

# Parallel Algorithm for Solving Time Convolution Equations and Application to CEM

Alain Bachelot\*    Pierre Charrier \*    Agnès Pujols†  
Danièle Rouart \*

## Abstract

Boundary Integral Equation Methods, when used to study transient problems, require to solve a marching-in-time scheme which is the discrete equivalent of the convolution character of the continuous integral operator. Here we are interested in new issues for solving such schemes on parallel computers. We present and analyse a parallel algorithm and consider applications to Computational Electromagnetics. Results of numerical simulations on the CRAY T3D are provided.

## 1 Dense Systems of Discrete Time Convolution Equations

In this paper, we consider the solution of the following problem in  $R^n$

$$\begin{cases} \vec{X}^l = 0 & \text{for } l \leq 0 \\ M^0 \vec{X}^1 = \vec{B}^1 \\ M^0 \vec{X}^l = -\sum_{p=1}^{p_{max}} N^p \vec{X}^{l-p} + \vec{B}^l & \text{for } 2 \leq l \leq l_{max} \end{cases}$$

where  $M^0$  is a nonsingular  $n \times n$  matrix and  $N^p$ ,  $p = 1, p_{max}$  are given non zero  $n \times n$  matrices. In a first step we assume that all these matrices are dense.  $\vec{B}^l$  is a given sequence of vectors in  $R^n$ . Moreover we will consider situations where  $l_{max} > p_{max}$ . The problem is to compute the solution vectors  $\vec{X}^l$  for  $1 \leq l \leq l_{max}$ . This is a quasi-explicit marching in time scheme since, once the  $\vec{X}^k$ ,  $k = 1, l_{max}$  are available,  $\vec{X}^l$  can be computed by solving a linear system. For this, we have to perform at each time step the following operations

---

\*Applied Mathematics, University Bordeaux I, Talence France

†SIS, CEA-CESTA, Le Barp France.

- compute the convolution product  $\vec{C}^l = -\sum_{p=1}^{p_{max}} N^p X^{l-p} + \vec{B}^l$ ,
- solve in  $\vec{X}^l$  the linear system  $M^0 \vec{X}^l = \vec{C}^l$  and store the result  $\vec{X}^l$ .

We therefore obtain the algorithm given at Figure ??.

```

do l=1,lmax
  C= B(l)
  do p = 1, min(l-1,pmax)
    C=C- N(p) X(l-p)
  end do
  solve, for X(l), MX(l)=C
end do

```

Figure 1: Sequential Algorithm 1

A specific feature in this problem is that solving the linear system is not the main step in term of computational and storage complexity. The first step (convolution product) costs  $\mathcal{O}(3p_{max}n^2)$  flops and  $\mathcal{O}(p_{max}n^2)$  data to be stored. Solving the linear system (for instance by using a Conjugate Gradient type iterative method) costs ( $\mathcal{O}(Kn^2)$ ) floating operations and  $\mathcal{O}(n^2)$  data to be stored. As an example we give the time for solving such a problem on one processor of IBM SP 2 in Figure ??.

Table 1: Timing on one processor IBM SP 2 (in second)

n	$p_{max}$	$l_{max}$	total time	linear system	convolution product
1296	19	90	4170	135	4035

The numerical test confirms that the convolution products take the most important part of the execution time. So, when designing a parallel implementation for such a problem, we must take a special care of this step.

## 2 Parallel Algorithm

### 2.1 general remarks

In order to design a parallel algorithm to solve a dense system of discrete time convolution equations, we analyse dependences in algorithm 1. At a first level, we note that the various matrix-vector multiplications  $N^p X^l$ , for  $p = 1, p_{max}$  and a fixed  $l$ , are independent. Since a product  $N^i X^j$  gives a contribution to the right hand side  $C^{i+j}$ , the various terms  $N^p X^l$ , for  $l$  given, are associated with different time steps. So this first level of parallelism can be seen as a *time parallelism*. We can put it in evidence by rewriting algorithm 1 to get algorithm 2 described in Figure ??.

```

do p = 1, pmax
    Y(p)=0
fin pour
C=B(1)
solve, for X, MX=C
do l=2,lmax
    C= B(1)
    do p = 1,min(lmax -1, pmax)
        Y(p)=Y(p+1)- N(p) X
    end do
    C= C-Y(1)
    solve, for X, MX = C
end do

```

Figure 2: Algorithm 2

In algorithm 2  $X$  is the solution vector at time step  $l$  (i.e.  $X^l$ ), and the vector  $Y^p$  is an intermediate result containing at the beginning of the iteration  $l$  the partial sum

$$Y^p = - \sum_{q=p}^{l-2+p} N^q X^{l-1+p-q}.$$

Algorithm 2 is equivalent to Algorithm 1 and has the same computing and storage cost, but is well suited for parallel implementation since the various iterations of the inner  $p$  loop are independent. At the end of each iteration of the outer loop we have to solve a linear system. Finally let us

remark that a second level of parallelism inside the matrix-vector multiplications can be exploited.

## 2.2 Message Passing Parallel Implementation

We consider now a parallel implementation of algorithm 2 with a message passing SPMD programming model.

Each matrix  $N^p$  is distributed across the processors using the same row partitioning. So each processor stores a part of matrix  $N^p$  named  $N_{loc}^p$ . For load balancing, each processor has the same number of rows. Vectors  $B^l$  and  $Y^p$  are distributed according to the same scheme and local vector is named  $Y_{loc}^p$ .

The data structure for storing matrix  $M^0$  depends on the method used to solve the linear system. For an iterative solver such as a Conjugate Gradient method, the main computational part of the algorithm is a matrix-vector multiplication, and therefore the matrix  $M^0$  is distributed across the memory of processors by using the same scheme as for matrices  $N^p$ .

Vector  $X$  is computed in parallel as the solution of a linear system  $M^0 X = B$ , at the end of each iteration of the outer loop. Hence, its components are distributed. However, for performing efficient parallel row-oriented matrix-vector multiplications, it is convenient to have the complete vector  $X$  on each processor. Indeed, in this case, a parallel matrix-vector multiplication can be performed without communication. So, after solving the linear system, the vector  $X$  is reconstructed on every processor. This operation is equivalent to a *gather* followed by a *broadcast*, and is named in the following a step of *reconstruction* of the vector  $X$ . This means that there are two data structures to store  $X$ .

For solving the linear system we use a preconditioned Conjugate Gradient method (PCG). The preconditioning is performed by a  $LDL^T$  factorization of each local block associated with the row-partitioning and therefore is completely local.

Figure 3 gives the pseudo-code of the message passing parallel implementation of this algorithm.

Table ?? and Table ?? give timings for solving systems of discrete time convolution equation on a Cray T3D using the algorithm described above. It appears that the parallel efficiency obtained for the computation of the convolution term is quite good. At the opposite the parallel PCG solver exhibit a poor scalability. Two reasons can explain this behaviour

```

do p = 1,pmax
    Yloc(p)=0
end do
C=B(1)
solve in parallel, for X(1), MX(1)=C
reconstruction of X
do l=2,lmax
    Cloc= Bloc(l)
    do p = 1,min (lmax-l,pmax)
        Yloc(p)=Yloc(p+1)- Nloc(p) X
    end do
    Cloc= Cloc-Yloc(1)
    solve in parallel, for X, MX=C
    reconstruction of X
end do

```

Figure 3: Pseudo-code of the message-passing parallel implementation

- Consider a system of  $n$  equations and  $p$  processors. Each iteration of PCG involves (per processor)  $n^2/p$  flops (matrix-vector multiplication) and  $(p-1)n/p$  data sent (reconstruction). Hence the ratio computation/communication becomes worse when  $n/p$  becomes small, i.e. for small granularity.
- The block-diagonal preconditioning tends to be less efficient when the number of processor increases.

The convolution product which is the dominant component in sequential computations is implemented with a quasi perfect scalability. As a consequence, solving the linear systems can become the dominant part for parallel simulations with a high number of processors. Nevertheless the overall parallel performance achieved is quite satisfactory.

Table 2: Timings for n= 411 on a Cray T3D (in second)

# proc	total time	linear systems	convolution products
4	85	65	17
8	49	38	8
16	34	27	4
32	31.5	26	2

Table 3: Timings for n=1296 on a Cray T3D (in second)

# proc	total time	linear system	convolution products
16	248	155	85
32	138	90	43
64	85	60	22

### 3 Application to CEM

Many problems in science and engineering involve determining the scattering of a transient wave by an obstacle. For solving such problems the Boundary Integral Equation Method has the advantage over domain-based formulations that only the compact boundary of the obstacle has to be meshed for finite element computations. Moreover they do not need any special treatment to take into account the radiation condition at infinity. After approximation by finite element these models lead to systems of discrete time convolution equations.

We recall some background on this method in the case of an electromagnetic wave scattered by a conducting obstacle. We denote  $\Gamma$  the boundary of the tridimensional obstacle,  $\vec{E}$  the electromagnetic field,  $\vec{c}(t, x)$  the incident wave and we introduce an auxiliary unknown  $\vec{\varphi}$  defined on  $\Gamma$  by

$$\vec{\varphi} = [\text{rot} \vec{E} \wedge \vec{n}]$$

Then we can prove ([?], [?], [?]) that  $\vec{\varphi}$  is solution of the following variational

problem :

Find  $\vec{\varphi}$  such that  $\forall \vec{\psi}$ ,

$$\int_0^{+\infty} \int_{\Gamma \times \Gamma} \left\{ \frac{\vec{\varphi}(t - |x - y|_c, y) \partial_t \vec{\psi}(t, x)}{4\pi |x - y|} + c^2 \frac{\text{div}_\Gamma \partial_t \vec{\psi}(t, x)}{4\pi |x - y|} \int_0^{t - |x - y|_c} \int_0^s \text{div}_\Gamma \vec{\varphi}(r, y) dr ds \right\} dy dx dt$$

$$= \int_0^{+\infty} \int_\Gamma \vec{c}(t, x) \partial_t \vec{\psi}(t, x) dx dt$$

The finite element approximation is obtained by taking

$$\vec{\varphi}_h(t, y) = \sum_{j=1}^n \alpha_j(t) \vec{\varphi}_j(y) \quad \text{and} \quad \vec{\psi}_h(t, x) = \beta_i(t) \vec{\varphi}_i(x)$$

where  $\vec{\varphi}_j$  are the basis functions of the finite element method of Raviart and Thomas ([?]). For the time approximation, we use the  $P_0^t \times P_0^t$  scheme introduced in [?] and defined by

$$\alpha_j(t) = \sum_{m \geq 1} \chi^m(t) X_j^m \quad \text{and} \quad \beta_i(t) = \chi^l(t), \quad l \leq 1$$

where

$$\chi^m(t) = \begin{cases} 1 & \text{if } t \in [t_{m-1}, t_m[ \\ 0 & \text{else} \end{cases}$$

Then the vectors  $X^l$  satisfy a system of discrete time convolution equations with  $p_{max}$  matrices  $N^p$  where

$$p_{max} = \left\lceil \frac{diam}{c\Delta t} \right\rceil + 2$$

where  $diam$  is the diameter of the obstacle (so for 3D problems  $p_{max} = \mathcal{O}(n^{1/2})$ ).

A finite element code for transient BIEM is structured in two steps. The first step constructs the matrices  $M^0$  and  $N^p$  and vectors  $B^l$ . The second step solves the system of discrete time convolution equations. Table 4 gives a timing for these two steps on a problem associated with a mesh with 1296 degrees of freedom.

Table 4: Timings for  $n=1296$  on one processor Cray J 90

Total time	Construction step	Solution time
4836	2449	2376

## 4 Parallel Implementation of BIEM for Transient Problems

A parallel code for transient BIEM is also structured in two steps performed one after the other.

The first step constructs matrices  $M^0$  and  $N^p$ ,  $p = 1, p_{max}$ , and vectors  $B^l$ ,  $l = 1, l_{max}$ . Since the finite element basis functions have a compact support, each of these matrices is sparse but has a specific sparsity pattern. By taking into account this sparsity the amount of storage and the computing time can be strongly reduced. The sparsity pattern of the matrices is determined in a preliminary phase of the parallel code (*symbolic construction*) and the matrices are stored using the CSR SPARSKIT format. They are distributed across the processors, as indicated in paragraph 2.2. Therefore we do not take into account their symmetric structure. The computations of the various element matrices are independent and are distributed among the processors but the assembly involves some communication.

The second step solves the system of discrete time convolution equations constructed in the first step, using the algorithm described in Figure 3 and the above data structure. The parallel PCG solver is adapted to the sparse structure of the matrix  $M^0$ .

Table ?? gives the timings for a numerical test over a sphere with 1080 d.o.f. and  $l_{max} = 90$ . Table ?? gives the timing for a numerical test over a pyramid with 1296 d.o.f. and  $l_{max} = 90$ .

Table 5: Timings for n=1080 on a Cray T3D

# proc	total time	symbolic construction	numerical construction	solution time	convolution products	linear systems
16	211	5.7	167	37.7	20	9.5
32	123	2.8	87	29.1	9.6	8.6
64	65	39	46.6	15.8	4.99	7.5

Table 6: Timings for n=1296 on a Cray T3D

# proc	total time	symbolic construction	numerical construction	solution time	convolution products	linear systems
16	337	8.7	253	73	30	32
32	188	4.2	135	46	15	25
64	125	2.6	78	42	8.2	24

## 5 Conclusion

This work shows that high performance can be achieved in computational electromagnetics based on transient Boundary Integral Equation Method on MPP computers. This is crucial for application to high frequency problems which induce large size systems very expensive in memory space and computing time.

## References

- [1] A. Bachelot and A. Pujols, *Boundary Integral Equation Method in Time Domain for Maxwell's System*, in Computing Methods in Applied Sciences and Engineering, Nova Science Publishers, New York, 1991.
- [2] A. Bachelot and V. Lange, *Time Dependent Integral Method for Maxwell's System*, in Mathematical and Numerical Aspects of Wave Propagation SIAM, 1995.

- [3] M. Filipe, A. Forestier and T. Ha-Duong, *A time Dependent Acoustic Scattering Problem*, in *Mathematical and Numerical Aspects of Wave Propagation* SIAM, 1995.
- [4] P. A. Raviart and J. M. Thomas, *A mixed finite element method for 2nd order elliptic problems*, *Lecture Notes in Math.*, 606, Verlag, Berlin, 1975.