,0)$
$(0,2)(3,2)(6,3)$	$*\,(sing)$	$*\,(sing)$	$*\,(sing)$
$(0,3)(4,2)(7,2)$	$*\,(sing)$	$*\,(dep)$	$*\,(dep)$
$(0,4)(4,3)(7,2)$	$*\,(sing)$	$*\,(1000,0)$	$*\,(1000,0)$
$(0,5)(6,3)$	$*\,(sing)$	$10^{-3}(1000,0)$	$10^{-3}(1000,0), 10^{-13}(75,4)$
$(0,4)(4,5)$	$*\,(sing)$	$10^{-3}(500,0), 10^{-4}(100,2)$	$10^{-3}(500,0), 10^{-4}(100,4)$

$Ex2, 6 \times 6$	N	G	$FNCG$
$(0,2)(3,2)$	$*\,(sing)$	$*\,(500,0)$	$*\,(500,0)$
$(0,3)(3,2)$	$*\,(sing)$	$*\,(dep)$	$*\,(dep)$
$(0,1)(2,3)$	$*\,(sing)$	$*\,(sing)$	$*\,(sing)$
$(0,3)(4,1)$	$*\,(sing)$	$*\,(sing)$	$*\,(sing)$

$Ex2-pre, 6\times 6$	N	G	$FNCG$
$(0,2)(3,2)$	$10^{-13}(5,0)$	$10^{-7}(50,0)$	$10^{-7}(50,0), 10^{-9}(4,2)$
$(0,3)(3,2)$	$10^{-5}(4,0)$	$10^{-4}(50,0)$	$10^{-4}(50,0), 10^{-5}(10,2)$
$(0,1)(2,3)$	$10^{-15}(4,0)$	$10^{-4}(50,0)$	$10^{-4}(50,0), 10^{-6}(20,2)$
$(0,3)(4,1)$	$10^{-11}(5,0)$	$10^{-6}(100,0)$	$10^{-5}(100,0), 10^{-6}(20,2)$

$Ex2, 8\times 8$	N	G	$FNCG$
$(0,3)(4,3)$	$*\,(sing)$	$*\,(500,0)$	$*\,(500,0)$
$(0,4)(4,3)$	$*\,(sing)$	$*\,(500,0)$	$*\,(500,0)$
$(0,2)(3,4)$	$*\,(sing)$	$*\,(500,0)$	$*\,(500,0)$
$(0,4)(5,2)$	$*\,(sing)$	$*\,(500,0)$	$*\,(500,0)$
$(0,2)(3,1)(5,2)$	$*\,(sing)$	$*\,(500,0)$	$*\,(500,0)$

$Ex2, 8\times 8$	N	G	$FNCG$
$(0,3)(4,3)$	$10^{-6}(10,0)$	$10^{-6}(80,0)$	$10^{-6}(80,0), 10^{-6}(8,2)$
$(0,4)(4,3)$	$*\,(50,0)$	$10^{-6}(50,0)$	$10^{-6}(50,0), 10^{-6}(15,2)$
$(0,2)(3,4)$	$*\,(dep)$	$10^{-7}(200,0)$	$10^{-7}(200,0), 10^{-7}(30,2)$
$(0,4)(5,2)$	$*\,(1000,0)$	$10^{-6}(60,0)$	$10^{-6}(60,0), 10^{-8}(60,2)$
$(0,2)(3,1)(5,2)$	$*\,(100,0)$	$*\,(100,0)$	$*\,(100,0)$

7 Implementation

In this section we present some possibilities of implementation of the synchronous and asynchronous algorithms for nonlinear systems. We focus here on a few classical iterative methods such as Newton method, gradient method (Fridman variant), nonlinear conjugate gradient method (a variant considered in [**?**]). Our main purpose is to study the behavior of these methods from the convergence and from the rate of convergence view of point. For this reason, the parallelism will be only simulated, so that the pseudocode, which will be done in the MathCad language, can be performed on a serial computer.

The decomposition of the system in n blocks is defined by n pairs $(s_1, t_1), (s_2, t_2)$ $, ..., (s_n, t_n)$, where s_i is the component index of the block i and t_i represents the number of components of the block minus one. For example, the decomposition considered in the section 2 is defined by the pairs: (1,4), (4,4), (9,3), (12,3).

7.1 Synchronous algorithms

. *Block-Newton (Jacobi).* The main computation of the Block Newton Jacobi algorithm, that is the computation

$$y_i = x_i(t) - (\frac{\partial F_i(x(t))}{\partial x_i})^{-1} f_i(x(t))$$

is done in the routine *nj(x,s,t)*, where x is the current value of unknown and the pair (s,t) define the current block function and the diagonal block of Jacobian (the Jacobian computed in x is denoted by $D(x)$). The block function and the diagonal block of Jacobian corresponding to (s,t) are firstly moved in the vector b and in the matrix a; then some Newton steps are performed, $y \leftarrow y - a^{-1}b$, the initial value of y being the corresponding block of the current value of x. The main programm *Jacobi* update each component calling the routine nj and store the resulting updated block in a matrix z. Afterwards the next value of x is yielded from the z columns. The cycle *for* in which every block component is updates corresponds with the cycle **For** from pseudocode. These computations are repeated of km times in the first cycle *for* which corresponds to the cycle **Repeat** in pseudocode. The initial iteration $x0$, the number of blocks n and the value of km, are defined outside of the main programm.

The MathCad codes are given in fig. 8.

Block-Newton (Gauss-Seidel). The Gauss-Seidel variant is very similar with Jacobi; the unique difference consists in the fact that the cycle *for* in which the columns of z are moved in the suitable places of x, is inside of tha cycle *for* in which every block is updates. The MathCad code is gien in fig. 9.

7.2 Asynchronous algorithms

. *Block-Newton (Gauss-Seidel).*

We will refer to the asynchronous algorithms described in the subsection 4.1. An updated block component x_i is wrote in the shared memory as soon as it is obtained by some processor and it is used by the other processors at the corresponding time, so that only Gauss-Seidel model is appropriate for asynchronous case. Usually, the number of processors is less than the number of blocks, so that an assignment of blocks to processors is needed. When the time t belongs to the set T^i at which the component x_i must be updated, then the corresponding processor will achieve this, eventually after waiting that earlier updating process (or processes) is finished. Such waiting times may be of course included in the corresponding delays for every component and at every time (the interrupted lines in fig. 5).

The main idea of our implementation consists in building a matrix z hose dimensions are $N \times tm$, where N is the dimension of the space and tm is the maximum value of time (the maximum number of iterations). Every column of this matrix comprises the corresponding value of general variables x at the time t, the index of that column. The first columns must be filled in with the initial values of the iteration process and the starting value of t must be greater by one than the number of these initial columns.

The initial dates are:

(a) The number of partitions n; as the index in MathCad begins with zero, n is setting on the number of partitions minus one.

```
nj(x, s, t) :=
    f ← F(x)
    d ← D(x)
    for  k ∈ 0..0
        for  i ∈ 0..t
            $y_i \leftarrow x_{s+i}$
            $b_i \leftarrow f_{s+i}$
            for  j ∈ 0..t
                $a_{i,j} \leftarrow d_{s+i,s+j}$
        $y \leftarrow y - a^{-1} \cdot b$
    return  y
```

```
jacobi :=
    x ← x0
    for  k ∈ 0..km
        for  i ∈ 0..n − 1
            $z^{\langle i \rangle} \leftarrow nj(x, s_i, t_i)$
        for  i ∈ 0..n − 1
            for  $j \in 0..t_i$
                $x_{s_i+j} \leftarrow \left(z^{\langle i \rangle}\right)_j$
    return  x
```

```
gs :=
    x ← x0
    for  k ∈ 0..km
        for  i ∈ 0..n − 1
            $z^{\langle i \rangle} \leftarrow nj(x, s_i, t_i)$
            for  $j \in 0..t_i$
                $x_{s_i+j} \leftarrow \left(z^{\langle i \rangle}\right)_j$
    return  x
```

Fig. 11

(b) The partitions, which are defied by an array with $n+1$ rows ant two columns; every row contains a pair which defines a partition, the first number being the relative address of partition (the index of the first variable or equation in the corresponding block), and the second being number of the simple variables of block variable minus one. For example, in the case of the decomposition given in section 2, this array is

$$\begin{pmatrix} 0, & 4 \\ 3, & 4 \\ 8, & 3 \\ 11, & 3 \end{pmatrix}$$

(c) The initial values, a matrix with N rows and dm (the maximum value of delay) columns, which contains the starting values of x. Notice that the iterative process must begin with the iteration index $dm+1$.

(d) The maximum value of inner iterations, km, which defines the number of iterations of a particular method for a given block (throughout this paper it is supposed that the number of inner iteration is constant).

The strategy, the sets of times T^i and delays $\{d^i_j(t)\}$ are established as follows: the delays are constant with respect to the times, so that the delays are defined by a vector d with n components; the times T^i are defined in two ways: first, by yielding an array str_t which contains for each t the corresponding index of a block (the elements of str are integers between 0 and n); second, random generation the indices of blocks witch follow to be updated. This is done in the main programm by means of the instruction $i \leftarrow floor(|runif(1,0,n+1)|)$. The reason of this least choice of the strategy is motivated by the intention to study the behavior of different asynchronous algorithms with respect to such strategy.

The routines and the main programm are given in fig. 10.

We briefly describe the routines and the main program.

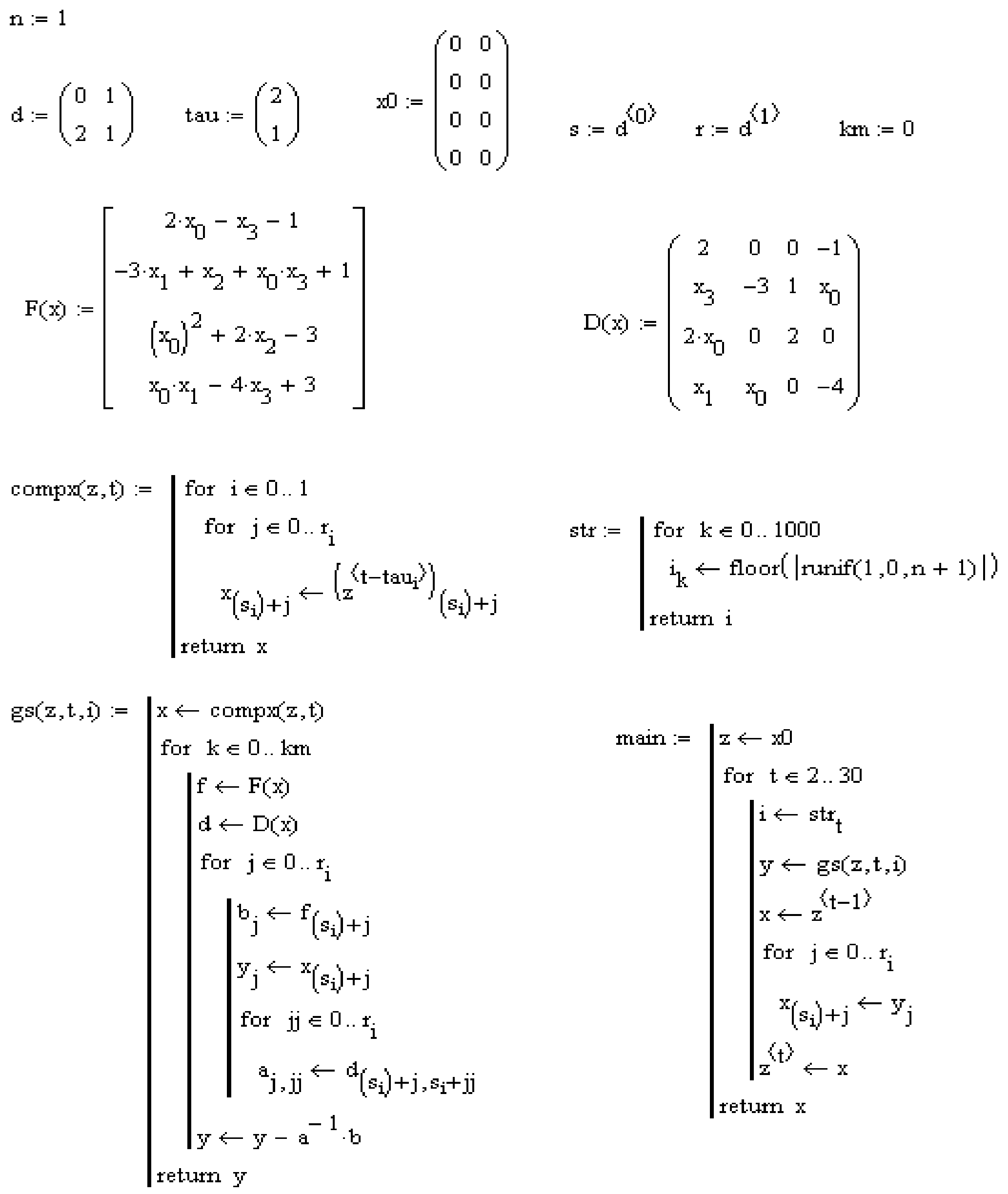
The routine compx(z,t). Given the old values of z till current time t and the delays for each block component, the routine actualizes the value of global variable x.

The routine gs(z,t,i). Iterates of km times a particular serial method. The diagonal block and the block function corresponding to a partition, are first loaded in the variables a and b respectively. Then a step of the serial method is performed in last but one instruction, like (for Newton method):$y \leftarrow y - a^{-1}b$.

The main programm main. First, the array z is setting by the initial values $x0$. Then a cycle on t, starting with $dm+1$ is performed until a maximum value tm. At every cycle an updated x is added to the array z. The programm returns the last value of x.

8 Stoping criterion

In the case of asynchronous algorithms each processor possesses only partial information on the general progress of the calculus. Some local conditions concerning the termination of the iteration for each processor must be defined in order to release the corresponding processor and introduce it into a load balancing process (next section). The simplest idea for the termination of the iteration



$n := 1$

$d := \begin{pmatrix} 0 & 1 \\ 2 & 1 \end{pmatrix}$ $\quad tau := \begin{pmatrix} 2 \\ 1 \end{pmatrix}$ $\quad x0 := \begin{pmatrix} 0 & 0 \\ 0 & 0 \\ 0 & 0 \\ 0 & 0 \end{pmatrix}$ $\quad s := d^{\langle 0 \rangle}$ $\quad r := d^{\langle 1 \rangle}$ $\quad km := 0$

$$F(x) := \begin{bmatrix} 2\cdot x_0 - x_3 - 1 \\ -3\cdot x_1 + x_2 + x_0\cdot x_3 + 1 \\ (x_0)^2 + 2\cdot x_2 - 3 \\ x_0\cdot x_1 - 4\cdot x_3 + 3 \end{bmatrix} \qquad D(x) := \begin{pmatrix} 2 & 0 & 0 & -1 \\ x_3 & -3 & 1 & x_0 \\ 2\cdot x_0 & 0 & 2 & 0 \\ x_1 & x_0 & 0 & -4 \end{pmatrix}$$

```
compx(z,t) :=
  for i ∈ 0..1
    for j ∈ 0..$r_i$
      $x_{(s_i)+j} \leftarrow \left(z^{\langle t-tau_i \rangle}\right)_{(s_i)+j}$
  return x
```

```
str :=
  for k ∈ 0..1000
    $i_k \leftarrow \mathrm{floor}(|\mathrm{runif}(1,0,n+1)|)$
  return i
```

```
gs(z,t,i) :=
  $x \leftarrow \mathrm{compx}(z,t)$
  for k ∈ 0..km
    $f \leftarrow F(x)$
    $d \leftarrow D(x)$
    for j ∈ 0..$r_i$
      $b_j \leftarrow f_{(s_i)+j}$
      $y_j \leftarrow x_{(s_i)+j}$
      for jj ∈ 0..$r_i$
        $a_{j,jj} \leftarrow d_{(s_i)+j,\, s_i+jj}$
    $y \leftarrow y - a^{-1}\cdot b$
  return y
```

```
main :=
  $z \leftarrow x0$
  for t ∈ 2..30
    $i \leftarrow str_t$
    $y \leftarrow gs(z,t,i)$
    $x \leftarrow z^{\langle t-1 \rangle}$
    for j ∈ 0..$r_i$
      $x_{(s_i)+j} \leftarrow y_j$
    $z^{\langle t \rangle} \leftarrow x$
  return x
```

Fig. 12

for processor i (the stoping criterion) is to stop the iteration when a condition of the form $\|x_i(t+1) - x_i(t)\| \leq \varepsilon$ is satisfied, where ε_i is a small positive constant reflecting the accuracy of the solution x_i at iteration i and $\|.\|$ is a suitable norm. To a certain extent, the problem is similar with the problem of stoping criterion for inexact Newton method, for which the second iteration (for the computing an approximative solution of the linear system) must stops when a certain accuracy is obtained. Note that, as in inexact Newton method, it's no use to compute the component x_i with a very high accuracy, because other components can be yet far of necessary accuracy and so to perform useless extra-computation for component x_i.

The subject was little treated in the specific literature and there are a few number of paper devoted to this problem. We will develop this issue following a paper of Bertsekas and Tsitsiklis [**?**]

Consider an asynchronous iterative algorithm, given of the iterative function $f : I\!R^n \to I\!R^n$, and suppose that the convergence conditions are fulfilled. For what following we need to impose some additional conditions.

Condition 9.1.

(a) If $t \in T^i$ and $x_i(t+1) \neq x_i(t)$, then processor i will send a message to every other processor;

(b) If a processor i has sent a message with the value of $x_i(t)$ to some other processor j, then processor i will send a new message to processor j only after the value of x_i changes (due to an update by processor i);

(c)Messages are received in the order that they are transmitted;

(d) Each processor sends at least one massage to every other processor.

It is easy to see that the condition $lim_{t\to\infty}\tau_j^i(t) = \infty$ follows from 9.1. Indeed, every processor gets informed of the changes in the variables of the other processors (according to 9.1 a). Also, if a processor i stops sending any massages (because x_i has stopping changing) then the last message received by processor j is the last message that was sent by processor i (due to condition 9.1 c) and therefore processor j will have updated information on x_i.

The concrete implementation of asynchronous algorithms with stoping criterion can be done by defining a new iteration function $g : I\!R^n \to I\!R^n$ as:

$$\begin{array}{ll} g_i(x) = f_i(x), & if\ \|f_i(x) - x_i\| \leq \varepsilon, \\ g_i(x) = x, & otherwise. \end{array}$$

The following natural question arise: If the asynchronous iteration given by the function f is convergent, is still the asynchronous algorithm given by the function g, convergent? Unfortunately the answer is negative, as the following example shows.

Example. Consider the function $f : I\!R^2 \to I\!R^2$ defined by $f_1(x) = -x_1$, if $x_2 \geq \epsilon/2$, $f_1(x) = 0$, if $x_2 < \epsilon/2$, and $f_2(x) = x_2/2$. It is clear that the asynchronous iteration given by the function f yields a sequence in $\Re^2$ which converge to $x^* = (0,0)$. In particular, x_2 is updated according to $x_2 := x_2/2$ and obviously it becomes smaller than $\epsilon/2$. Thus, processor 1 receives a value of x_2 smaller tan $\epsilon/2$ and the processor sets x_1 to zero. On the other hand, considering the asynchronous algorithm given by the function g and if the x_2 is initialized between $\epsilon/2$ and ϵ, then the value of x_2 never change, and the processor 1 will keep executing nonconvergent iteration $x_1 := -x_1$.

In the following we will analyse the conditions in which the asynchronous algorithm given by the function g is guaranteed to terminate. Let I be a subset of the set $\{1, ..., p\}$ of all processor. For each $i \in I$, let θ an element of X_i and consider the function $f^{I,\theta}$ defined by $f_i^{I,\theta}(x) = f_i(x)$ if $i \notin I$, and $f_i^{I,\theta}(x) = \theta_i$ if $i \in I$. We consider the asynchronous iteration given by the function $f^{I,\theta}$.

Theorem 4. *Let assumption 9.1 hold. Suppose that for any $I \subset \{1, ..., n\}$and for any choice of $\theta \in X_i$, $i \in I$, the asynchronous iteration given by the function $f^{I,\theta}$ is guaranteed to converge. Then the asynchronous iteration given by the function g terminates in a finite time.*

Proof. Let I be the set of all indices i for which the variable $x_i(t)$ changes only a finite number of times and for each $i \in I$, let θ_i be the limiting value of $x_i(t)$. Since f maps X into itself so does g. It follows that $\theta_i \in X_i$ for each i. For each i processor i sends a positive but finite number of messages [Assumptions 9.1(d) and 9.1(b)]. By assumption 9.1(a), the last message sent by processor i carries the value θ_i and by assumption 9.1(c) this is also the last message received by any other processor. Thus, for all t large enough, and for all j, we will have $x_i^j(t) = x_i(\tau_i^j(t)) = \theta_i$. Thus, the iteration given by the function g becomes identical with the iteration given by the function $f^{I,\theta}$ and therefore converge. This implies that the difference $x_i(t+1) - x_i(t)$ converges to zero for any $i \in I$. On the other hand, because of the definition of the mapping g, the difference $x_i(t+1) - x_i(t)$ is either zero, or its magnitude is bounded below by $\epsilon > 0$. It follows that $x_i(t+1) - x_i(t)$ settles to zero for every $i \notin I$. Thus shows that $i \in I$; this is possible only if $I = \{1, ..., n\}$, which proves the desired result.□

We now identify certain cases in which the main assumption in Theorem 9.1 is guaranteed to hold.

Consider first the case of monotone iterations and we assume that the iteration mapping f satisfies assumption ?. For any I and $\{\theta_i \in I\}$, the mapping $f^{I,\theta}$ inherits the continuity and monotonicity properties of f. Let u and v be as in assumption ? and suppose that $X = \{x | u \leq x \leq v\}$. Let θ_i be such that $u_i \leq \theta_i \leq v_i$. Sice f satisfies assumption ?(d), we have $f^{I,\theta(u) \geq u}$ and $f^{I,\theta(v) \leq v}$. We conclude that the mapping $f^{I,\theta}$ satisfies parts (a), (b) and (c) of assumption ?. Assumption ?(c) is not automatically true for the mapping $f^{I,\theta}$, in general; however, if it can be independently verified, then the asynchronous iteration given by $f^{I,\theta}$ is guaranteed to converge, and theorem 9.1 applies.

Let us now consider the case where f satisfies the contraction condition (?). Unfortunately, it is not necessarily true that the mappings $f^{I,\theta}$ also satisfy the same contraction condition. In fact, the mappings $f^{I,\theta}$ are not even guaranteed to have a fixed point. Let us strengthen (?) and assume that

$$\|f(x) - f(y)\| \leq \alpha \|x - y\|, \ \forall x, y \imath \Re^n,$$

where $\|.\|$ is again a block maximum norm, and $\alpha \in [0,1)$. We have $f^{I,\theta}(x) - f^{I,\theta} = \theta_i - \theta_i = 0$ for all $i \in I$. Thus

$$\begin{aligned}\|f^{I,\theta}(x) - f^{T,\theta}(y)\| &= \max_{i \notin I} \frac{1}{w_i} \|f_i(x) - f_i(y)\| \\ &\leq \|f(x) - f(y)\| \leq \alpha \|x - y\|.\end{aligned}$$

Thus, the mappings $f^{I,\theta}$ inherit the contraction property. This guarantees asynchronous convergence and therefore theorem 3 applies again.

9 Load Balancing

The load balancing problem arise as a natural matter in parallel computer architecture. A number of models and algorithms have been proposed for to solve it; especially so called *local* algorithms was studied, in which it is assumed that does not exists a central processor that coordinate the load balancing and that every processor contributes to perform it. The main goal of these algorithms is to control the processing in a parallel computer (or in a cluster of computers) such that the performance of the system be improved. The algorithms attempt to distribute the total load among the processors of the system as evenly as possible, so that to ensure that certain processors are not over-loaded while others are left idle. Briefly, the central idea of a local load balancing algorithm is to transfer some load from the highly loaded processor to its less loaded neighbors.

There exist two main classes of local algorithms [?].

(1) *Static algorithms.* The total load is available at the start of processing and no new load is added or existing load is removed. The static algorithms distribute the total load amongst processors before initiating the processing. From different reasons, new imbalance can appear soon after the processing was started, so that the load balancing algorithm must periodically called.

(2) *Dynamic algorithms.* The load is dynamically balanced. It is allowed at the beginning of each time (round) to introduce load at some processors (corresponding to the new jobs) and remove load from others (corresponding to jobs that have finished). Subsequently, the algorithm transfers loads across edges to tray to maintains the balance. Of course, the dynamic load balancing is more corresponding to real case and more challenging than the static version. Note that the variable topology of the network is also interprets as a characteristics of dynamic algorithms [?]

The static load balancing problem have been chiefly studied and various algorithms have been proposed. Conditions on the amount of the load which is transported and on the delay in transferring process, are usually imposed. The common results for static case state that load in every processor tends to the average of the total initial load [?]. In [?] the *single-port* model and *multi-port* model were analyzed and tight time bounds and imbalance bounds have been established in the both cases. It is supposed that in one unit of time at most one task (token) can be transmitted across an edge of the graph in each direction.

The effect of time delays on the stability of dynamic load balancing algorithms has been considered in [?], [?], [?], both for linear and for nonlinear cases. A processor receives the information concerning the level of the load of the neighborings processors delayed by a finite amount of time and then it uses this information to compute its local estimate of the average load in the network. The transfer itself is delayed as well. The stability of the processors loads is established provided that the both delays are bounded.

A new framework for dynamic load balancing in which the jobs traffic is modelled by an adversary have been introduced in [?] and further developed in [?]. The adversary controls the arrivals and removals of loads in the process. If the height of load in any processor and at any time is kept close to the average height of all processors , it says that the algorithm is stable against the given adversary. The main condition for stability is the *cut* condition which limits the increase of load in any subset of processors.

In [?] a static load balancing algorithm have been considered in the frame of discrete event systems. Suitable Lyapunov function is constructed for stability analysis.

Local balancing algorithms restricted to some particular network (asynchronous ring and a network consisting in two servers) have been studied in [?] and [?]

Usually, one considers that a single unit of load (a token) is transmitted across each edge in a round of time. If it supposed that a certain amount of load may transferred between two processors, then one of the key in the performance (rate of convergence, stability, etc.) of load balancing algorithms is just this amount of load.

9.1 The model

The network is described by a connected undirected graph $G = (V, E)$, where E is a finite set (the vertex) representing the processors, $Card(V) = p$, and $E \subseteq V \times V$ (the edges) is the set of arcs connecting the processors. There exists a general task which must be executed by network; the processors are cooperating for to achieve this goal. We shall suppose that the general task is shared in small subtasks (tokens) and that any token can be executed by any processor. The tokens are spreading over processors; we shall suppose that the load (the tokens) which is keeps by a processor may be described by a continuous variable. Let $x_i(t)$ be the level of load (the number of tokens) of the processor i at the time t. Each processor in the network sends its level of load to all its neighbors. A neighbor processor j receives this information form processor i delayed by a finite amount of time, τ_{ij}, that is, it receives $x_j(t - \tau_{ij})$. These information about load levels of others processors are used by each processor to compute the amount of load which is transferred to its neighbors. Further, the load sent by the processor i is received by processor j with the same delay.

Let a_t denote the average level of load per processor in the system at the beginning of time t. We say that a load balancing algorithm is stable if there exists constant B such that $|x_i(t) - a_t| \leq B$ for all processors and time t. For a set $S \subseteq V$, let $e(S)$ denote the set of edges with exactly one end in S, and let $\delta_t(S)$ be the net increase in level in set S due to the addition and removal of tokens in time t (note that $\delta_t(S)$ can be negative). We say that the dynamics of the system satisfies *cut* condition [?] if

$$|\delta_t(S) - |S|(a_{t+1} - a_t)| \leq |e(S)|.$$

Let $N_i \subseteq V$ denote the neighbors of the processor i, $N_i = \{j|(i,j) \in E\}$.

The following algorithm uses a weight factor c_i for each processor i which control the amount of tokens transferred from the processor i to every its neighbors. Let $\delta_{ij}(t) = x_i(t) - x_j(t - \tau_{ij})$ denote the total amount which processor i would transfers to processors j if no weight factor is used.

The algorithm:

```
At each time $t$ and for each processor
    for all $j \in N_i$
        If $\delta_{ij}(t) > 0$ then send $c_i\delta_{ij}$ tokens from $i$ to $j$.
```

9.2 Implementation on non-parallel computers

The implementation on a non-parallel computer is motivated for the possibilities of the delays control; it can simply assign an integer values to each delay τ_{ij} and suitable organize the processing. For instance, if all processors are executing the same task (say an iterative scheme for linear or nonlinear equations) in a synchronous mode, it can simulate this by building a random ordering in covering the cycle.

To each processor i it is assigned a vector x_i which memorizes the successive values of the numbers of tokens in that processor as time progress, that is the values $x_i(0), x_i(1), ...$

Static load balancing without delays

Algorithm SLBOD

```
    $x_i(0) := x0_i, \quad i = 1, ..., n;$
    For $t = 0, 1, ...$
        For $i \in [1, n]$ in random mode
```

$$x_i(t+1) := x_i(t) + \sum_{j \in N_i} c_j(x_j(t) - x_i(t)). \tag{1}$$

The values $x0_i, \quad i = 1, ..., n$, represent the initial loads of processors and let $\sum x0_i = L$ be the total initial load. This initial load is conserved, that is $\sum x_i(t) = L$ for all t. The conservation property result simply from the equation

$$\sum_{i=1}^{n} \sum_{j \in N_i} c_j(x_j(t) - x_i(t)) = 0. \tag{2}$$

The random execution of load balancing means, in fact, the random order in which iteration formula (1) is executed with respect the index i.

Static load balancing with delays

Initially, all components of state vectors are setting with values zero. In this way, a load sent from a processor i to processor j at the time t and which will reach the processor j at time $t + \tau_{ij}$, may be accumulate in the position $x_i(t +$

τ_{ij}). Let τ_m be the maximum values of delays, $\tau_m = max\tau_{ij}$. The components $x_i(0), ..., x_i(\tau_m)$ must be setting with the initials values in initialization step also. Note that the comparison between the levels of two neighboring processors is supposed delayed with the same value, that is $x_i(t)$ must be compared with $x_j(t - \tau_{ij})$. Let be $\delta_{ij}(t) := x_i(t) - x_i(t) - x_j(t - \tau_{ij})$.

Algorithm SLBWD

```
    $For\ i = 1, ..., n\ and\ fort = 0, ..., \tau_m$
        $x_i(t) := x0_i(t);$
    $For\ t = \tau_m + 1, ...$
        $For\ i = 1, ..., n$
            $x_i(t+1) := x_i(t+1) + x_i(t);$
        $For\ i \in [1, n]\ in\ random\ mode$
            $for\ j \in N_i$
                $if\ \delta_{ij} > 0$
```

$$x_i(t+1) := x_i(t+1) - c_j\delta_{ij}(t); \tag{3}$$

$$x_j(t+\tau ij) := x_j(t+\tau ij) + c_j\delta_{ij}. \tag{4}$$

The kernel of the SLBWD are the iterative formulas (3), (4), in which the principle of the algorithm is performed. If the processor i is highly loaded in comparison with its neighbor j, that is $\delta_{ij}(t) := x_i(t) - x_i(t) - x_j(t - \tau_{ij}) > 0$, then a portion of its load, controlled by c_j is transferred to processor j.

Dynamic load balancing with delays

Suppose that the arrivals and removals of load to the processors are performed with a specific , constant amount of load for each processor. This means that the load variation is linear in time. Such dynamics can be simulated by adding in the algorithm a new step of the form $x_i(t) := x_i(t) + \Delta l_i$ at the beginning or of the form $x_i(t+1) := x_i(t+1) + \Delta l_i$ at the end of the cycle for i in SLBWD algorithm.

9.3 Experiments

The experiments were made on different multi-computing systems, with or without delays and using static and dynamic load balancing algorithms. The main characteristics of our algorithm are that the load is transferred from a high loaded processor to all its less loaded neighbors and that the amount of transferred load if proportional with the difference between the two processor weighted with a weight factor c. We present here three multi-computing system with four processors each of them and with the connection graphs given in fig. 14. The system a is without delays; the other systems, b and c, are with delays and we suppose that the delays are constant in time and that are the same in the both directions on each edges (these delays are marked on the each edges by integer numbers).

The first experiment works with a static algorithm. The weight factor is chosen equal to 0.3. We see that after a few steps (about 15) the amount of

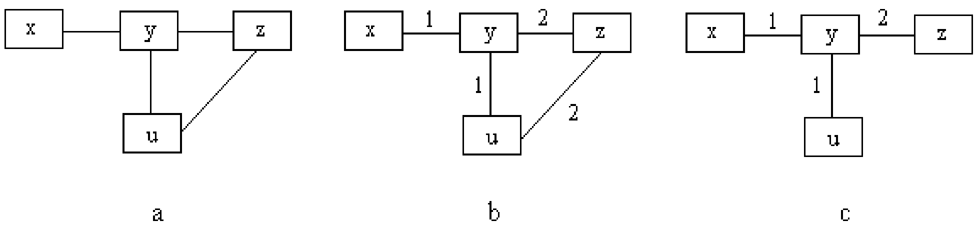


Fig. 13

loads for each processor tend to the average value (10) of total constant load (30), Fig. 14. Note also that the convergence seems to be monotone, at least after a few number of steps.

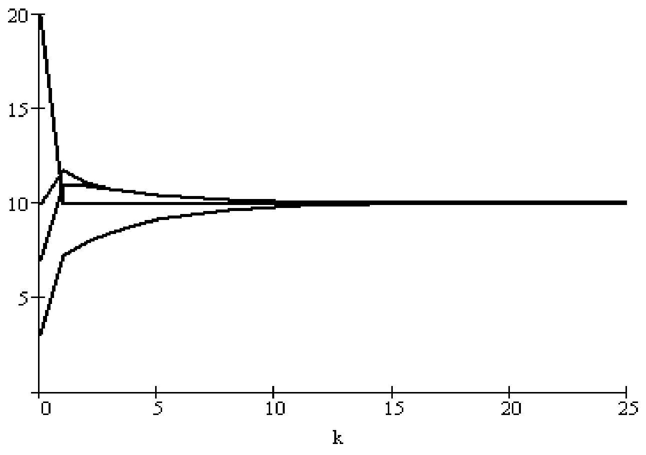


Fig. 14

The second experiment on the delayed system (b) works also with a static algorithm. The weight factor is 0.3. Because of delays, we need three initial values of loads for each processor and the balancing process must begins with the time 3. The loads on each processor tend also to the average value (5) of the total last initial values of loads (10+1+7+2=20), fig. 15. The convergence is obviously not monotone.

The third experiment is the most realistic, the multi-computing system is delayed in sone extent, as in computing grid, and the balance is performed dynamically, that is the load balancing algorithm is activated at every "tact" of a general cloak. The process of removal load for each processor is supposed to be linear, that is at each step the value of load for each processor is diminished with some positive constant. The weight factor is 0.5. The behavior of the system

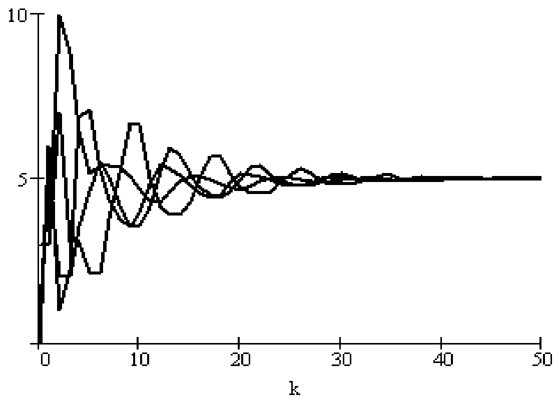


Fig. 15

is total different form those of static systems. The amount of loads for each processor does not converges to any value, but the system comes into a stability state, fig. 18.

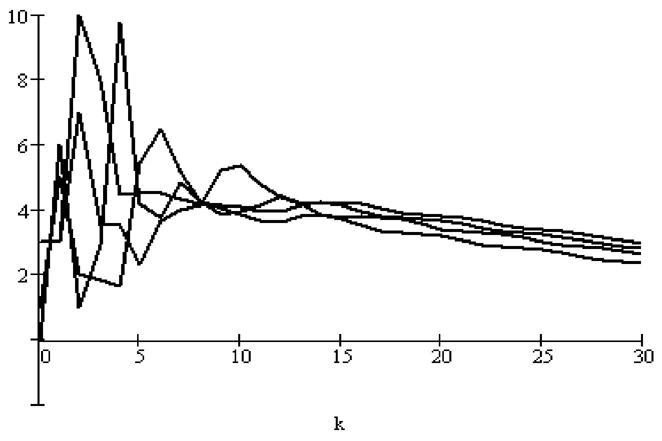


Fig. 16

10 Asynchronous Methods and Load Balancing Algorithms

Consider a nonlinear system which must be solved and an asynchronous method which has been chose for perform it; of course, the selection of a method for solving a given nonlinear system is a nontrivial task and there are few criterions of selection. Suppose also that a strategy J_k, $k = 1, ..., p$ was given and that a decomposition of the system was made. If the number n of blocks is greater then the number p of available processors, there are not any criterions for establishing the strategy, that is there are not any criterions for association of the block components to processors. Also, there are not criterions of decomposition. Thus, as the computation progress, certain in-balances between processors can appear. In this subsection we discuss the problem of using dynamic load balancing in asynchronous iterative algorithms, following the ideas of the paper [**?**].

The inbalance in the iterative process can appear from two reasons:

(a) The characteristics of the functions g_i attached to some processor p_k. For instance, if an iteration functions like Newton or gradient are used, then the rate of convergence depends essentially of the condition number of the Jacobian involved in the iterations.

(b) The heterogeneity of multi-computing system; usually, the machines involved in the parallel computing are of different speed and different capabilities.

Thus, the progression towards the solution is not the same for all the components of the system and some of them reach the fixed point faster than others. By performing a load balancing with some criteria based on this progression (the residual for example), it is then possible to enhance the repartition of the actually evolving computations over the processors. Of course, it is less imperative to have at all times exactly the same amount of work on each processors. The goal is rather to avoid too large differences of progress between processors. A non-centralized strategy of load balancing appears to be best suited since it avoids global communications which would synchronize the processors. The principle was described in previous subsection and consists in the fact 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 and distributes a part of its load: (a) to all these processors or, (b) only to the lightest loaded neighbor. In the our numerical experiments (next section), the first variant was considered.

The first problem is to define the "load of a processor" in our case. A simple and natural idea is to consider the total weighted residuals of the components associated to a processor at the time t as the load of the processor at that time. More precisely, let $J_k = \{k_1, ..., k_r\}$ be the strategy set corresponding to the processor k and let $\omega = \{\omega_{k_1}, ..., \omega_{k_r}\}$ be the weights associated by the components $x_{k_1}, ..., x_{k_r}$. Then we say that the load of that processor P_k is

$$Ld_k = \sum_j \omega_j \|g_{k_j}(x(t)) - x_{k_j}(t)\|,$$

for some norm $\|.\|$.

Remark. The weights ω should measure the inner characteristics of iteration functions g_i.

The second problem is to concrete define the transfer process of load form a processor to their neighbors. There are two ways for solve this, the both being an indirect distribution of the load.

1. The dynamic exchange the strategy. More precisely, let $J_k = \{k_1, ..., k_r\}$ and $J_l = \{l_1, ..., l_s\}$ be the strategy sets of indices corresponding to a processor

P_k and to its neighbor P_l at the time t; this means that processor P_k is updating the global variables $x_{k_1}, ..., x_{k_r}$ and the processor P_l is updating the variables $x_{l_1}, ..., x_{l_s}$. Suppose that $Ld_l < Ld_k$ and that a portion of load of P_k must be transferred to P_l. Then the simplest way is to pull out one (or more) component from J_k and add it (or its) to J_l. Thus, at the time $t+1$ the strategy sets becomes (for one component transferred): $J_k = \{k_1, ..., k_{j-1}, k_{j+1}, ..., k_r\}$ and $J_l = \{l_1, ..., l_s, k_j\}$. The choice of the component which is transferred should be an other subject of analysis. For example, we can chose that component which has the greatest residual, because our aim is to reduce the load of P_k in a meaningful measure.

2. The dynamic exchange the decomposition. Since the characteristics of the systems are generally unknown, there are not criteria of decomposition. Of course, a decomposition in the equal parts or so, do not guarantees the balance of loads, because the iteration functions g_i, which depend in a great extent of the system components involved in a given iteration function, can have totally different convergence properties. Thus, the decomposition of the system is a very difficult task and more challenging that strategy establishing. We suggest the dynamic exchange the decomposition based on the simple idea:

Simple algorithm for improving the decomposition. *At every step of iteration the convergence speed are checking for every component g_i and diminish by some extent (usually by one) the dimension of that component which has the most slow rate of convergence and enlarge with the same amount the component with the most fast speed.*

The first question is how the convergence speed can be estimated for a given iteration function g_i. Of course, a simple solution is to estimate the rate of convergence from some old values of that component, but this will increase the computational effort per iteration.

An other question is about the chance of improving the decomposition by applying this algorithms. It seems that there are few odds to obtain an improved decomposition by applying it. Nevertheless, this algorithm being rather an heuristic one and acting at every step with slow computational effort, it can considerable improving the decomposition. The numerical experiments yielded on this subject confirms the efficiency of the algorithm.

10.1 Experiments

The author has done a great number of numerical experiments concerning the asynchronous iteration for nonlinear systems. The following issues were pursued:

1. The decomposition of the system in blocks; both the number of blocks and the dimensions of each block were taken into consideration.

2. The strategy of the parallel computation, that is the assignation of blocks to processors.

3. The comparison of the different methods used in asynchronous algorithms.

The analysis of these experiments will be done in a future work. We note now only that the asynchronous algorithms (various implementations) have very interesting and surprising behavior, entirely different form sequential algorithms. We present below the case of a nonlinear system of 6 equations with 6 unknowns. The system has the form:

$$\begin{aligned}
&3x_0 + x_2^2 + x_5 - 5 = 0\\
&x_0^2 + 2x_1x_4 - 3 = 0\\
&x_1x_3x_5 + x_0 + x_4 + 3x_2 - 6 = 0\\
&2x_3 + x_1x_5 - 3\\
&2x_0x_4 + x_3^2 + x_1x_5 - 4 = 0\\
&x_1 + x_2x_4 + x_5^2 - 3 = 0
\end{aligned}$$

A solution of the system is $x^* = (1, 1, 1, 1, 1, 1)^T$ and the Jacobian is nonsingular for $x = x^*$, the system being medium conditioned (the condition number of Jacobian computed for $x = x^*$ is 7.276).

Both classical Newton and Gradient methods work well and the solution (with a precision of 10^{-8}) is obtained in 5 and 250 iterations respectively. Note that the quadratic rate of convergence for Newton method is fulfilled (we need only 5 iteration for obtaining the solution), and also that the gradient method seems to converges linearly to the same solution.

The behavior of asynchronous iterations is totaly different. We have tested the two methods, Newton and Gradient (Fridman variant), for six decomposition, each of them in three blocks. The decompositions are:

1	2	3	4	5
0 2	0 1	0 3	0 0	0 0
2 1	2 1	4 0	1 2	1 1
4 1	4 1	5 0	4 1	3 2

The behavior of the two asynchronous algorithms are given in the table below.

	1	2	3	4	5
N	500	$*$	380	$*$	$*$
G	600	550	650	380	250

The numbers written in the table mean the iteration number needed for Newton method (N) or Gradient method (G) to reach the precision 10^{-8}; the asterisk means that the method fails (does not converge).

The first remark is that the Gradient method is more sound than the Newton method; while Gradient method works for all cases, the Newton method fails for three times. An other important remark is that the behavior depends in a great extent of the decomposition; for example, the Gradient method needs 600 iterations in the case of the first decomposition and the same method needs only 250 in the fifth case.

References

1. Abdallah, C.T., et.al., Load balancing instabilies due to time delays in parallel computation, Proceedings of the 3th IFAC Conference on Time Delay Systems, Dec. 2001, Santa Fe, NM.
2. Abdallah, C.T., et.al., The effect of time delays in the stability of load balancing algorithms for parallel computation, from papper list of home page.
3. Aiello, W., Awerbuch, B., Maggs, B., Rao,S., Approximate load balancing on dynamic and asynchronous networks, Proc. ACM Symp. on Theory of Computing, 1993.
4. Anshelevich,E., Kempe, D., Keinberg, J., Stability of load balancing algorithms indynamic adversarial systems
5. O.Axelson, A.T.Chronopoulos, On the nonlinear generalized conjugate gradient methods, Numer. Math., Vol. 69 (1994), pp. 1-15.
6. J.M.Bahi, S.Contassot-Vivier, R.Couturier, Coupling dynamic load balancing with asynchronism in iterative algorithms on the computational grid, Laboratoire d'Informatique de Franche-Comte (LIFC), IUT de Belfort-Montbeliar, 2004
7. Z.Z.Bai, V.Migallon, J.Pedanes, D.B.Szyld, Block and asynchronous two-stage methods for mildly nonlinear systems, Numer. Math., Vol. 82 (1999), pp. 1-20.
8. Benzi, M., Choi, H., Szyld, D.B.,Threshold ordering for Preconditionong Nonsimetric Problems,
9. Bertsekas, D.P., Tsitsiklis, J.N., Parallel and distributed computation: Numerical Methods, MIT, Athena Scientific, Belmont, Massachusetts, 1997.
10. D.P.Bertsekas, J.N.Tsitsiklis, Convergence rate and termination of asynchronous iterative algorithms, Proceeging of the 1989 International Conference on Supercomputing, Crete, Greece, June, 1989, pp. 461-470
11. Bhaskar Ghosh, et. al., Tight analyses of the local load balancing algorithms, from papper list of home page.
12. Birdwell, J.D, et. al., The effect of time delays in the stability of load balancing algorithms for parallel computations, from papper list of home page.
13. A.Bojanczyk, Optimal asynchronous Newton method for the solution of nonlinear equations, J. Assoc. Comput. Math., Vol. 31 (1984). pp. 792-803.
14. Burges, K.L., Passino, K.M., Stability analysis of load balancing systems, Int. Journal of Control, Vol. 61, No. 2 (1995), pp. 357-393
15. Y.Chen, D.Cai, Inexact overlapped block Broyden method for solving nonlinear equations, Appl. Math. Comput., Vol. 26, No. 2/3 (2003), pp. 215-228.
16. A.T.Chronopoulos, Nonlinear CG-like iterative methods, J. Comput. Appl. Math., Vol. 40 (1992), pp. 73-89.
17. A.T.Chronopoulos, Z.Zlatev, Iterative methods for nonlinear operator equations, Appl. Math. Comput., Vol. 51 (1992), pp. 167-180.
18. J.W.Daniel, The conjugate gradient method for linear and nonlinear operator equations, SIAM J. Numer. Anal., Vol. 4, No. 1 (1967),pp. 10-26.
19. Dembo, R.S., Eisenstat, S.C., Steihaug, T., Inexact Newton methods, SIAM J. Num. Anal., Vol. 19 (1982), pp. 400-408.
20. Duff, I., Meurant, G.A., The effect of ordering on preconditioned conjugate gradient , BIT, Vol. 29 (1989), pp. 635-657.
21. A.Formmer, D.B.Szild, On asynchronous iterations, J. Comput. Appl. Marh., Vol. 123 (2000), pp. 201-216.
22. V.Fridman, An iterative process with minimum errors for nonlinear operator equations, Dokl. Akad. Nauk SSSR, Vol. 139 (1961), pp. 1063-1066.
23. Gehrke, J.E., Plaxton, C.G., Rajaraman, R., Rapid convergence of a load balancing algorithm for asynchronous rings

24. Jenkins, E.W., et al, A Newton-Krylov-Schwrtz methods for Ricard's equations, Technical report, North Carolina State University, Center for researche in Scientific Computational, Oct., 1999
25. Kees, C.E. at al.,Versatile Multilevel Schwartz Preconditioners for Multiphase Flow,
26. C.T.Kelley, Iterative methods for linear and nonlinear equations, SIAM, Philadelphia, 1995.
27. Kleinberg, J.M., A lower bound for two servrs balancing algorithms
28. I.Lazar, On the convergence of the asynchronous block Newton methods for nonlinear systems of equations, Studia Univ. Babes-Bolyai, Informatica, Vol. XLVII, No. 2 (2002), pp. 75-84.
29. L.Loghin, D.Ruiz, A.Touhami, Adaptive preconditioners for nonlinear systems of equations, CERFACS Technical Report TR/PA/04/02
30. St. Maruster, The stability of gradient-like methods, Appl. Math. Comput., Vol. 117 (2001), pp. 102-115.
31. St. Maruster, On the two stepgrsdient method for nonlinear equations, Proceeding of the Colloquium on Approximation and Optimization, Cluj-Napoca, 1984, Oct. 25-27, pp. 88-104.
32. St. Maruster, On the conjugate gradient method for nonlinear equations, Anal. Univ. Timisoara, Ser. Math. Info., Vol. XXXVII (1999), pp. 37-43.
33. J.M.Martinez, An extension of the theory of secant preconditioners, J. Comput. Appl. Math., Vol. 60 (1995), pp. 115-125.
34. J.M.Martinez, A Theory of Secant Preconditioners, Math. Comput., Vol. 60 (1993), pp. 681-698.
35. Muthukrishnan, M., Rajaraman, R, An adversial model for distributed dynamic load balaning, Proc. ACM Symp. on Parallel Algorithms and Architecture, 1998.
36. O'NeilJ., Szyld, D.B., A block ordering method for sparse matrices, SIAM J. Sci. Stat. Comput. Vol. 11 (1990), pp. 811-823.
37. Van Der Vorst, H.A., Dekker, K., Conjugate gradient type methods and preconditioning, J. Comput. Appl. Math., Vol. 24 (1988), pp. 73-87.
38. Xiao C.Cai, D.E.Keyes, L.Marcinkowski, Nonlinear Additive Schwarz Preconditioners and Applications in Computational Fluid Dynamics,?
39. Xiao C.Cai, D.E.Keyes, L.Marcinkowski, Two-level nonlinear Schwarz preconditioned inexact Newton methods, 2001 (in preparation)
40. Xiao C.Cai, D.E.Keyes, Nonlinearly preconditioned inexact Newton algorithms, SIAM J. Sci. Comput. 2000 (submited).
41. J.J.Xu, Convergence of partially asynchronous block quasi-Newton methods for nonlinear equations, J. Comput. Appl. Math., Vol. 321 (1999), PP. 307-321.
42. G.Zilli, L.Bergamaski, Parallel Newton Methods for sparse systems of nonlinear equations