Distributed and parallel version of the FDTD simulation algorithm

Giuseppe Amenta, Grazia Lo Sciuto[1]
[1]Department of Electrical, Electronic and Informatics Engineering
University of Catania
Catania, Italy
glosciuto@dii.unict.it

***Abstract*—This paper proposes an innovative algorithm for running in a distributed and parallel fashion the well-known Finite-Difference Time-Domain method. Given the dependence among data, for the proposed distributed version, care has been taken in the approach splitting the involved data. This is to avoid a great deal of data exchange among different hosts, so as to minimize the communication overhead. Performance comparisons between the proposed solution and the sequential one has shown that improvements can be achieved using the proposed version running on several hosts. Moreover, the distributed version let overcome the problem of central memory exhaustion, typical of this method, due to the large matrices to be handled.**

***Index Terms*—FDTD, parallelization, distribution, algorithms, data dependence**

I. Introduction

The FDTD Finite-Difference Time-Domain method [15], [19], [13] provides, in its initial formulation, the discretization of fields on two interlaced orthogonal Cartesian grids. This method is expressed as an integral formulation describing the electromagnetic field by means of state variables, which are defined as integral quantities associated to well-defined geometric elements of the dual grids (closed path integral along the lines and flows through the corresponding surfaces) [8], [7].

The original formulation consists of a sequential algorithm executing a possibly huge number of iterations on a single core of a host. Distribution is highly desirable for solving the typical problems on the domain under investigations, i.e. electromagnetic fields, etc., due to the high amount of memory used and computation necessary. However, a distributed version of the algorithm has to minimize communication overhead due to data exchange among hosts. When the distributed version is available, then an underlying platform can be put to use for achieving real parallelism and distribution [1], [2], [12], [16].

A proper analysis of the data dependencies among the several parts of the initial sequential algorithms is therefore needed to achieve a distributed version minimising data exchanges. In this field, several automatic or semi-automatic techniques can assist in finding structural and data dependencies [5], [10], [11], [14], [9], [4]. In order to find our proposed solution, we have put to use such techniques as well as manually devising our distribution approach.

This paper analyses the most important aspect of distributing the typical FDTD algorithm and shows the performance improvements when the distributed version is used. The following sections of the paper describe, firstly the mathematical background, secondly the sequential version of the algorithm, thirdly an initial optimization for the algorithm, fourthly the distributed and parallel version. Finally, conclusions are drawn.

II. Background

The technique called FIT (Finite Integration Theory [3]) represents a numerical method which is implemented by electromagnetic simulation software systems (such as CST studio suite [18]). Such a technique has been applied to arbitrary grids and corresponds to the FDTD on orthogonal hexagonal grids, when the time derivative is discretized by the leap-frog algorithm. The resulting algorithm can be further developed for taking advantage of parallelization. We have explored the equations of the FDTD algorithm with the integral variables. We take for simplicity the one-dimensional case (this can be obviously extended) of a linearly polarized electromagnetic wave in x-direction, propagating along the z direction. The equations relevant to the discussion here are Maxwell's curl equations, as:

$$\begin{aligned} -\partial_t B &= \nabla \times E \\ \partial_t D &= \nabla \times H \end{aligned}$$

These equations can be integrated on dual meshes at interlaced time intervals, based on the leap-frog scheme as shown in Fig. 1.

This method leads to the following equations, representing an iterative scheme.

$$\begin{aligned} \tilde{D}_x^{n+\frac{1}{2}}(k) &= \tilde{D}_x^{n-\frac{1}{2}}(k) - \frac{\Delta t\, c_0}{\Delta z}[H_y^n(k+\frac{1}{2}) - H_y^n(k-\frac{1}{2})] \\ \tilde{E} &= \frac{1}{\varepsilon_r}\tilde{D} \\ \tilde{B}_y^{n+1}(k+\frac{1}{2}) &= \tilde{B}_y^n(k+\frac{1}{2}) - \frac{\Delta t\, c_0}{\Delta z}[\tilde{E}_x^{n+\frac{1}{2}}(k+1) - \tilde{E}_x^{n+\frac{1}{2}}(k)] \\ \tilde{H} &= \frac{1}{\mu_r}\tilde{B} \end{aligned}$$

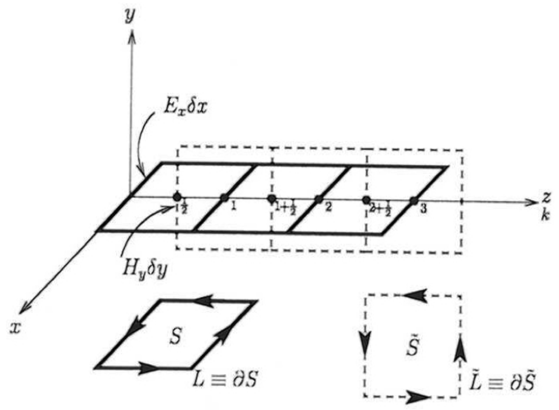

Fig. 1. Discretization in one-dimension with the dual grids scheme. $S = \delta x \times \delta z$, $\tilde{S} = \delta y \times \delta z$, where δx and δy are the discretization steps.

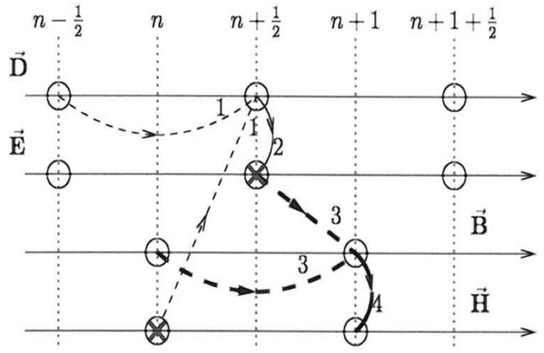

Fig. 2. Leap frog scheme. Time interlacing of the fields E, D, H and B in the FDTD method.

The above iterative equations are updated (time marching) as shown in the leap-frog scheme of Fig. 2. The steps marked in Fig. 2 with 2 and 4 are relative to the costitutive relations of the fields and are punctual operations. While the steps marked with 1 and 3 are relative to the Maxwell's curl equations (the data used for these updates are located in adjacent cells).

Of course, an electromagnetic wave can not propagate at a speed higher than the light velocity. Therefore, the time step must not exceed the time that the wave would take to propagate between 2 points on the grid. In 2D the wave propagation direction changing the traveling distance must be taken into account (see Fig. 3 and 4).

In 3D in order to implement the iterative scheme we use the Yee cell that is shown in the Fig. 4.

A. The Gaussian Pulse

A continuous Gaussian pulse is defined as:

$$f_g(t) = e^{-(\frac{t-d_g}{w_g})^2} \tag{1}$$

Where the d_g component represents the time delay, while the w_g parameter generically is the width of the pulse. The corresponding peak value is obtained for $t = d_g$. If we express the dg delay and the w_g pulse width, depending on the time step Δt, we have the discrete version of the previous equation:

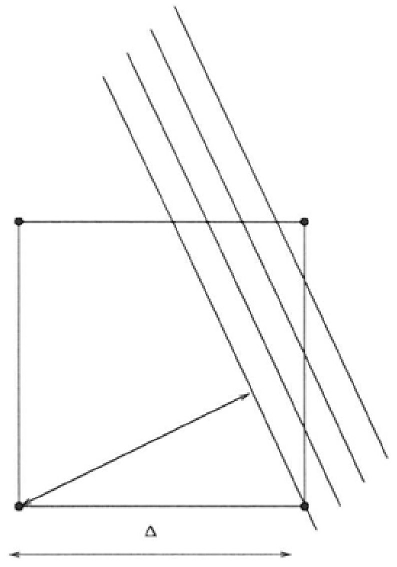

Fig. 3. Incidence directions in 2D.

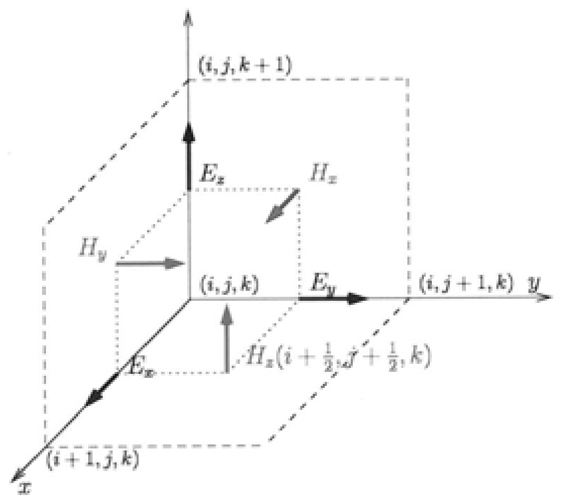

Fig. 4. The Yee cell.

$$f_g(n\Delta t) = f_g(n) = e^{-(\frac{n-m}{p})^2} \tag{2}$$

In the Gaussian pulse equation in the discrete version (2), the Δt temporal time is not included explicitly.

The time-discrete version of the equation is appropriate to be translated into an algorithm which can be computed for a given set of data. Finally, it is necessary to take into account other features of the physical problem, that is the following three constraints.

1) The electric boundary condition, since in the internal points of a perfect conductor the electric field $\mathbf{E}$ vanishes.
2) The magnetic boundary condition, which can ba easily handled if the computational domain stops at magnetical nodes.

3) The radiation condition, for the ability to absorb waves without reflection.

III. Sequential Implementation

The study of the physical phenomenon and its mathematical functions lead us to outline its implementation. The algorithm takes as input several fundamental values for the study of this phenomenon, including 5 matrices, initialized to the needed values for the computation, and initialized in an optimized way (see the following). Once values are received, the algorithm executes some checks and then reaches the 4 fundamental *for* loops, and the *pulse* variable calculation.

The inputs for the algorithm are:

1) the initial matrices `ga[][], dz[][], ez[][], hx[][] and hy[][]`, containing the necessary values required to perform the calculation;
2) the number of steps *nsteps* to be performed, expressing the number of times that the algorithm has to be executed;
3) *T* the absolute time, from which the study of the physical phenomenon starts;
4) the source parameter t_0, essential for calculating the *pulse* variable (usually set to 20.0);
5) the *spread* of the source, needed for calculating the pulse variable (set to 6.0).

There are two control loops, an external *while* and an inner *for*. The first external loop checks whether or not the condition that ends the execution of the algorithm (value 0 or a negative number) has been reached. It is a condition taken as an input.

The second loop checks the number of nsteps that have to be executed by the algorithm. It also responsible for increasing the *T* absolute time variable, which measures the time from the zero instant of the beginning of the experiment until to the instant n where the experiment ends, and also is used to compute the electromagnetic values. The time variable is a *float*, since time flows with steps of 0.5.

The subsequent 4 *for* loops compute 4 of the 5 starting matrices (`ga[][]` is constant). The first loop computes `dz[][]` matrix, using `hy[][]` and `hx[][]` matrices, multiplied by the constant 0.5. With respect to the elements of matrix `hy[][]` to be computed, the previous row of matrix `hy[][]` and the next column of `hx[][]` have to be made available. The dependence from such data has great implications for the transformation of the algorithm into its distributed version.

Then, element by element there will be some additions to compute the `dz[][]` matrix, as in the following:

```
dz[i][j] = dz[i][j] + 0.5*(hy[i][j] -
hy[i-1][j] - hx[i][j] + hx[i][j-1]);
```

Before the second *for* loop, a block of instructions sets the pulse variable according to time T. After that, the resulting value is assigned to a given portion of `dz[][]`, previously calculated. This is said the Gaussian impulse of the source. The corresponding code is as follows:

```
pulse = exp(-0.5*(pow((t0-T)/2.0)));
dz[IC][JC] = pulse;
```

The second *for* loop calculates matrix `ez[][]`, by means of the matrices `ga[][]` and `dz[][]`, (the second matrix is updated by the first *for* loop and the gaussian impulse). The code is as follows:

```
ez[i][j] = ga[i][j] * dz[i][j];
```

The third *for* loop calculates the matrix `hx[][]`, adding it to the multiplication between the constant value 0.5 and the updated `ez[][]` matrix. The next column of the matrix is required (with further implications, as in the previous cases). The relative code is:

```
hx[i][j] = hx[i][j] + 0.5 * (ez[i][j] -
ez[i][j+1]);
```

Finally, the fourth *for* loop calculates the matrix `hy[][]`, like in the third loop, by using matrix `hy[][]`, the constant value 0.5 and the updated matrix `ez[][]`. The corresponding code is:

```
hy[i][j] = hy[i][j] + 0.5 * (ez[i+1][j]
- ez[i][j]);
```

The said algorithm has been developed in *C++*, and even the sequential version has better performances than some corresponding commercial software systems. The simulation times for our developed tool were significantly lower than those required by simulations–using the same hardware. This reduction is extremely advantageous when complex electromagnetic structures are studied.

IV. Dynamic vs Static Initialization

Firstly, it is required to initialize the 5 matrices that represent the electromagnetic magnitudes under study. They are: `ga[][], dz[][], ez[][], hx[][], hy[][]`. Both the size of the matrices and the values they have to have, will be given as input to the algorithm. In particular, the size of all matrices must be identical between them, while the values can be different. This derives from the mathematical function used to study the physical phenomenon. Such values can be given in a configuration file, in order to make the program more versatile and suitable for the investigation of the phenomenon from different starting conditions.

The dynamic initialization of the 5 starting matrices is provided through the *initMatrix* function, which has as input the matrix to initialize, its size, and the initial value that the matrix elements must take. These elements are float types. *initMatrix* function consists of 2 *for* loops, one *for* loop scans the rows and one *for* loop scans the columns. Within the 2 *for* loops, it is performed an indexed assignment of the matrices to the desired value. It can be expected that a small part of the

TABLE I
INITIALIZATION VALUES RECOMMENDED FOR STUDYING THE PHYSICAL PHENOMENON.

Matrices	Initialization Values
`ga[][]`	1.0
`dz[][]`	0
`ez[][]`	0
`hx[][]`	0
`hy[][]`	0

TABLE II
EXECUTION AND COMPILATION TIMES OF THE STATIC AND DYNAMIC VERSIONS.

	Static Initialization			Dynamic Initialization	
Dim.	Compiling	Execution	Executable	Compiling	Execution
50	6.326 s	2.142 s	1.5 Mb	0.712 s	2.102 s
70	15.843 s	2.995 s	4.1 Mb	0.495 s	3.106 s
80	95.257 s	5.591 s	6.0 Mb	0.453 s	5.649 s

array will be set to a starting value other than the rest (which is usually 0 or 1.0).

A performance comparison between the proposed dynamic initialization and a static initialization of the matrices is shown in table II. Compilation and execution average times are given for both versions.

If the matrix size increases, a slight improvement in performances of the static initialized matrices has been observed. This result is apparently positive, considering the big size typical of the matrices, however it only affects about 0.7% of the initialization time, and would depend on the matrix size. In addition, this approach has several limits, some of which are really considerable. A brief list of such limits is reported below:

1) The compilation time, which must be actually added to the runtime program for each run, is in percentage really high compared to the run time (due to the huge amount of data needed for pre-storage). This consideration, combined with the fact that data are not constant, but taken as input, will be such that this time has to be considered whenever the program is executed.
2) The limit on the number of elements that can be compiled, which causes a virtual memory consumption error (depending on the host, and for a standard host being around 85^3 elements).
3) The dependence on a *c++* compiler (or *g++*) that each host must have installed on itself to let the program work.
4) The inability to create an executable version of the software system, which in fact reduces the user-friendliness for less experienced users.

To further improve the dynamic initialization performance, we have used pointers to access matrix elements, since the arithmetic pointer is faster than array indexing.

```
k = rows * columns;
for (i = 0; i <k; i++)
*(matrix + i) = value;
```

The *matrix* variable (a float pointer) holds the one-dimensional array representing the matrix; the integers *i* and *j* are used to scan the matrix; the *row* and *column* are *int* variables holding the total number of rows and columns; and finally the *value* contains the *float* value of the cells in the array, given to *initMatrix* function. Further tests indicated an improvement in performance, hence reducing the slight gap between the two approaches.

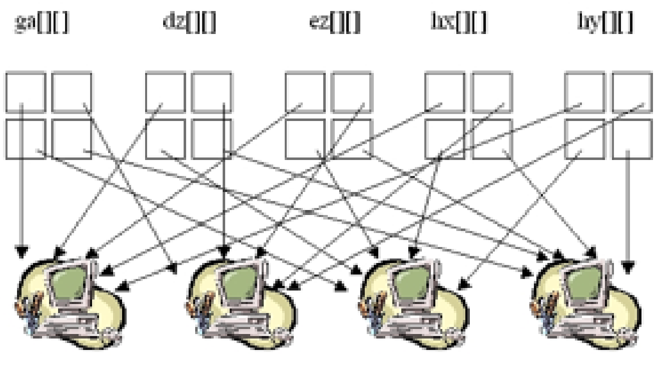

Fig. 5. Example of the split into 4 sub-matrices, for the 5 initial matrices (ga[][], dz[][], ez[][], hx[][] and hy[][]).

V. PARALLEL AND DISTRIBUTED VERSION

By distributing the 5 starting matrices on multiple hosts it is possible to parallelize and distribute execution, overcome the physical limits of a single host (large amount of data in volatile memory, etc.), and speed up the processing [6], [17]. The conceived solution splits the matrices into *x* submatrices, and distributes them into the *x* hosts available for the processing. The results on each host should then be sent to a server collecting and showing them.

The intrinsic structure of the implementation (see the above description of the sequential version), limits the distribution approach. In order to compute matrices (i.e. `dz[][]`, `hx[][]` and `hy[][]`) some rows or columns are needed from, previous or successive, other matrices (`hy[][]`, `hx[][]` and `ez[][]`). The parallelism of the problem at hand can be seen according to the producer-consumer model. Once the row or column values have been calculated, which are needed by another host to calculate other sub-matrices, these are sent by means of sockets. The underlying idea for distribution is to have either a chain of *n* hosts, or a number of hosts organised as a grid. Moreover, each host running a part of the processing manages the communication with the nearest hosts.

The first part of the execution is handled by *server.cpp* program that chooses the number of hosts that will be used (among the available *n*), according to the size of the 5 starting matrices: `ga[][]`, `dz[][]`, `ez[][]`, `hx[][]` and `hy[][]`. Once the *x* hosts have been chosen, the above matrices are split into *x* sub-matrices, and sent to the *x* hosts. The sub-matrices are spread depending on the position of the host.

The criteria for splitting the 5 matrices (`ga[][]`, `dz[][]`, `ez[][]`, `hx[][]` and `hy[][]`) into *x* sub-

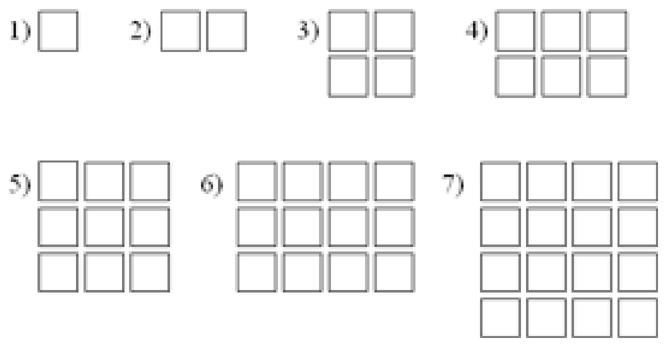

Fig. 6. Subdivision and layout of matrices in the grid approach (multiple hosts).

matrices are mainly two: the first concerns the x quantity of the submatrices to be created, the second the arrangement of them.

The quantity x of the sub-matrices, which is equal to the x number of the n computers to be available, depends on two factors: the size of the starting matrices, and a function that determines the accurate "multiple" of computers and their logical interconnection.

According to some size ranges of the matrices, memory occupation and execution times on a sample host are observer, and then a number y is advised as a possible way to handle the data. The size split will provide an exact multiple of hosts (see Fig. 6, in which each box represents both a submatrix and a host).

This number of hosts and parts is described by the following two mathematical equations

$$x1 = \frac{y+1}{2} \cdot \frac{y+1}{2} = \frac{y^2+2y+1}{4}$$

where y is the number suggested according to the initial occupancy test on a sample host. Moreover, the following equation will be used.

$$x2 = \frac{y}{2} \cdot \frac{y+2}{2} = \frac{y^2+2y}{4}$$

The equation pair gets the y value and returns the number of the sub-matrices in which the starting matrices have to be split. The number given by the first equation $x1$ represents the number of rows where submatrices will be arranged, whereas the second equation, giving $x2$ defines the number of columns. To make the system more efficient and increase parallelism, it is necessary to reduce drastically the waiting times of all computers. By arranging submatrices in this way, each host that finishes its calculations, will provide inputs to, and activate, both the host to its right (right), and the underlying host (lower), according to the scheme shown in Fig. 7.

This system significantly reduces the times for the global calculation of starting matrices compared to the chain method, as shown in the following example reported in Fig. 8.

At the time t_1 (for instance equal to 10 minutes), the first host finishes its calculations, and passes the results to the right

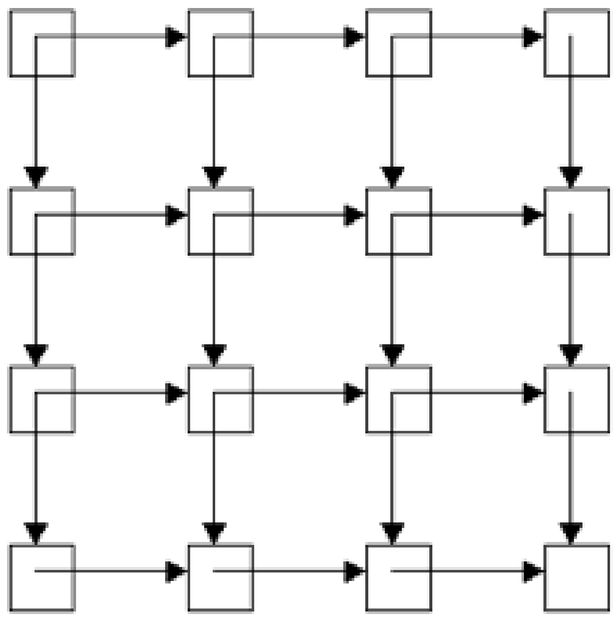

Fig. 7. Activation scheme for hosts used to calculate the starting matrices.

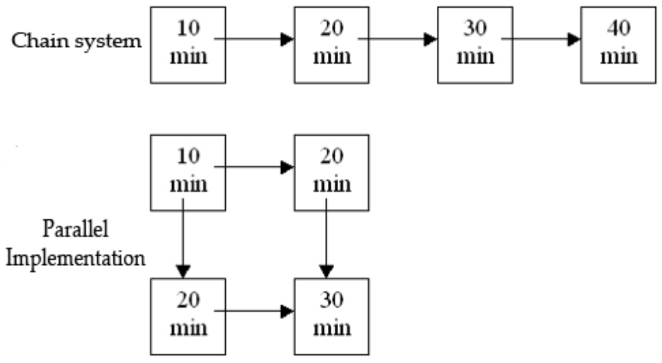

Fig. 8. Execution times using the chain system and using the grid approach. Although the first, calculating the same amount of data and using the same number of hosts it takes 40 minutes total, the second 30.

and the bottom host, ending their processing at the instant t_2 (20 minutes). Both of them finally relocate their results to the last host, which ends up to calculate their data at t_3 time (equal to 30 minutes).

Increasing the number of sub-matrices in which the starting matrices are divided, the times for the grid version of the algorithm improve, in contrast with the time for the chain version.

VI. Conclusions

This paper described the development of a distributed system for calculating 5 electromagnetic magnitudes ((`ga[][]`, `dz[][]`, `ez[][]`, `hx[][]` and `hy[][]`). It was necessary to initialize the matrices to well-defined values, through the dynamic initialization. It was then through the *server.cpp* program that the 5 starting matrices have been split into as many sub-matrices as the selected *x* hosts among the available *n*. Then data are transferred to the *client.cpp* program, and then are synchronized. The sockets offered extreme flexibility and ease of use for handling communication. At this point, the necessary computation has been spread among several hosts, each having a components performing the needed task.

References

[1] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, et al. A view of cloud computing. *Communications of the ACM*, 53(4):50–58, 2010.

[2] G. Borowik, M. Wozniak, A. Fornaia, R. Giunta, C. Napoli, G. Pappalardo, and E. Tramontana. A software architecture assisting workflow executions on cloud resources. *International Journal of Electronics and Telecommunications*, 61:17–23, 2015.

[3] S. Brenner and R. Scott. *The mathematical theory of finite element methods*, volume 15. Springer Science & Business Media, 2007.

[4] A. Calvagna and E. Tramontana. Automated conformance testing of Java virtual machines. In *Proceedings of Complex, Intelligent and Software Intensive Systems (CISIS)*. IEEE, July 2013.

[5] A. Calvagna and E. Tramontana. Delivering dependable reusable components by expressing and enforcing design decisions. In *Proceedings Of Compsac*, pages 493–498, Kyoto, Japan, 2013. IEEE.

[6] G. Capizzi, G. L. Sciuto, C. Napoli, E. Tramontana, and M. Woźniak. Automatic classification of fruit defects based on co-occurrence matrix and neural networks. In *Federated Conference on Computer Science and Information Systems (FedCSIS)*, pages 861–867. IEEE, 2015.

[7] R. Damaševičius, C. Napoli, T. Sidekerskienė, and M. Woźniak. Imf remixing for mode demixing in emd and application for jitter analysis. In *IEEE Symposium on Computers and Communication (ISCC)*, pages 50–55. IEEE, 2016. DOI: 10.1109/ISCC.2016.7543713.

[8] R. Damaševičius, C. Napoli, T. Sidekerskienė, and M. Woźniak. Imf mode demixing in emd for jitter analysis. *Journal of Computational Science*, page in press, 2017. DOI: 10.1016/j.jocs.2017.04.008.

[9] R. Giunta, G. Pappalardo, and E. Tramontana. Using Aspects and Annotations to Separate Application Code from Design Patterns. In *Proceedings of ACM Symposium on Applied Computing (SAC)*, March 2010.

[10] R. Giunta, G. Pappalardo, and E. Tramontana. Superimposing roles for design patterns into application classes by means of aspects. In *Proceedings Of ACM Symposium on Applied Computing (SAC)*, pages 1866–1868, Riva Del Garda (Trento), Italy, March 26-30 2012.

[11] T. Kapuściński, R. K. Nowicki, and C. Napoli. Application of genetic algorithms in the construction of invertible substitution boxes. In *International Conference on Artificial Intelligence and Soft Computing*, pages 380–391. Springer, 2016.

[12] V. W. Lee et al. Debunking the 100x gpu vs. cpu myth: an evaluation of throughput computing on cpu and gpu. In *Proceedings of International Symposium on Computer Architecture (ISCA)*. ACM, 2010.

[13] R. J. Luebbers and F. Hunsberger. Fdtd for nth-order dispersive media. *IEEE transactions on Antennas and Propagation*, 40(11):1297–1301, 1992.

[14] M. Mongiovì, G. Giannone, A. Fornaia, G. Pappalardo, and E. Tramontana. Combining static and dynamic data flow analysis: a hybrid approach for detecting data leaks in java applications. In *Proc. of Symposium on Applied Computing (SAC)*, pages 1573–1579. ACM, 2015.

[15] T. Namiki. A new fdtd algorithm based on alternating-direction implicit method. *IEEE Transactions on Microwave Theory and Techniques*, 47(10):2003–2007, 1999.

[16] C. Napoli, G. Pappalardo, and E. Tramontana. A mathematical model for file fragment diffusion and a neural predictor to manage priority queues over bittorrent. *International Journal of Applied Mathematics and Computer Science*, 26(1):147–160, 2016.

[17] D. Połap, M. Woźniak, C. Napoli, E. Tramontana, and R. Damaševičius. Is the colony of ants able to recognize graphic objects? In *International Conference on Information and Software Technologies*, pages 376–387. Springer, 2015.

[18] Dassault Systems. CST - computer simulation technology. www.cst.com.

[19] D. M. Sullivan. *Electromagnetic simulation using the FDTD method*. John Wiley & Sons, 2013.