Implementation of a finite element and absorbing boundary conditions package on a par allel shared memory computer

Christian Vollaire, Laurent Nicolas

Implementation of a Finite Element and Absorbing Boundary Conditions Package on a Parallel Shared Memory Computer

C. Vollaire, L. Nicolas
CEGELY - UPRESA CNRS 5005 - Ecole Centrale de Lyon
BP 163 - 69131 Ecully Cedex - France

***Abstract*–A nodal-based finite element formulation coupled with absorbing boundary conditions has been developed to solve open boundary microwave problems. Only parallel computation enables to modelize large devices. We show in this paper how the code has been implemented on a parallel shared memory computer. Each step of the code is analyzed. Two types of storage for the matrix and two preconditioning methods for the conjugate gradient algorithm are particularly compared.**

***Index terms*–Finite element methods, parallel algorithms, shared memory systems, electromagnetic scattering.**

I. Introduction

We are dealing with the modeling of unbounded frequency domain microwave problems. The Finite Element (FE) formulation is directly written in terms of **E** or **H** vector field. 3D Enquist-Majda absorbing boundary conditions (ABC) are introduced to simulate open boundary domains [1]. This leads to 3 complex unknowns per node (6 degrees of freedom). To obtain a good accuracy, 10 nodes per wavelength are necessary, and a minimal distance equal to 1 wavelength needs to be ensured between the microwave device and the outer boundary. These requirements lead that only simple problems (10000 nodes) can be modelized on scalar workstations. Only parallel computation actually enables to modelize real large devices because it reduces computation time and mainly arranges enough memory.

The purpose of this paper is to show how to implement such a code on a CRAY C98 in order to obtain good parallel and vector performances. Speedup and vector performances are presented for the different steps of the program. All the performances are analyzed on a 60000 degrees of freedom test problem because the analysis of the parallel performances involves a monoprocessor computing. The example of an airplane illuminated by a plane wave is finally presented.

II. FE Formulation and Sequential Code

The weak Galerkin form of the entire formulation in term of total field **H** is given by (1), where $\mathbf{H}_i$ is the incident field.

$$\int_v \left[(\nabla W \times \frac{1}{\varepsilon_r} \nabla \times \mathbf{H}) + W k_0^2 \mu_r \mathbf{H} \right] dv - \int_v [(\nabla W)(\nabla \cdot \mathbf{H})] dv + \int_{S_{ext}+S_{pec}} W \mathbf{n} \nabla \cdot \mathbf{H} ds - \int_{S_{ext}} W g_{ABC}(\mathbf{H}) ds = \int_{S_{pec}} W \left[g_{ABC}(\mathbf{H}_i) - \mathbf{n} \times \nabla \times \mathbf{H}_i \right] ds \quad (1)$$

with $g_{ABC}(\mathbf{H}) = j k_0 \mathbf{H}_t - \frac{j}{2k_0} \nabla_t^2 \mathbf{H}_t$ the ABC.

Main steps of the sequential code are:

1-*Create the data structure of the global matrix*: because the global matrix is sparse, a compressed storage is used with only the non-zero terms.

2-*Assemble the global matrix*: each elementary matrix related to a volume FE is computed. Results are then distributed in the global matrix.

3-*Introduce ABC on the external boundaries, Boundary Conditions (BC) on conductors and on the symmetry planes.*

4-*Symmetrize the matrix*: because of the ABC, the FE matrix is non symmetric. It is approximately symmetrized by addition with its transposed.

5-*Solve the system of equations*: because the matrix system is sparse, the solving is performed using the Conjugate Gradient (CG) method.

III. Automatic Parallelism

The algorithms operating on monoprocessor calculator do not need to be entirely revised in order to work on a CRAY C98, which is a parallel shared memory computer. The code remains the same and runs quite immediately. Our code was first developed in a well structured way with many subroutine calls or error handling. Unfortunately, these points prevent the compiler from getting a good parallelization level. The compiler itself got only 17% of parallelization. Only simple loops such as initializations had been processed. Neither the matrix assembling nor the solver had gained parallelism. They only gained some vectorization. To obtain better results it is necessary to add manually compiler directives and to find the scope of the variables, as presented in the next sections.

IV. Matrix Representation

It has been shown previously that the matrix representation has a great influence on vectorial performances [3-4]. Two kinds of matrix representation have then been compared.

Manuscript received November 3, 1997.

C. Vollaire, vollair@trotek.ec-lyon.fr; L. Nicolas, laurent@trotek.ec-lyon.fr, http://cegely.ec-lyon.fr/.

This work was supported in part by the Institut du Developpement et des Ressources en Informatique Scientifique (CNRS).

A. Sparse row-wise matrix representation (storage #1): because the matrix is sparse and symmetric, only the non zero terms of its lower part are stored after its symmetrization [2].

B. Redundant Sparse row-wise matrix representation (storage #2): because the algorithms used in the solver need to access by column to the terms of the matrix, the entire FE matrix stays in memory even after its symmetrization. The memory space used is twice larger, but the access to the non zero terms of a column are adjacent in memory [3].

Sky line matrix representation has also been tested. Because all the terms contained in the bandwidth of the FE matrix are stored, this method is then too much memory consuming and cannot be used on very large devices.

V. Assembling the FE Matrix

A. The creation of the global matrix structure: this step prepares the storage of the non-zero terms of the matrix. It is very difficult to parallelize it because of many data dependencies. Furthermore, it decreases the parallel performances because it is especially sequential.

B. The assembling by elementary contributions: the elementary matrix related to a FE needs to have a private scope, and semaphores are used to avoid memory conflicts when the global matrix, stored in shared memory, is modified. So a parallel region is created which includes the loop on the finite elements. The global matrix is modified in a critical region, inside the parallel region. The code is executed in a sequential way in this region.

C. Introduction of BC on conductors and on the symmetry planes: due to the method used to introduce BC [2], a global modification of the FE matrix is required. Each processor performs a part of the modifications on the global matrix. Semaphores are used to avoid memory conflicts.

D. Symmetrization of the matrix: all the terms of each row are added with those corresponding of the transposed matrix. The external loop (rows) is parallelized while the internal one (columns) is vectorized. The variables used for this step can be shared because no memory conflict is possible.

Fig. 1 shows the speedups for the creation of the FE matrix (steps *A, B, C, D*) for both types of storage. Obviously the matrix representation has no influence.

VI. Diagonal Preconditioning Method

The diagonal preconditioning (DP) is the easiest preconditioning method to implement with the CG algorithm. The preconditioning is made by a vector-vector multiplication. So, this method requires a multiplication matrix-vector per iteration to compute the new residual vector. This step is parallelized by splitting the loop on the lines. When storage #1 is used, each processor computes a partial residual vector in private memory (parallel region).

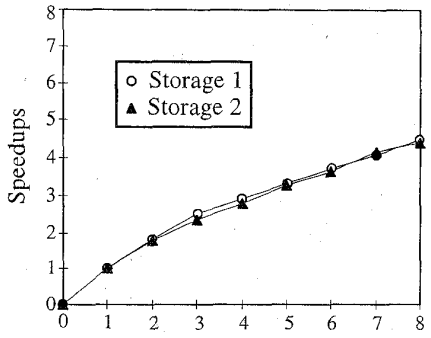

Fig. 1. Speedups for the assembling step (60000 degrees of freedom problem).

The addition of these partial results is performed in shared memory in a critical region (vectorization). When using storage #2, the multiplication is done in shared memory because the entire FE matrix is stored. Fig. 2 shows that the speedup is the same for both matrix representations.

VI. Incomplete Cholesky Preconditioning Method

The first step of this preconditioning method is the factorization of the FE matrix A in two matrices: $A = L \cdot L^t$. The building of the incomplete Cholesky matrix is performed by column [4] (fig. 3). The algorithm is given by (2). It is implicitly parallel because the L_{ij} terms can be computed independently once the diagonal term L_{jj} has been computed.

When using the storage #1, the computation of the terms located in the column j requires to search the rows i with a non zero term on the column j. This is expensive in CPU time. The parallelization is made by splitting this loop. Furthermore this algorithm requires the multiplication of the rows i and j. So an other search on the row i is necessary for all the terms of the row j to find the term of same rank. For this reason, the storage #1 is not adapted to the ICP. The vector performances obtained are low. However, only the lower part of the preconditioning matrix is built and only the lower part of A is stored in memory after its symmetrization: no additional allocation of memory is required.

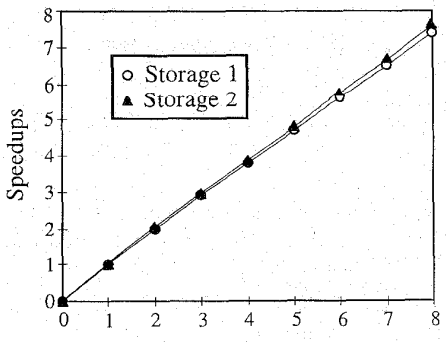

Fig. 2. Speedups for the CG with the diagonal preconditioning (60000 degrees of freedom problem).

For j = 1 to number of lines

$$L_{jj} = \sqrt{A_{jj} - \sum_{k=1}^{j-1} (L_{jk})^2}$$

For i = j+1 to number of lines

$$L_{ij} = \frac{1}{L_{ii}} (A_{ji} - \sum_{k=1}^{i-1} L_{ik} L_{jk}) \quad (2)$$

End

End

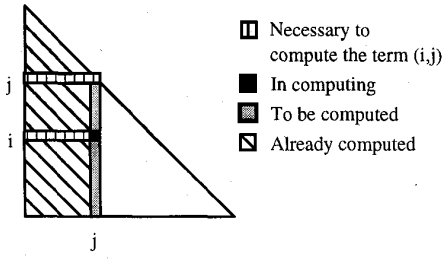

Fig. 3. Building of the incomplete Cholesky matrix per column.

The storage #2 allows to access directly to the number of rows i with a non zero term on the column j. It improves the vector performances of the code during the building of the preconditioning matrix. The entire preconditioning matrix is built to keep the advantage of the adjacent access to the terms of a same column (necessary for next step). The entire FE matrix is also stored in memory after its symmetrization. So this storage requires twice more memory than the storage #1.

From fig. 4, it appears that the parallel performances during the building of the incomplete Cholesky matrix do not depend on the storage method. On the other hand, it has only a great influence on vector performances (Table I), preventing the use of the storage #1 for large problems.

While solving using CG, both matrix-vector multiplication (parallelized as above) and forward-back substitutions steps are necessary. These last one have to be parallelized too. The system A.x=b is usually substituted by $(L.L^t).x=b$. L.y=b and $L^t.x=y$ are then computed (3).

Forward substitution

For i = 1 to number of lines (n)

$$yi = (bi - \sum_{k=1}^{k=i-1} Lik\ yk) / Lii$$

End (3)

Back substitution

For i = number of lines (n) to 1

$$xi = (yi - \sum_{k=i+1}^{k=n} Lki\ xk) / Lii$$

End

The forward substitution cannot be parallelized by splitting the loop on the row because of the back dependency. So, the produce of the terms $L_{ik}.y_k$ is parallelized if the number of operations remaining to perform is at least equal to the number of processors available. Otherwise the produce is vectorized. This strategy allows a good load balancing.

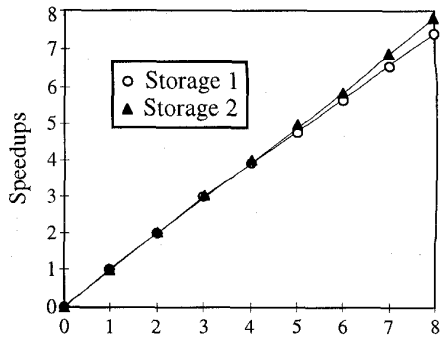

Fig. 4. Parallel performances for the building of the incomplete Cholesky matrix (60000 degrees of freedom problem).

Partial results are added in a critical region. Because the access to the terms L_{ik} is made by row, the vector performances are good. The back substitution is parallelized in the same way but the access by columns to the terms L_{ki} decreases the vector performances (Table I).

Figure 5 shows the speedups for forward-back substitutions depending on the matrix representation for the test problem. As previously, when using the storage #1, the search of the term L_{ki} is penalizing in term of vector performances. So the overhead introduced by splitting the loop to perform the produce $L_{ki}.x_k$ is negligible (good parallel performances). The use of the storage #2 involves less efficiency in term of parallel performances because the overhead introduced by the splitting is not negligible. On the other hand, the vectors performances are increased because the terms L_{ki} are adjacent in memory.

VII. Comparison Between Methods

Both convergence rate and vector-parallel performances have to be evaluated to compare the different methods. The test problem consists of the scattering of a plane wave by a perfect electric conducting (pec) cylinder. It is meshed with first order hexahedral, leading to 60000 degrees of freedom.

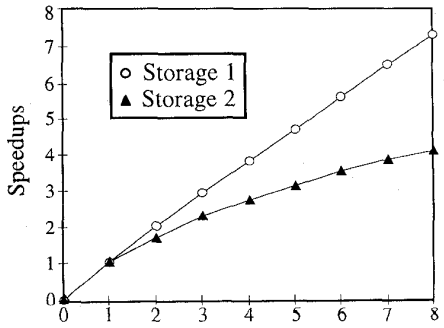

Fig. 5. Speedups for the back-forward substitution (60000 degrees of freedom problem).

The electromagnetic scattering of a plane wave by a perfect electric conducting airplane has also been modeled (fig. 7): a first problem is meshed with 33431 nodes and 201556 1st order tetrahedral, leading to 200586 degrees of freedom, and a second one is meshed with 51183 nodes and 308922 1st order tetrahedral, leading to 307098 degrees of freedom.

Fig. 6 shows the total speedups and Table I compares the vector performances, the memory used and the number of iterations needed to solve the test problem. For this small example, the DP with the storage #2 is the most efficient. Furthermore it does not increase the memory space needed. On the other hand, when solving large problems and when the matrix system has a poor conditioning rate, the ICP with the storage #2 is more efficient, as shown in Table II and in Table III. This is due to the reduction of the number of iterations. However, this method requires twice more memory space. The increase of the problem size tends to raise the vector performances of the ICP method up to an acceptable level.

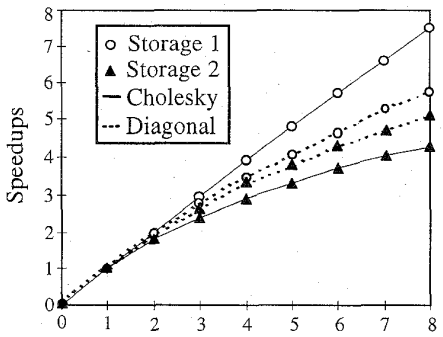

Fig. 6. Speedups for the entire code (60000 degrees of freedom problem)- incomplete Cholesky or diagonal preconditioning.

TABLE I
60000 DEGREES OF FREEDOM PROBLEM (HEXAHEDRAL)

		CPU time	Mflops	Memory	Iterations
DP	storage #1	263 s	19.3	18 Mw	232
	storage #2	187 s	40.1	18 Mw	232
ICP	storage #1	5314 s	2.5	18 Mw	76
	storage #2	733 s	10.2	34 Mw	76

TABLE II
200586 DEGREES OF FREEDOM PROBLEM (TETRAHEDRAL)

		CPU time	Mflops	Memory	Iterations
DP	storage #1	15357 s	52	28.5 Mw	32301
	storage #2	6654 s	151	28.5 Mw	32501
ICP	storage #1	too much CPU time consuming			
	storage #2	5648 s	98	42 Mw	10701

TABLE III
307098 DEGREES OF FREEDOM PROBLEM (TETRAHEDRAL)

		CPU time	Mflops	Memory	Iterations
DP	storage #1	23035 s	56	48 Mw	41507
	storage #2	9981 s	161	48 Mw	41773
ICP	storage #1	too much CPU time consuming			
	storage #2	8072 s	113	75 Mw	13667

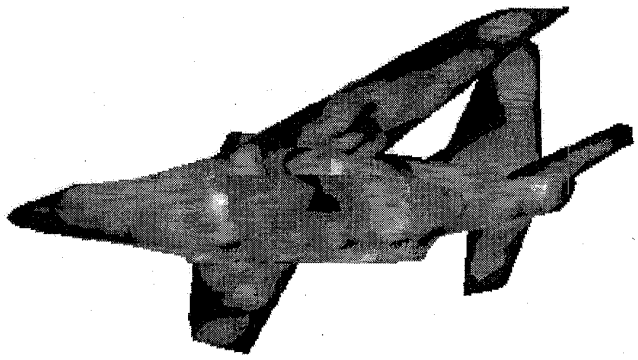

Fig. 7. Pec airplane illuminated by a plane wave at 0.3 GHz, magnitude of **H**.

IX. CONCLUSION

The adaptation to a shared memory computer is easy because it does not require a complete restructuring of the code. This remains the same and runs quite immediately. The CRAY compiler can try to parallelize and vectorize it automatically. However it only works on simple loops and leads to bad parallel performances. So, the programmer have to manually add parallelization and vectorization directives and to check the scope of every variable. This method allows to keep the control on the parallelism granulity.

The relative performances due to the vectorization are very weak with a classical matrix representation because the data streams are short: the matrix is sparse. The use of the redundant sparse row-wise matrix representation allows to obtain acceptable vector performances. However, this method requires twice more memory space.

It seems that MIMD with shared memory parallel computers have reached their limits in terms of memory space and number of interconnected processors. The constructors build now distributed memory architectures. The implementation of sequential codes on such computers is much more difficult because a complete restructuring of the algorithms is then necessary [2].

X. REFERENCES

[1] L. Nicolas, K. A. Connor, S. J. Salon, B. G. Ruth, L. F. Libelo, "Three dimensional FE analysis of high power microwave devices," *IEEE Trans. on Mag.*, no 2, March 93, pp. 1642-1645.

[2] C. Vollaire, L. Nicolas, and A. Nicolas, "Finite Elements Coupled with Absorbing Boundary Conditions on Parallel Distributed Memory Computer," *IEEE trans. on Mag*, vol. 33, no 2, pp. 1448-1451, March 1997.

[3] H. Margin, J.L. Coulomb, "Parallel and vectorial solving of finite element problems on a shared-memory multiprocessor," *IEEE Trans. on Mag.*, Vol.28, no 2, March 1992, pp. 1712-1715.

[4] H. Magnin, J.L. Coulomb, "A parallel and vectorial implementation of basic linear algebra subroutines in iterative solving of large sparse linear systems of equations," *IEEE Trans. on Mag.*, Vol.25, no 4, July 1989, pp. 2895-2897.