

# Parallel Method of Moments Analysis of Microstrip Structures Using the *MPI* Library

Eugenio Jiménez Yguácel  
 Departamento de Señales y Comunicaciones  
 Universidad de Las Palmas de Gran Canaria  
 Campus de Tafira S/N, Las Palmas 35017  
 Email: ejimenez@dsc.ulpgc.es

Francisco Cabrera Almeida  
 Departamento de Señales y Comunicaciones  
 Universidad de Las Palmas de Gran Canaria  
 Campus de Tafira S/N, Las Palmas 35017  
 Email: fcabrera@dsc.ulpgc.es

**Abstract**—In this paper we present a parallel Method of Moments (MoM for short) technique using the MPI library. Here, the MoM is used to analyze microstrip structures. The main goals to achieve are efficient parallel coefficient computation and efficient linear equation system solving. The efficiency and accuracy of the parallel-processing MoM code is analyzed through several examples.

## I. INTRODUCTION

The MoM [1] provides a numerical solution to linear equation problems. The form of the linear equation can be written as

$$F(g) = h \quad (1)$$

in which  $F$  is a linear operator,  $h$  is known, and  $g$  is to be determined. Let  $g$  be expanded in a series of functions  $g_1, g_2, g_3 \dots$  in the domain of  $F$

$$g \simeq c_1 g_1 + c_2 g_2 + \dots + c_N g_N = \sum_{n=1}^N c_n g_n \quad (2)$$

where the  $c_n$  are constants. The  $f_n$  are called *expansion functions* or *basis functions*. Substituting (2) into (1) and using the linearity of  $F$ , one has

$$\sum_{n=1}^N c_n F(g_n) = h \quad (3)$$

If (3) represents an approximate equality, then the difference between the exact and approximate equation is

$$R = \sum_{n=1}^N c_n F(g_n) - h \quad (4)$$

which is called the *residual*  $R$  and has to be minimized. Now define a set of *testing functions* or *weighting functions*,  $w_m$ , in the range of  $F$ . Take the inner product between  $R$  and  $w_m$  and set all the weighted residuals equal to zero.

$$\langle w_m | R \rangle = 0 \quad (5)$$

Finally, substitute (4) into (5) to obtain

$$\sum_{n=1}^N c_n \langle w_m | F(g_n) \rangle = \langle w_m | h \rangle \quad m = 1 \dots N \quad (6)$$

This set of equations can be written in matrix form as

$$[F_{mn}] [c_n] = [h_m] \quad (7)$$

in which  $F_{mn}$ ,  $c_n$  and  $h_m$  can be written as

$$[F_{mn}] = \begin{bmatrix} \langle w_1 | F(g_1) \rangle & \dots & \langle w_1 | F(g_N) \rangle \\ \langle w_2 | F(g_1) \rangle & \dots & \langle w_2 | F(g_N) \rangle \\ \vdots & & \vdots \\ \langle w_N | F(g_1) \rangle & \dots & \langle w_N | F(g_N) \rangle \end{bmatrix} \quad (8)$$

$$[c_n] = \begin{bmatrix} c_1 \\ c_2 \\ \vdots \\ c_N \end{bmatrix} \quad [h_m] = \begin{bmatrix} \langle w_1 | h \rangle \\ \langle w_2 | h \rangle \\ \vdots \\ \langle w_N | h \rangle \end{bmatrix} \quad (9)$$

If  $[F]$  is nonsingular, its inverse exists, and  $[c]$  is given by

$$[c_n] = [F_{mn}]^{-1} [h_m] \quad (10)$$

## II. MOM AND MICROSTRIP GEOMETRIES

In microstrip geometries, the Electric Field Integral Equation (EFIE for short) can be written as

$$\bar{E}^i(x, y) + j\omega\mu_0 L \left[ \bar{G}_{EJ}(x, y | x', y') \bar{J}_S(x', y') \right] = 0 \quad (11)$$

Let  $\hat{x}$  and  $\hat{y}$  components of  $\bar{J}_S$  be expanded in a series of *basis functions*  $J_{x_j}$  and  $J_{y_j}$

$$J_{Sx} = \sum_{j=1}^N A_{x_j} J_{x_j}(x', y') \quad (12)$$

$$J_{Sy} = \sum_{j=1}^N A_{y_j} J_{y_j}(x', y') \quad (13)$$

Substituting (12) and (13) into (11), the discretized EFIE is written as

$$E_x^i = -j\omega\mu_0 \left( \sum_{j=1}^N A_{x_j} L_{E_{xx}} [J_{x_j}] + \sum_{j=1}^N A_{y_j} L_{E_{xy}} [J_{y_j}] \right) \quad (14)$$

$$E_y^i = -j\omega\mu_0 \left( \sum_{j=1}^N A_{x_j} L_{E_{yx}} [J_{x_j}] + \sum_{j=1}^N A_{y_j} L_{E_{yy}} [J_{y_j}] \right) \quad (15)$$

Basis functions  $J_{x_j}$  and  $J_{y_j}$  are chosen as the product of two functions

$$J_{x_j}(x', y') = T_j(x')Q_j(y') \quad (16)$$

$$J_{y_j}(x', y') = Q_j(x')T_j(y') \quad (17)$$

and testing functions are chosen to be the same as basis functions (Galerkin method). Finally, after setting all the weighted residuals equal to zero, one has

$$-j\omega\mu_0 \left( \sum_{j=1}^N A_{x_j} \langle L_{E_{xx}} [T_j(x')Q_j(y')] | T_i(x)Q_i(y) \rangle + \sum_{j=1}^N A_{y_j} \langle L_{E_{xy}} [Q_j(x')T_j(y')] | T_i(x)Q_i(y) \rangle \right) = \langle E_x^i | T_i(x)Q_i(y) \rangle \quad i = 1 \dots N \quad (18)$$

$$-j\omega\mu_0 \left( \sum_{j=1}^N A_{x_j} \langle L_{E_{yx}} [T_j(x')Q_j(y')] | Q_i(x)T_i(y) \rangle + \sum_{j=1}^N A_{y_j} \langle L_{E_{yy}} [Q_j(x')T_j(y')] | Q_i(x)T_i(y) \rangle \right) = \langle E_y^i | Q_i(x)T_i(y) \rangle \quad i = 1 \dots N \quad (19)$$

Again, this set of equations can be written in abbreviated matrix form as

$$\begin{pmatrix} Z_{xx}^{ij} & Z_{xy}^{ij} \\ Z_{yx}^{ij} & Z_{yy}^{ij} \end{pmatrix}_{2N \times 2N} \begin{pmatrix} I_x^i \\ I_y^i \end{pmatrix}_{2N \times 1} = \begin{pmatrix} V_x^j \\ V_y^j \end{pmatrix}_{2N \times 1} \quad \begin{matrix} i = 1 \dots N \\ j = 1 \dots N \end{matrix} \quad (20)$$

where each one of the elements of the  $Z$  matrix in (20) can be written as a convolution integral

$$\int dx' \int dx \int dy' \int F_j(x', y') F_i(x, y) G_\alpha(x - x', y - y') dy \quad (21)$$

where  $F_j$  and  $F_i$  are basis and testing functions and  $G_\alpha$  are components of the microstrip Green function (Sommerfeld integrals). An in-house developed *Sommerfeld* library [2] is used to compute these  $G_\alpha$  functions.

### III. PARALLEL MOM TECHNIQUE

In many MoM problems, the matrix fill time is the dominant factor, as each matrix element typically involves the computing of a numerical integral of a complex function. Usually a  $2N$  fold numerical quadrature of a complex function has to be computed, being  $N$  the geometry dimension (vg  $N = 1$  for wires,  $N = 2$  for planar geometries and  $N = 3$  for 3-D geometries). This type of problem is typically easy to parallelize as the computation of any matrix element is completely independent of the value of any other matrix element.

As structures become electrically large, the matrix factorization begins to dominate the overall solution time. The memory and computation requirements for these large structures can

become daunting as grow as  $O[N^3]$  being  $N$  the number of unknowns

In a fully parallelized MoM, the  $F$  matrix is never stored into the memory of any processor but blocks of it. Therefore, filling and factorization steps are closely linked.

#### A. Parallelized scheme

Now we are going to distribute the computation and the factorization of the  $Z$  matrix among a set of  $nproc$  processors. There is a main processor which collects the geometry data input, distributes the work and collects and stores final data. Each of the other processors, including the main one, computes and factorizes one block of the  $Z$  matrix. This is shown in figure 1

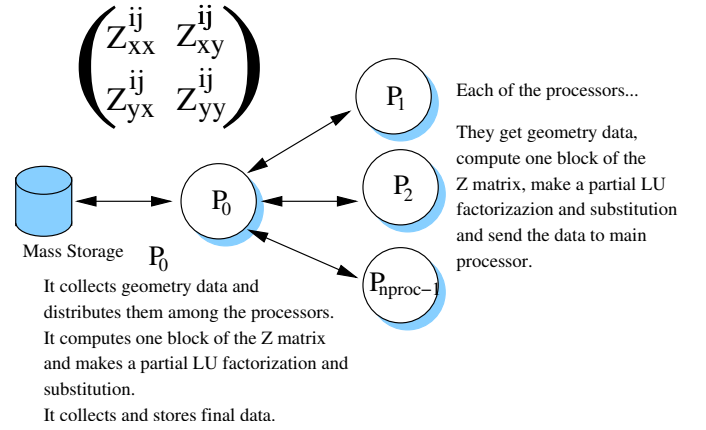


Fig. 1. Work distribution among processors

#### B. The MPI paradigm

One of the most successful parallel computational models is the message-passing model. This model posits a set of processors that have only local memory but are able to communicate with other processes by sending and receiving messages. *MPI* [3], which stands for Message Passing Interface, is an attempt to collect the best features of the message-passing systems that have been developed over the years, improve them when appropriate, and standardize them. It is a library not a language. It specifies the names, calling sequences and results from functions to be called from Fortran/C/C++ programs. *MPI* is a specification not a particular implementation. A correct *MPI* program should be able to run on all *MPI* implementations without change.

A minimal message interface between two processes should be built using two primitives: send and receive.

For the sender, the things that must be specified are the data to be communicated and the destination process to which the data is to be sent. The minimal way to describe data is to specify a starting address and a length (in bytes) and a destination field (usually an integer)

On the receiver's side, the minimum arguments are the address and length where received data is going to be placed and a variable to be filled in with the identity of the sender.

Although this minimum interface could be adequate for some applications, one key notion is missing; *matching*. A process must be able to control which messages it receives. This is the type or *tag* of the message. Finally it is useful for the receiver to specify a maximum message size (the actual length) for a given *tag*. Therefore, our minimal message interface has become

```
send(address, length, destination, tag)
and
receive(address, length, source, tag, actlen)
That basic interface is implemented in MPI using MPI_send
and MPI_receive
```

### C. Matrix filling and factorization

The way the  $Z$  matrix is filled depends on the way it is factorized. The programmer always tries to minimize communications between processors and to equalize the load among them. We have studied several ways to factorize the matrix and each one of those ways is linked to a matrix filling scheme. In this paper we are going to show only one scheme: the cyclic two-dimensional data distribution.

In this scheme, each processor computes and, partially, factorizes (LU) a block of the  $Z$  matrix. Those blocks are chosen so for a given pair of indexes  $(i, j)$  the four parameters  $Z_{xx}^{ij}$ ,  $Z_{xy}^{ij}$ ,  $Z_{yx}^{ij}$  and  $Z_{yy}^{ij}$  are computed in the same processor and there are no communication among processors. Obviously, while factorizing the matrix, processors need to share information among them. To equalize the load among processors, they work in a cyclic way as it is shown in the example of figure 2. Due to data dependencies in LU factorization, there are processors that have to wait for other processors to finish. That idle time can be minimized if each processor works with a relatively small matrix block.

$P_0$	$P_1$	$P_2$	$P_3$	$P_0$	$P_1$	$P_2$	$P_3$	$P_0$	$P_1$	$P_2$	$P_3$
$P_4$	$P_5$	$P_6$	$P_7$	$P_4$	$P_5$	$P_6$	$P_7$	$P_4$	$P_5$	$P_6$	$P_7$
$P_8$	$P_9$	$P_{10}$	$P_{11}$	$P_8$	$P_9$	$P_{10}$	$P_{11}$	$P_8$	$P_9$	$P_{10}$	$P_{11}$
$P_{12}$	$P_{13}$	$P_{14}$	$P_{15}$	$P_{12}$	$P_{13}$	$P_{14}$	$P_{15}$	$P_{12}$	$P_{13}$	$P_{14}$	$P_{15}$
$P_0$	$P_1$	$P_2$	$P_3$	$P_0$	$P_1$	$P_2$	$P_3$	$P_0$	$P_1$	$P_2$	$P_3$
$P_4$	$P_5$	$P_6$	$P_7$	$P_4$	$P_5$	$P_6$	$P_7$	$P_4$	$P_5$	$P_6$	$P_7$
$P_8$	$P_9$	$P_{10}$	$P_{11}$	$P_8$	$P_9$	$P_{10}$	$P_{11}$	$P_8$	$P_9$	$P_{10}$	$P_{11}$
$P_{12}$	$P_{13}$	$P_{14}$	$P_{15}$	$P_{12}$	$P_{13}$	$P_{14}$	$P_{15}$	$P_{12}$	$P_{13}$	$P_{14}$	$P_{15}$
$P_0$	$P_1$	$P_2$	$P_3$	$P_0$	$P_1$	$P_2$	$P_3$	$P_0$	$P_1$	$P_2$	$P_3$
$P_4$	$P_5$	$P_6$	$P_7$	$P_4$	$P_5$	$P_6$	$P_7$	$P_4$	$P_5$	$P_6$	$P_7$
$P_8$	$P_9$	$P_{10}$	$P_{11}$	$P_8$	$P_9$	$P_{10}$	$P_{11}$	$P_8$	$P_9$	$P_{10}$	$P_{11}$
$P_{12}$	$P_{13}$	$P_{14}$	$P_{15}$	$P_{12}$	$P_{13}$	$P_{14}$	$P_{15}$	$P_{12}$	$P_{13}$	$P_{14}$	$P_{15}$

Fig. 2. Cyclic two-dimensional data distribution. 16 processors, 3 cycles

To maximize per processor performance, there is an optimum block size which depends on the matrix size and processors' cache memory. A block size smaller than optimum means more communications and less computations.

A block size greater than optimum means more out-of-core computations and less speed.

In figure 3 pseudocode to run in each processor is shown.

```
do  $c_{col} = 1, \dots, ncycles_{col}$ 
   $c_{row} = 1$ 
  do  $l = 1, \dots, id(proc_{col}, c_{col})$ 
    Update rows( $id(proc_{row}, c_{row})$ )
    Get pivots( $id(l, c_{col})$ )
    LU( $l$ ) and V( $l$ ) factorization
  end do
  if ( $id(proc_{row}, c_{col}) \leq id(proc_{col}, c_{col})$ ) then
    do  $k = 1, \dots, ncol(id(proc_{col}, c_{col}))$ 
      Swap rows( $id(proc_{row}, c_{row})$ )
      LU( $k$ ) and V( $k$ ) factorization
      Send column( $k$ ) to the rest of  $id(proc_{row}, c_{col})$ 
    end do
    Send V data to  $id(proc_{col} + 1, c_{col})$ 
  end if
  if ( $id(proc_{row}, c_{col}) = id(proc_{col}, c_{col})$ )  $c_{row} = c_{row} + 1$ 
  if ( $c_{col} = nciclos_{col}$ ) then
    if ( $id(proc_{row}, c_{col}) > id(proc_{col}, c_{col})$ ) then
      do  $l = id(proc + 1, c_{col}), \dots, nproc_{col}$ 
        Swap rows ( $id(proc_{row}, c_{row})$ )
      end do
    else
      do  $l = id(proc_{row} + 1, c_{col}), \dots, nproc_{col}$ 
        Update rows( $id(proc_{row}, c_{row})$ )
        Get pivots( $id(l, c_{col})$ )
        LU( $l$ ) and V( $l$ ) factorization
      end do
    end if
  end do
```

Fig. 3. Pseudocode for parallel LU factorization

Here  $ncycles_{col}$  is the number of cycles per column (3 in figure 2),  $proc_{col/row}$  is the position of a processor in a row or a column (it varies between 1 and 4 in figure 2) and  $id(proc_{col/row}, cycle_{col/row})$  identifies the position in the matrix of processor  $proc_{col/row}$  in cycle number  $cycle_{col/row}$ .

1) *ScaLAPACK, BLAS and linear algebra software*: This pseudocode only works if  $ncycles_{col} = ncycles_{row}$  and the number of processors is a perfect square (as in example shown in figure 2). Dealing with a non perfect square number of processors complicates the code quite a lot. Instead of re-coding the algorithm, we looked for mature well established code that was able to help us.

*ScaLAPACK* [4] is an optimized library of linear algebra subroutines that run in cluster environments. Its main components are: a low level BLAS optimized for single processor, a communication library for BLAS subroutines (BLACS) that uses some message passing model (MPI or PVM) and a parallel version of BLAS library (PBLAS) that uses the aforementioned libraries.

*ScaLAPACK* uses a two-dimensional cyclic data distribution that helped us to easily map our matrix data distribution to

*ScaLAPACK's* one.

#### IV. RESULTS

In this section, we investigate the efficiency of the proposed parallel processing. The evaluation was carried out on the GIC cluster "Maxwell". The cluster details are as follows:

- Number of processors: 23 Intel Pentium IV Prescott 2.8 GHz, 1GB RAM
- Network: 100 Mb/s
- Operating system: Linux 2.6 kernel series, Bproc based cluster
- Tools: GNU tools (gcc, g77,...)
- MPI: LAM-MPI
- BLAS: ATLAS

##### A. Parallel processing efficiency analysis

The efficiency of parallel-processing code depends not only on how we develop the code, but also on the computer hardware and networking equipment. We employ the conventional definition of scalability or speedup of the parallel processing code as

$$S_p = \frac{T_s}{T_p} \quad (22)$$

where  $T_s$  is the simulation time when the entire problem is simulated using a single processor, and  $T_p$  is the simulation time in the  $p$ -processors parallel processing. During the simulations, we had to define a new figure of merit that we called *parallel speedup*. It is used when the entire problem does not fit in one single processor. It is defined as

$$PS_p = \frac{T_{1c}}{T_p} \quad (23)$$

where  $T_{1c}$  is the simulation time using  $p$  processors but the data distribution has only one cycle per row and column (the worst case).

In figure 4, speedup is shown vs block size and for three processor grids (16 processors). The square arrangement is clearly better than the other two.

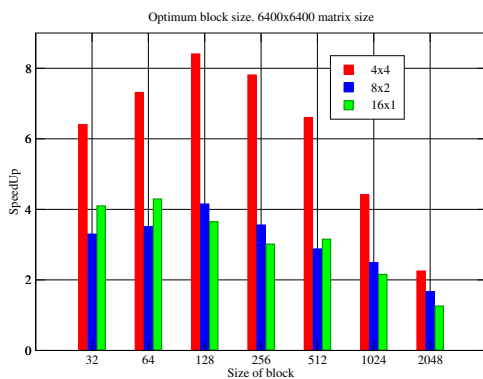


Fig. 4. Optimum block size and speedup for three processor grids

In figure 5, parallel speedup vs block size is shown for three matrix sizes and for a 5x4 processor grid.

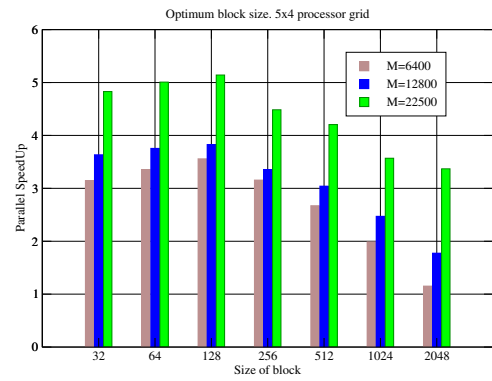


Fig. 5. Optimum block size and parallel speedup vs matrix size

Figure 6 shows the Kiviat graph for optimum block size, 4x4 processor grid, and  $M=22500$  (45000x45000 matrix size). The Kiviat graph shows the processor load and a nearly circular shaped graph means the load is balanced among processors

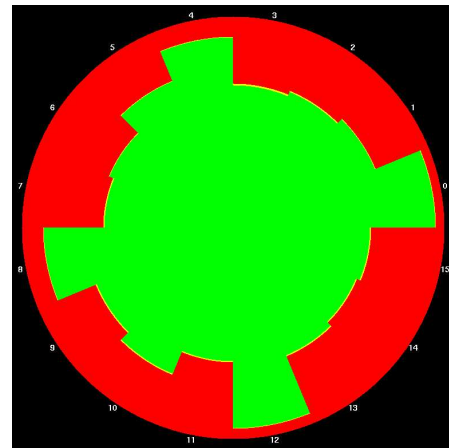


Fig. 6. Kiviat graph

#### V. CONCLUSION

We have presented a parallel MoM analysis of microstrip structures. Our main goal has been to improve the efficiency of the process modifying two parameters: processors grid topology and optimum block size. Some code has been in-house developed and later modified to suit to the *ScaLAPACK* standard library.

#### ACKNOWLEDGMENT

This work has been partially supported by the R&D Spanish National Projects TEC2004-09615-C03, TEC2005-08377-C03 and TEC2005-07010-C02

#### REFERENCES

- [1] Roger F. Harrington "Matrix Methods for Field Problems", *Proc IEEE*, vol. 55, no 2, pp. 136-149, February 1967
- [2] E. Jimenez, F.Cabrera. "Sommerfeld: a library for computing Sommerfeld integrals", *IEEE Ant. and Prop. Symposium*, Baltimore, pp. 966-969, July 1996.
- [3] <http://www.mpi-forum.org>
- [4] <http://www.netlib.org/scalapack>