Stud.Cercet.Stiint., Ser.Mat., 16 (2006), Supplement
Proceedings of ICMI 45, Bacau, Sept.18-20, 2006, pp. 409-422

Synchronous and asynchronous parallelization of some classical methods for systems resolution

Ioan Dzitac, Simona Dzitac and Emma M. Valeanu

Abstract

In this paper we present some synchronous and asynchronous parallelization of the Jacobi and Gauss-Seidel methods for linear and nonlinear systems resolution. Finally, we compare complexity of a parallel-synchronous variant implementation of the classical Jacobi method, in linear case, versus classical serial Gaussian elimination method.

Key words: parallelization, synchronous, asynchronous, linear system, nonlinear system

1. INTRODUCTION

Let us consider the following system of algebraic equations:

(1) $$f_i(x_1, x_2, ..., x_n) = 0, \quad i = 1..n$$

We will write the system in an equivalent way that is convenient for the application of iterative methods in order to obtain a numerical solution:

(2) $$x_i = f_i(x_1, x_2, .., x_n), \quad i = 1..n$$

The system described in (2) can be also written as a fixed point equation:

(3) $$x = f(x)$$

where

(4) $$f = (f_1, f_2, ..., f_n), \; x = (x_1, x_2, ..., x_n)$$

The classical iterative methods consist in finding the recurrent series looking like:

(5) $x^{k+1} = g(x^k), x^0 \in D,\ k = 0,1,2,....,$

that could converge to the separate solution over D,

$$x^* = (x_1^*, x_2^*, ..., x_n^*) .$$

For the classical iterative methods the components for the solution vector x^* are approximated sequentially by one serial computer.

In parallel computing the components can be approximated with distinct processors which can exchange information between them in different ways. We will describe some of these exchange methods in this article.

2. SYNCHRONOUS PARALLELIZATION OF JACOBI ALGORITHM

The simple successive approximations method (Jacobi) uses the following series:

(6) $x_1^{k+1} = g_i(x^k) = f_i(x^k),\ i = 1..n,\ k = 0,1,....$

We will use a virtual parallel computer. Also we will suppose that we have exactly *n* processors $P_1, P_2, ..., P_n$ (see Figure 1) for the approximation of the *n* components of the solution vector (if we have less then *n* processors, the tasks will be divided between them in a balanced way).

The P_i processor will iterate the *i* component after the formula:

(7) $x_i^{k+1} = f_i(x^k), k = 0,1,...,$

At every iteration step each of the *n* processors will deposit the result in the common memory MC, which is managed and controlled by the host processor *P*. The host processor will wait for the synchronization (in order that each processor will deposit its result) and will test the stop criteria. If the stop criteria is not fulfilled then the calculations will be resumed after the same algorithm with *k=k+1*.

Observation: If the operator $f = (f_1, f_2, ..., f_n) : D \to D,\ D \subset R^n$ is a contraction according to the canonical vector norm over D, then the algorithm will converge according the 2.2.9 theorem and *Robert-Charnay-Musy theorem* [1].

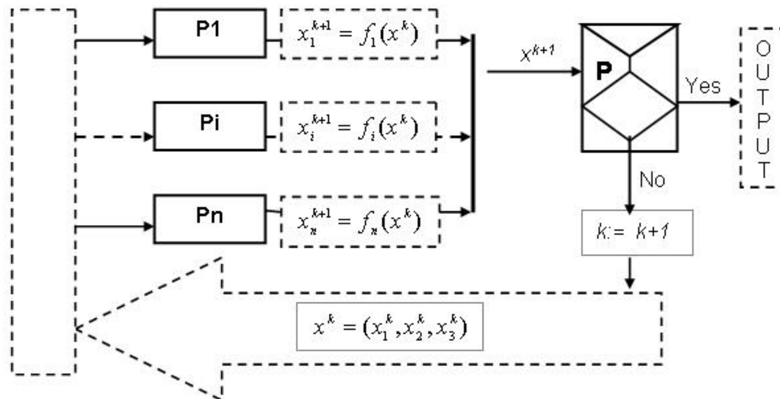

Figure 1: Synchronous parallelization of Jacobi algorithm

The pseudo-code procedure for the slave processor *P[i]* is presented in Table 1.

Table 1: The Jacobi procedure for the slave P[i]

```
procedure Jacobi P[i]
begin
        receive xold from P;
        {computes the i component with the formula}
          x[i]=f[i](xold);
        send    x[i] to P;
        wait message from P;
end.
```

Table 2: The synchronous parallel Jacobi procedure for the master P

```
procedure PAR_SINC_JACOBI
//vectors with n components: xinit, xold, xnew, error
select xinit {vector}; ε:= 0.00...01;
begin
xold:=xinit;
while (error > (ε,ε,…,ε))
        for i:=1 to n do in parallel
                send  xold to P[i];
        endfor;
        for i:=1 to n do in parallel
                cobegin P[i];
```

```
                    coend;
                    receive x[i] from P;
          endfor;
          xnew=(x[1],x[2],...,x[n]);
          error=||xnew - xold||→;
          xold: =xnew;
endwhile;
end.
```

3. ASYNCHRONOUS PARALLELIZATION OF JACOBI ALGORITHM

We present the asynchronous variant of the same parallel model in Figure 2. The P_i processor will iterate the *i*-th component using the formula:

$$x_i^t = q_i^t f_i(x^{t_i}) + (1 - q_i^t) x^{t_i}, \tag{8}$$

where

$$q_i^t = \begin{cases} 1, \textit{ if the period } t - t_i \textit{ is enough for computing and deposit of } x_i^t \\ 0, \textit{otherwise} \end{cases};$$

x_i^t is the *i* component of the current approximation of the solution vector for the *t* moment of time;

$x_i^{t_i}$ is the *i* component of the current approximation of the solution vector for the $t_i < t$ previous moment of time;

At every elementary iteration step each of the n processors will deposit its result in the common memory MC, managed and controlled by the host processor P which will not wait for the synchronization but will test the stop criteria. If this criterion is not fulfilled, the computations will be resumed after the same algorithm with the current approximation vector as it is in that moment.

In this way a certain processor that finished iteration will not wait anymore all the other processors to fulfil their computations and will iterate its component after the formula:

(9) $x_i^t = f_i(x^{t'})$,

where $x^{t'}$ is the current approximation vector which was formed from the modification of the components, from deposing its result and, eventually, from the results of the processors that finish in the same moment (casual synchronization).

Observation: If the operator $f = (f_1, f_2, ..., f_n) : D \to D,\ D \subset R^n$ is a contraction according to the canonical vector norm over D, then the algorithm converge according *Baudet theorem* [1].

Also, this algorithm is convergent every time the corresponding synchronous algorithm converges.

The functionality of the asynchronous algorithm procedure can be described in the following way.

For the asynchronous algorithm from Figure 2 the evaluation of the components is made with the formulas:

(10) $x_1^{p+1} = f_1(x^P)$,

where x^p is the value of the approximation of the *solution vector* that exists in the common memory from *P* in the moment when *P1* waits to execute again its instruction (without synchronization with the other processors; the synchronization can be also made but in a casual way);

(11) $x_2^{q+1} = f_2(x^q)$,

where x^q is the value of the approximation of the *solution vector* that exists in the common memory from *P* in the moment when *P2* waits to execute again its instruction (without synchronization with the other processors; the synchronization can be also made but in a casual way);

(12) $x_3^{r+1} = f_3(x^r)$,

where x^r is the value of the approximation of the *solution vector* that exists in the common memory from *P* in the moment when *P3* waits to execute again its instruction (without synchronization with the other processors; the synchronization can be also made but in a casual way).

In this way the components of the solution vector will change all the time until it is obtained the wished approximation. This is tested (as in the first case) by the *P* processor. According to Baudet theorem, if *f* is

a contraction in a vector norm, then the asynchronous algorithm is convergent.

If T_s is the *serial computation time* for solving the system (in other words, if we would use only one processor for the evaluation of all components), then, accordingly to the algorithm from figure 1, the *parallel synchronous computation time* T_{ps} will be:

(13) $$T_{ps} = T_s / n + T^s,$$

where T^s is the time wasted for the synchronization and transferring of the information.

If we eliminate the synchronization, in other words each processor will not wait anymore that is finished the computation of the other components after it finish the computation of its own component but will continue to iterate its component with the current value of the approximation of the solution vector, then we will obtain the asynchronous algorithm from figure 2 for which the parallel asynchronous computation time T_{pa} will be reduced considerably.

Then we will have:

(14) $$T_{pa} = T_s / n + T^a,$$

where $T^a < T^s$ is the time wasted only with the transmission of the information.

Using the procedure for the slaves processors described in Table 1, the asynchronous parallel Jacobi procedure for the master is described in Table 3.

Table 3: The asynchronous parallel Jacobi procedure for the master P

```
procedure PAR_ASYNC_JACOBI
//vectors with n components: xinit, xold, xnew, error
select xinit {vector}; ε:= 0.00...01;
begin
xold:=xinit;
while (error > (ε,ε,...,ε))
 for i:=1 to n do in parallel asynchronous
   send  xold to P[i];
   parbegin P[i];
   parend; {asynchronous}
```

```
    receive x[i] from P;
    change in xnew the received components;
    error=|| xnew-xold||→;
    xold:=xnew;
  endfor;
endwhile;
end.
```

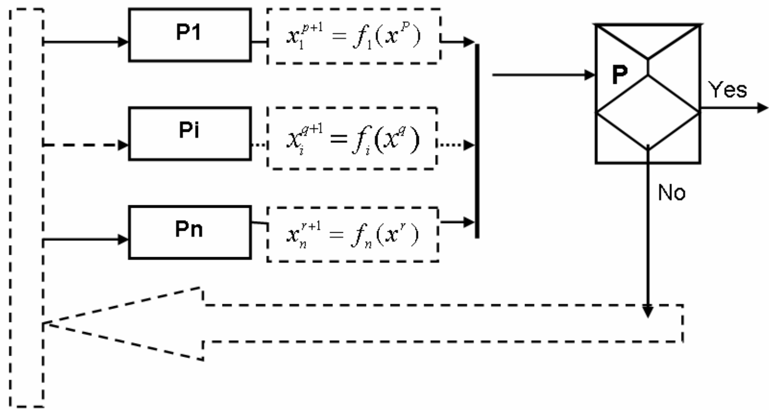

Figure 2: Asynchronous parallelization of Jacobi algorithm

4. PIPELINE MODEL OF GAUSS-SEIDEL ALGORITHM

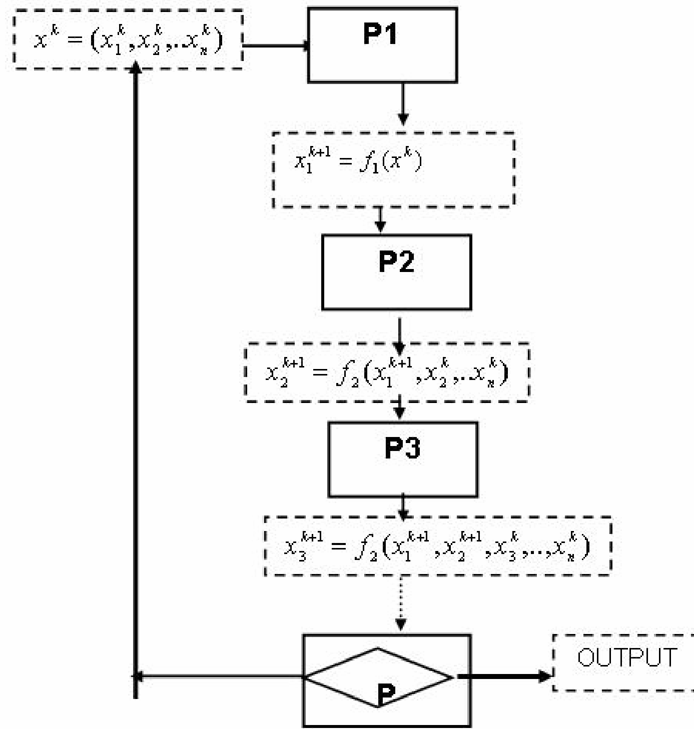

Figure 3: Pipeline variant (Gauss-Seidel algorithm)

Gauss-Seidel variant for the successive approximations method makes the following modification:

(15) $$x_i^{k+1} = g_i(x^k) = f_i(x_1^k,..., x_{i-1}^k, x_i^{k-1},..., x_n^{k-1}$$

In this case, the easiest method of parallelization is using a pipeline computation structure (see Figure 3).

In this model the computations are organized in the following way:

- The *P* processor sends to P_1 processor the initial value of the vector from the successive approximations series x^0, which will compute its component after the formula $x_1^* = f_1(x^0)$ .After the

computation it will send to P_2 processor the vector: $x^1 = (x_1^*, x_2^0, ..., x_n^0)$;

- The P_{i+1} processor take the vector x^i from the P_i processor and it will evaluate its component after the formula $x_{i+1}^* = f_i(x^i)$, and will send to its successor the vector. In this vector it will modify only the component that is computed by it self: $x^{i+1} = (x_1^i, ..., x_i^i, x_{i+1}^*, x_{i+1}^i, ..., x_n^i)$;
- The P processor receives the vector x^n from P_n, tests if the approximation is correct. If it's not then it will send the vector $x^0 = x^n$ to P_1 processor. The process will continue until the stop criteria are reached.

Observation: If the operator $f = (f_1, f_2, ..., f_n) : D \to D,\ D \subset R^n$, is a contraction according to the scalar norm over D, and then the algorithm will converge according to the *Banach Theorem.*

5. SYNCHRONOUS MODEL OF GAUSS-SEIDEL ALGORITHM

If in the sequential algorithm of Gauss-Seidel, at the vector step k we have the approximation of the system solution found with the vector $x^k = (x_1^k, x_2^k, ..., x_n^k)$, then at the grouped (vector) step $k+1$, the x^{k+1} vector components will be found with the following system:

$$\begin{cases} x_1^{k+1} = f_1(x^k) \\ x_2^{k+1} = f_2(x_1^{k+1}, x_2^k, ..., x_n^k) \\ ... \\ x_i^{k+1} = f_i(x_1^{k+1}, ..., x_{i-1}^{k+1}, x_i^k, ..., x_n^k) \\ ... \\ x_n^{k+1} = f_n(x_1^{k+1}, ..., x_{n-1}^{k+1}, x_n^k) \end{cases}$$

We can write the above system:

(16) $\quad x_i = f_i(x_1, x_2, ..., x_n), i = 1..n$,

as following:

(17) $g_i(x_1, x_2, ..., x_n) = 0,\ i = 1..n$

using the transformation $g_i(x) = x_i - f_i(x)$.

We reorganize the algorithm in the following way: we will solve the equations in comparison with x_i:

(18) $g_i(x_1^k, .., x_{i-1}^k, x_i, x_{i+1}^k, ..., x_n^k) = 0,$

After solving the equation we will assign $x_i^{k+1} = x_i$.

For each i between 1 and n, the task used for finding x_i can be given to P_i processor, concordant with the model from Figure 1.

Work procedures of the processors can be similarly described with the ones from the synchronous parallelization of Jacobi algorithm.

Observation: If the operator $f = (f_1, f_2, ..., f_n) : D \to D,\ D \subset R^n$ is a contraction according to the canonical vector norm over D, then the algorithm converge according *Robert-Charnay-Musy theorem* [1].

6. ASYNCHRONOUS PARALLELIZATION OF GAUSS-SEIDEL ALGORITHM

P_i processor will determine the x_i vector from the equation (18), in which we replace k with t (excepting the i-th component, all the other components are the last approximations that exist at the t moment). After that we will assign $x_i^t = x_i$. After each elementary iteration step each of the n processors will deposit its result in the shared memory MC, managed and controlled by the host processor P. The host processor will not wait for synchronization but it will do the stop criteria test. If these criteria are not fulfilled then the computations will be resumed after the same algorithm using the current approximation vector as it is in that moment.

Remark: If the operator $f = (f_1, f_2, ..., f_n) : D \to D,\ D \subset R^n$ is a contraction according to the canonical vector norm over D, then the algorithm converge according to *Baudet theorem* [1].

7. DISCUSSION OF SOME PARTICULARIZATIONS FOR A SYSTEM OF ALGEBRAIC LINEAR EQUATIONS

Let us consider a system of algebraic linear equations given by the equations from the formula:

(19) $$\sum_{i=1}^{n} a[i,j]*x[j] = b[j],\ \ j = 1..n.$$

Using the direct method, Gauss successive elimination, we will have the serial procedure from Table 4. We suppose that *a[k,k]* $\neq 0$.

Table 4: The serial procedure for Gaussian successive elimination [2]

```
procedure GAUSSIAN_ELIMINATION
begin
 for k:=1 to n do
 begin
          for j:=k to n do
                   a[k,j]:=a[k,j]/a[k,k];
                   c[k]:=b[k]/a[k,k];
                   a[k,k]:=1;
          endfor;
    for i:=k to n do
     begin
           for j:=k to n do
        a[i,j]:=a[i,j]-a[i,k]*a[k,j];
        b[i]=b[i]-a[i,k]*c[k];
        a[i,k]:=o;
         endfor;
    endfor;
 endfor;
end.
```

In a large way, for the sequential execution of the procedure described above, we will need approximate $n^2/2$ divisions on rows and approximate $(n^3/3-n^2/2)$ subtractions and multiplications on columns.

Supposing that the time for executing one arithmetic operation is equal with a unit of time then the serial response time will be about:

(20) $$T_s = 2 * n^3 / 3$$

So the approximate complexity in serial computation will be $O(n^3)$.

If we use a parallel implementation of the algorithm, with an elementary partitioning of the system matrix over a hypercube

structure with *n* processors, then a summary evaluation of the parallel execution time will be about:

(21) $$T_p = 3*n*(n-1)/2 + T_m*\log n + T_c*n*(n-1)*\log n/2,$$

where T_m is the consumed time for the initiation of sending the messages between two processors directly connected, and T_c is the transfer time *per word*.

So the complexity according the time of parallel implementation can be of order $O(n^2)$.

The same is the nature of the complexity of the serial Jacobi algorithm which demonstrates that the parallel implementation of Gaussian elimination method is not favourable.

In the next part we will describe the synchronous parallel implementation of Jacobi algorithm for SEAL resolution, with the *a posteriori* estimation of the error of approximation. In a similar way it can be implemented with an *a priori* evaluation of the error of approximation, the **while** cycle being replaced by a **for** cycle.

Table 5: Jacobi procedure for slaves (system (19))

```
procedure Jacobi P[k]
begin
receive xold from P;
{computes the k component with the formula}
 xnew[k] = b[k] + xold[k] - Σ_{j=1}^{n} a[k, j] * xold[j];
send    xnew[k] to P;
wait message from P;
end.
```

Table 6: The parallel synchronous Jacobi procedure for the master P (SEAL)

```
procedure PAR_SYNC_JACOBI
//vectors with n components: xinit, xold, xnew, error
select xinit {vector}; ε:= 0.00...01;
begin
xold:=xinit;
while (error > (ε,ε,….ε))
```

```
    for k:=1 to n do in parallel
      send  xold to P[k];
    endfor;
                  for k:=1 to n do in parallel
      receive xnew[i] from P;
                    endfor;
          xnew=(x[1],x[2],...,x[n]);
          error=|| xnew - xold|| →;
          xold: =xnew;
endwhile;
end.
```

In the serial parallel implementation described above, the complexity according the time is:

(22) $$T[n^2, p] = [n^2 / p] + \log p = T[n^2, n] = n + \log n$$

So the approximate complexity order of the parallel algorithm is O(n).

Obviously such an implementation is favourable, but more favourable will be the *parallel asynchronous implementation* where the synchronization time is eliminated.

8. CONCLUSIONS

We can observe that if a serial algorithm, for example the "Gaussian elimination algorithm", has a complexity $O(n^3)$, its parallel implementation on a hypercube with n processors reduces this complexity to $O(n^2)$, and the "parallel synchronous Jacobi algorithm" reduces this complexity even more, at O(n).

The system solving time can be reduced more if we apply an asynchronous implementation, in other word if we enforce that every processor use for the iteration of the current vector the existent values in the moment in which the processor is ready to do a new iteration, without waiting the synchronization with the other processors.

REFERENCES

[1] Dzitac, *Parallel and distributed procedures in resolution of some operator equations,* Univ. "Babes-Bolyai" Cluj-Napoca, 2002 (PhD Thesis, in Romanian)

[2] V. Kumar, V. Grama, A. Gupta, G. Karypis, *Introduction to Parallel Computing/ Design and Analysis of Algorithms*, The Benjamin / Cummings Publishing Company. Inc., 1994

Department of Business Informatics, AGORA University, Oradea, Romania,
idzitac@univagora.ro

Department of Energy Research, Faculty of Energy, University of Oradea