

# Parallelizing NEC's Equation Solver Algorithm with OpenMP

Mario Trangoni, Victor Rosales  
`{mario.trangoni,victor.h.rosales}@intel.com`

Argentina Software Design Center (Intel)

**Abstract.** In order to take advantage of the multi-core architecture of modern processors, the legacy serial code must be analyzed to discover the regions where the parallelization effort can be more rewarding. This paper reports the parallelization and optimization procedures done on an HPC application. The Intel® VTune Amplifier XE tool was used to profile the NEC2C [7] software for simulation of electromagnetic response of antennas, and the proposed parallelization was implemented. The application got a speed up of nearly 7 times before a saturation effect was reached.

**Keywords:** Computer simulation, Parallel algorithms, Parallel programming, Performance analysis

## 1 Introduction

Traditionally, computer software has been written for serial computation. However, with the emergence of multi-core processors, parallel architecture is readily available on almost every computer and the software should take advantage of the benefits of parallel computing.

Parallel computing uses multiple processing elements simultaneously to solve a problem. Algorithms are adapted to parallel computing when the problem is broken into independent parts so that different processing elements can execute different parts of the algorithm simultaneously.

There are two main models of parallelization:

- In *Shared Memory*, the processing elements communicate each other by manipulating shared memory locations. For example, POSIX Threads [10], OpenMP [5], Intel Cilk<sup>TM</sup> Plus [11], and Intel TBB [12].
- In *Distributed Memory*, the processing elements communicate through messages, using a message passing protocol, like MPI [13].

In order to take advantage of this architecture, developers should write new parallel algorithms or adapt the serial ones. Since coding everything from scratch again would take considerable effort, it is desirable to analyze the legacy serial code to determine the regions where the execution takes more time and try to rewrite them as parallel code. The objective of this parallelization is to make those sections of the code perform as fast as possible to improve overall algorithmic efficiency. To identify this sections, called *bottlenecks* or *hot spots*, tools called *profilers* can be used.

To show the benefits of using Intel tools to parallelize serial legacy code, this work uses the public domain software NEC2C [4] (Numerical Electromagnetics Code translated to C). This software, used for the analysis of electromagnetic response of antennas and other metal structures, is freely available and simple enough to be a clear example of how to parallelize serial code.

In this paper the steps followed to parallelize NEC2C are explained, and the performance improvement obtained is shown and analyzed.

## 2 Discovering Bottlenecks

NEC was originally written in FORTRAN in the 1970s. From the several NEC versions, NEC-2, developed in 1981, is the newest version available as public domain. In 2003, it was translated from FORTRAN to C and renamed *nec2c* [7].

This software uses the method of moments [1] solution of the electric field integral equation for thin wires and the magnetic field integral equation for closed, conducting surfaces [2]. The algorithm has no theoretical limit and can be applied to very large arrays or for detailed modeling of very small antenna systems. NEC models can include wires buried in a homogeneous ground, insulated wires and impedance loads.

Models are defined as elements of wire or similar as an input text file. These models are then input into the NEC application to generate tabular results. The results can then be input into subsequent 'helper' applications for visual viewing and the generation of other graphical representations as Smith charts, etc. See Figure 1.

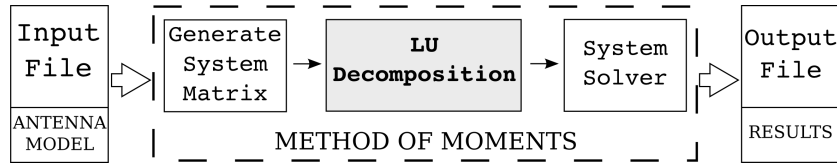


Fig. 1: NEC's modularized flow chart.

The software profiler Intel® VTune™ Amplifier [6] was used to detect the *hot spots* -the sections of the code that have the highest execution count-, by means of a performance analysis. The results of this analysis showed that the major bottleneck in *nec2c* is located in the factorization function *factr*.

The *factr* function uses *LU decomposition*, also called *LU factorization*, as a previous step to solve a system of linear equations. The algorithm is a matrix decomposition which writes a matrix as the product of a lower triangular matrix and an upper triangular matrix.

Results of the analysis (Fig. 2) show that, even with small inputs, *nec2c* spends most of the execution time in *factr*.

The second busiest function *\_mulxc3*, a third party function which will not be considered in this work, does complex numbers multiplication and is invoked mostly by *factr*.

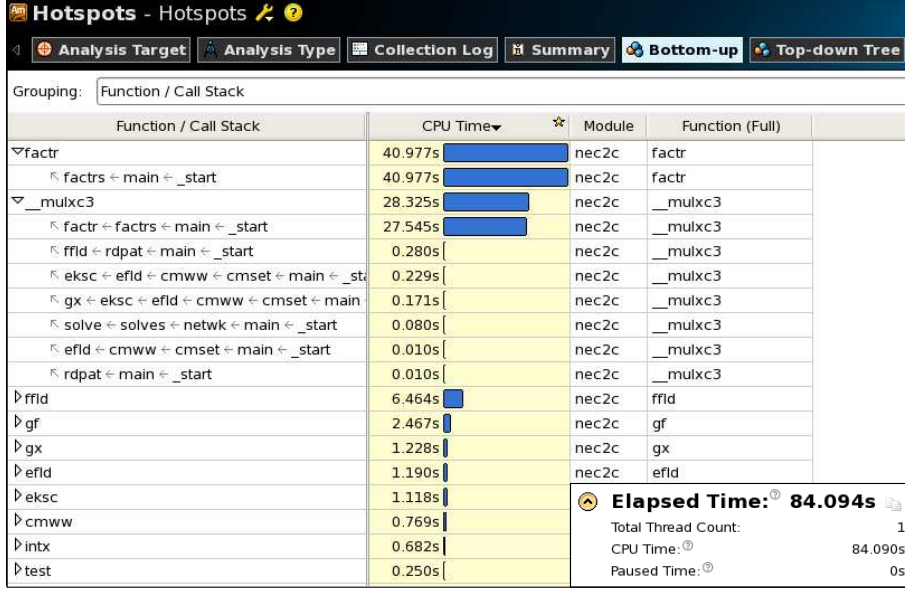


Fig. 2: Code profile result.

### 3 Analysis of the LU decomposition method

In order to propose a parallel algorithm for the LU decomposition method, this section shows how the method is implemented in *nec2c*.

The LU decomposition is basically a modified form of Gaussian elimination called Gauss-Doolittle method. It transforms the matrix  $A$  into an upper triangular matrix  $U$  by eliminating the entries below the main diagonal. The elimination is done column by column starting from the left, by multiplying  $A$  to the left with atomic lower triangular matrices. It results in a unit lower triangular matrix  $L$  and an upper triangular matrix  $U$ .

Doolittle showed that the Gaussian algorithm can be formulated as follows:  
 A real matrix  $A$  can always be represented as a product of two real matrices  $L$  and  $U$ :

$$A = L \cdot U \quad (1)$$

where  $U$  is an *upper triangular matrix*:

$$U = \begin{pmatrix} u_{11} & u_{12} & \dots & u_{1n} \\ 0 & u_{22} & \dots & u_{2n} \\ \cdot & & & \cdot \\ \cdot & & & \cdot \\ 0 & 0 & \dots & u_{nn} \end{pmatrix} \quad (2)$$

and  $L$  is a *unit lower triangular matrix*:

$$L = \begin{pmatrix} 1 & 0 & 0 & \dots & 0 \\ m_{21} & 1 & 0 & \dots & 0 \\ m_{31} & m_{32} & 1 & \dots & 0 \\ \cdot & & & & \cdot \\ \cdot & & & & \cdot \\ \cdot & & & & \cdot \\ m_{n1} & m_{n2} & m_{n3} & \dots & 1 \end{pmatrix} \quad (3)$$

The decomposition (1) of matrix  $A$  is called “ $LU$  decomposition of  $A$ ”.

The components  $m$  of  $L$  and  $u$  of  $U$  are obtained by using the following equations.

$$\begin{aligned} u_{ij} &= a_{ij} - \sum_{k=1}^{i-1} m_{ik} \cdot u_{kj} \quad i = 1, \dots, j-1 \\ \gamma_{ij} &= a_{ij} - \sum_{k=1}^{j-1} m_{ik} \cdot u_{kj} \quad i = j, \dots, n \\ u_{jj} &= \gamma_{jj} \\ m_{ij} &= \frac{\gamma_{ij}}{\gamma_{jj}} \quad i = j+1, \dots, n \end{aligned} \quad (4)$$

Because the diagonal elements of  $L$  are always the one, values  $m_{jj}$  do not need to be considered. Then,  $L$  and  $U$  can be written just as a  $(N \times N)$  matrix called the “ $LU$  matrix”:

$$LUMatrix = \begin{pmatrix} u_{11} & u_{12} & u_{13} & \dots & u_{1n} \\ m_{21} & u_{22} & u_{23} & \dots & u_{2n} \\ m_{31} & m_{32} & u_{33} & \dots & u_{3n} \\ \cdot & & & & \cdot \\ \cdot & & & & \cdot \\ \cdot & & & & \cdot \\ m_{n1} & m_{n2} & m_{n3} & \dots & u_{nn} \end{pmatrix} \quad (5)$$

This  $LU$  Matrix of  $A$  can now be used to calculate the solution vector as follows:

$$A \cdot \mathbf{x} \equiv L \cdot U \cdot \mathbf{x} = L \cdot (U \cdot \mathbf{x}) = \mathbf{b} \quad (6)$$

With  $U \cdot \mathbf{x} \equiv \mathbf{y}$ , the system  $L \cdot \mathbf{y} = \mathbf{b}$  can be obtained. The specific form of the matrix  $L$  can be solved using the forward substitution:

$$\begin{aligned} y_1 &= b_1 \\ y_i &= b_i - \sum_{j=1}^{i-1} m_{ij} \cdot y_j \quad i = 2, 3, \dots, n \end{aligned} \quad (7)$$

Since  $U$  is an upper triangular matrix, with knowledge of the auxiliary vector  $\mathbf{y}$ , it is simple to solve the system  $U \cdot \mathbf{x} = \mathbf{y}$  by backward substitution:

$$\begin{aligned} x_n &= \frac{y_n}{u_{nn}} \\ x_i &= \frac{1}{u_{ii}} \cdot \left[ y_i - \sum_{j=i+1}^n u_{ij} \cdot x_j \right] \quad i = n-1, n-2, \dots, 1 \end{aligned} \tag{8}$$

On the other hand, the new  $u_{ij}$  and  $m_{ij}$  values always take different places, so they can be saved as part of the original matrix  $A$ . It saves a great amount of space because there is one only matrix in memory, the original matrix, and not three (original, unit lower triangular and upper triangular matrix).

### 3.1 Rounding Error Optimization by Partial Pivoting

Suppose that there is a program that evaluate the inhomogeneous equations system after the LU decomposition. The average of the numerical results obtained in this program do not contain any procedural error (direct method) or any deviation from the “true” results but must therefore go back to the *rounding errors*.

Comparing the values of  $m_{ij}$  in the  $LU$  matrices, it can be noticed that the rounding error is the lowest when the  $m_{ij}$  values are the smallest. To keep  $m_{ij}$  as small as possible, the absolute value of  $\gamma_{jj}$  has to be as high as possible. This can be achieved simply by searching the element with the maximum absolute value in the column  $j$  and swap the whole row  $i$  with the row where this value is located.

Such a strategy is called an “*LU decomposition with partial pivoting*”. With partial pivoting, the possibly zero value in  $\gamma_{jj}$  can also be avoided, otherwise the decomposition would be impossible to solve.

## 4 Parallel Algorithm Proposal

### 4.1 Modularized View

The  $LU$  decomposition as currently implemented in NEC2C has many serial dependencies. One row operation depends on previous operations. To parallelize the code, we need to analyze the implementation and try to reorder the operations without changing the final result.

To simplify the explanation, we assume that the matrix size  $N$  is equal to 3 and that the maximums are placed at the diagonal (avoiding partial pivoting). In the following subsections, the gray boxes represent operations while the white boxes represent the required value.

## Original Implementation

This implementation works one row at a time from top to bottom. As the example is a matrix of size three, it should take 3 steps, one per row.

The first step (Row 0) requires only to divide by the diagonal element.

Row 0:

$$\begin{pmatrix} \boxed{a_{11}} & \boxed{a_{12}} & \boxed{a_{13}} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{pmatrix}$$

$$\text{Division: } a_{12} = \frac{a_{12}}{a_{11}} ; a_{13} = \frac{a_{13}}{a_{11}}$$

The second step (Row 1) requires linear combinations with results of the previous row and then to divide. It can be seen a strong dependency of this step with previous results, making the implementation non-parallelizable.

Row 1:

$$\begin{pmatrix} a_{11} & \boxed{a_{12}} & \boxed{a_{13}} \\ \boxed{a_{21}} & \boxed{a_{22}} & \boxed{a_{23}} \\ a_{31} & a_{32} & a_{33} \end{pmatrix}$$

$$\text{Subtraction: } a_{22} = a_{22} - a_{12} \cdot a_{21} ; a_{23} = a_{23} - a_{13} \cdot a_{21}$$

$$\begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & \boxed{a_{22}} & \boxed{a_{23}} \\ a_{31} & a_{32} & a_{33} \end{pmatrix}$$

$$\text{Division: } a_{23} = \frac{a_{23}}{a_{22}}$$

The third and final step (Row 2) requires two linear combinations, the third row with the first, and then third with the second. There are dependencies with the other two previous results.

Row 2:

$$\begin{pmatrix} a_{11} & \boxed{a_{12}} & \boxed{a_{13}} \\ a_{21} & a_{22} & \boxed{a_{23}} \\ \boxed{a_{31}} & \boxed{a_{32}} & \boxed{a_{33}} \end{pmatrix}$$

$$\text{Subtraction: } a_{32} = a_{32} - a_{12} \cdot a_{31} ; a_{33} = a_{33} - a_{13} \cdot a_{31}$$

$$\begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & \boxed{a_{23}} \\ a_{31} & \boxed{a_{32}} & \boxed{a_{33}} \end{pmatrix}$$

$$\text{Subtraction: } a_{33} = a_{33} - a_{23} \cdot a_{32}$$

### Parallelization Proposal

To remove dependencies, the implementation is rearranged from *row operations* to *matrices operations*. Thus, the first step operates on the full matrix and the subsequent steps will successively reduce the size of the operated matrix, as shown in the figure below, by one unit from the upper left corner.

Matrix ( $3 \times 3$ ):

$$\begin{pmatrix} \boxed{a_{11}} & \boxed{a_{12}} & \boxed{a_{13}} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{pmatrix}$$

$$\text{Division: } a_{12} = \frac{a_{12}}{a_{11}} ; a_{13} = \frac{a_{13}}{a_{11}}$$

At this point we do the complete matrix  $3 \times 3$  linear combination that allows to exploit the benefits of OpenMP [5] “*for*” parallelization.

$$\begin{pmatrix} a_{11} & \boxed{a_{12}} & \boxed{a_{13}} \\ \boxed{a_{21}} & \boxed{a_{22}} & \boxed{a_{23}} \\ a_{31} & a_{32} & a_{33} \end{pmatrix}$$

$$\text{Subtraction: } a_{22} = a_{22} - a_{12} \cdot a_{21} ; a_{23} = a_{23} - a_{13} \cdot a_{21}$$

$$\begin{pmatrix} a_{11} & \boxed{a_{12}} & \boxed{a_{13}} \\ a_{21} & a_{22} & a_{23} \\ \boxed{a_{31}} & \boxed{a_{32}} & \boxed{a_{33}} \end{pmatrix}$$

$$\text{Subtraction: } a_{32} = a_{32} - a_{12} \cdot a_{31} ; a_{33} = a_{33} - a_{13} \cdot a_{31}$$

Next, the same process is repeated by iterating over matrices that will reduce its size as indicated before.

Matrix ( $2 \times 2$ ):

$$\begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & \boxed{a_{22}} & \boxed{a_{23}} \\ a_{31} & a_{32} & a_{33} \end{pmatrix}$$

$$\text{Division: } a_{23} = \frac{a_{23}}{a_{22}}$$

$$\begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & \boxed{a_{23}} \\ a_{31} & \boxed{a_{32}} & \boxed{a_{33}} \end{pmatrix}$$

Subtraction:  $a_{33} = a_{33} - a_{23} \cdot a_{32}$

## Results

As a result of the aforementioned process, the following  $LU$  Matrix is obtained.

$$\begin{pmatrix} u_{11} & u_{12} & u_{13} \\ m_{21} & u_{22} & u_{23} \\ m_{31} & m_{32} & u_{33} \end{pmatrix}$$

## 4.2 Implementation

Since input sizes that are normally used in NEC2C are not large enough to justify the use of a distributed memory model, the shared memory model was selected for implementation of our proposed algorithm.

OpenMP is an API that supports multi-platform shared memory multiprocessing programming in C, C++, and FORTRAN on most processor architectures and operating systems. Moreover, it is supported by the principal ANSI C compilers.

Since the parallel algorithm works by operating over a matrix that decrease its order per iteration, at the very beginning the workload distributed per thread is bigger than the work at the end.

The three following subsections show the implementation of the three main tasks to achieve: “Partial Pivoting” to ensure that the row maximum is at the diagonal, “Division” by this maximum and “Linear combination” of the first row with the others in the matrix to obtain a factorized matrix.

An initial transposition was also implemented, in order to operate by rows instead of by columns, because ANSI C performs better in this way. The reason for this is based on how ANSI C manages memory allocation [14].

### Maximum Search

To implement a parallel partial pivoting in OpenMP, the analyzed row has to be divided into  $N$  sections, where  $N$  is equal to the number of threads available. Thus,  $N$  threads are carried out to search the largest element of each section. Then, the resulting vector is compared serially and the global maximum is found. Just the original position is saved in a especial vector that will be used at the *solve* function.

```
# pragma omp parallel shared (a,...) private(i,...)
{ /*Begin parallel zone*/
  rndim=r*ndim;
  dmax=creal(a[r+rndim]*conj(a[r+rndim]));
  ip[r]=r+1; /*Partial pivoting vector*/
  if((n-(r+1))>(threads*2)) {
```



```

        for (i=0; i<threads; i++){
            dmaxaux[i]=dmax;
        }
    # pragma omp for          /*compare modules per thread*/
        for (i=r+1; i<n; i++) {
            thread=omp_get_thread_num();
            elmagaux[thread]=creal(a[i+rndim]*conj(a[i+rndim]));
            if (elmagaux[thread]>dmaxaux[thread]) {
                ipaux[thread]=i+1; /*Max aux position*/
                dmaxaux[thread]=elmagaux[thread];
            }
        }
    # pragma omp barrier      /*compare the last modules*/
        for (i=0; i<threads; i++){
            if (dmaxaux[i]>dmax) {
                ip[r]=ipaux[i]; /*Max position*/
                dmax=dmaxaux[i];
            }
        }
    }
    else{ /*if the size of the array is small*/
    # pragma omp single
    {
        for (i=r+1; i<n; i++) {
            elmag=creal(a[i+rndim]*conj(a[i+rndim]));
            if (elmag>dmax) {
                ip[r]=i+1; /*Maximum position*/
                dmax=elmag;
            }
        }
    }
    }
}

```

---

### Row Exchange and Division by Row's Diagonal Element

This stage of the algorithm involves the exchange of rows only if the maximum is not located on the diagonal. It is also very important to avoid division by zero and to reduce significantly the precision error. Next, the division is performed by the diagonal element or first element of the row.

```

if ((ip[r]!=r+1) && (r!=n-1)) {
    i=ip[r]-1; /*if max not at diagonal, do interchange */
    # pragma omp for
        for (j=r; j<n; j++) { /*Row's interchange */
            arj=a[i+j*ndim];
            a[i+j*ndim]=a[r+j*ndim];
            a[r+j*ndim]=arj;
        }
    }

    # pragma omp for
        for (i=r+1; i<n; i++) { /*n-1 iterations */
            a[i+rndim]=a[i+rndim]/a[r+rndim];
        }
    }

```

}

---

## Linear Combinations

Here the reduction takes place in the matrix, taking advantage of the *Linear Independence* of an equation system. Having no limitations in the execution order, due to the rearrangement of the algorithm, this loop is simply parallelized with an OpenMP pragma.

```
# pragma omp for
for (j=r+1; j<n; j++) {
    /*n-1 iterations - rows */
    for (i=r+1; i<n; i++) {
        /*n-1 iterations - elements */
        a[i+j*ndim] -= (a[i+r*ndim]*a[r+j*ndim]);
    }
}
} /*Finish Parallel zone*/
```

---

## 5 Discussion of Results

By reviewing the code included in the *Maximum Search* subsection, it can be noticed that it is not an efficient implementation. Because of the complexity of the algorithm and data transfer times, there is no performance improvement or decline compared with the original version. For this reason, it was not included in the final implementation.

The code of the second subsection, *Row Exchange and Division by Row's Diagonal Element*, does not add big performance gains, but also it does not perform worse than the original version.

The last subsection, on the contrary, is where the discovered bottleneck was having more effect, and obviously where big gains are obtained.

An implementation with *Maximum Search*, *Row Exchange and Division by Row's Diagonal Element* and *Linear Combinations* subsections was executed and compared against an implementation including only the last two subsections. Table 1 shows the performance loss obtained by the former with respect to the latter, for different numbers of threads.

Number of Threads	Implementation 1 (sec.)	Implementation 2 (sec.)	Difference (Imp.1 – Imp.2)
2	906.71	870.15	4.20 %
4	531.62	516.49	2.93 %
8	344.77	343.93	0.24 %

Table 1: Comparison of performance between implementing the three subsections (Implementation 1) against the last two subsections (Implementation 2) executing 5 times over a matrix of 5000 x 5000 elements.

Based on these results, the final implementation included only the second and third subsections. The code was compiled with GCC 4.4.4 [8] using the flags: `-O2 -march=core2 -mtune=generic -fopenmp` and ran on a server *Intel Westmere-EP* (2 x Intel Xeon® X5670 (6 cores) @ 2.93 Ghz / 12 MB shared Last-Level Cache / 12Gb RAM DDR3-1066 (8533,33 MB/seg)) with *CentOS release 5.4 (Final)* [9].

For the analysis, the code was executed with an input size of 15,000 elements that generate a vector of  $2.25 \times 10^8$  complex long double variables of 64 bits *ANSI C*.

Five executions were ran to calculate the averages. The scalability of the code, Figure 3, shows how the program takes advantage of the hardware availability.

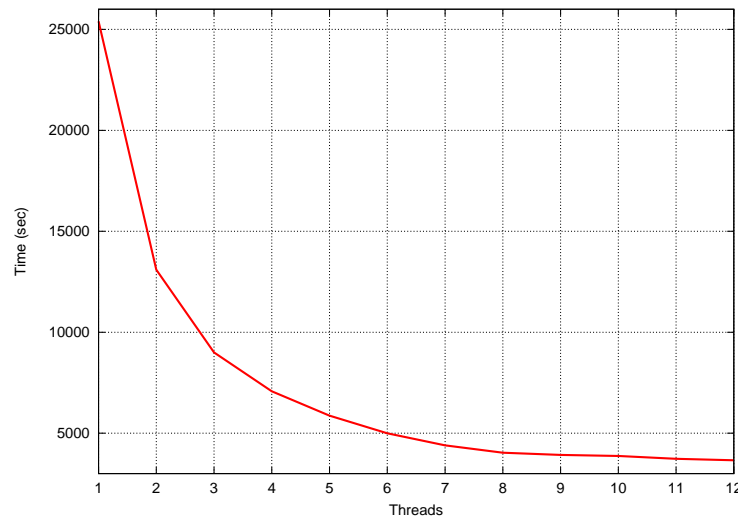


Fig. 3: Execution time versus number of threads for a fixed total problem size (15000 x 15000).

The speedup of the new parallel code is shown in Figure 4.

Over small inputs (matrix size of 5000) a *saturation effect*, shown in Figure 5, was found. Since the parallel algorithm works by operating over a matrix that decrease its order per iteration, at the very beginning the workload distributed per thread is bigger than the work at the end. Because of that, for a certain input size, there is a fixed number of threads until the speedup grow. The *saturation effect* means that the thread's work distribution does not compensate the overhead generated resulting on a flat speedup.

Similarly, higher levels of optimization (GCC's `-Ox` optimizations) reduce this ideal number of threads, which flattens the speedup. The reason is that the compiler improves its performance and the thread creation overhead becomes relevant.

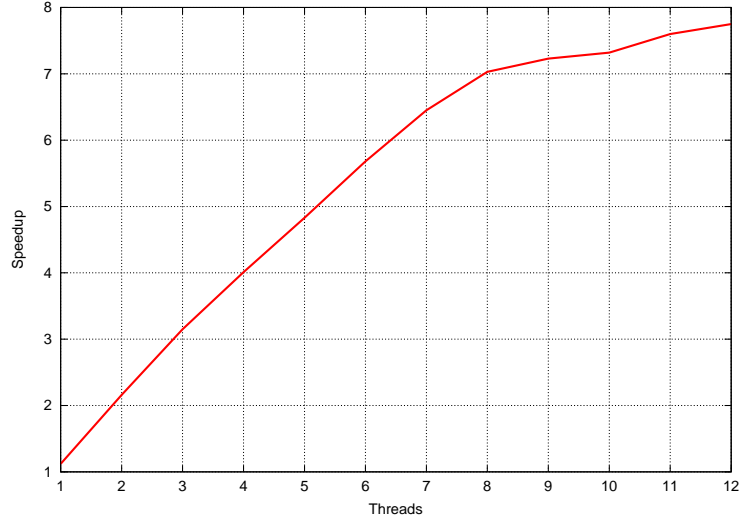


Fig. 4: Speedup of NEC's parallel code with input order 15000.

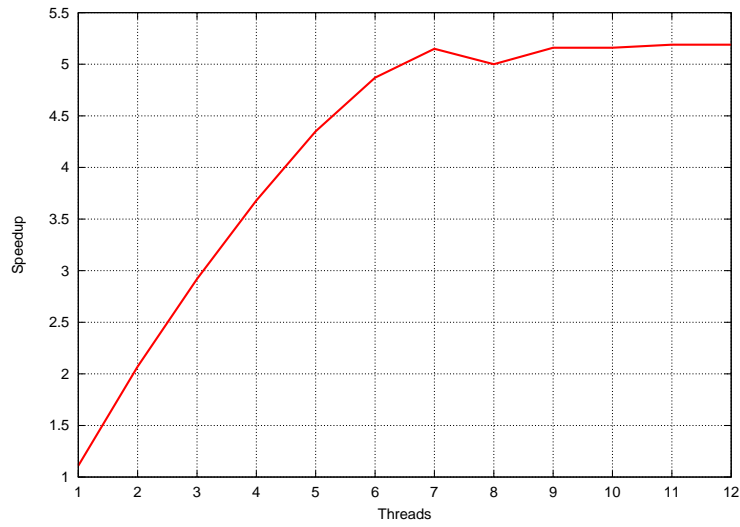


Fig. 5: Speedup of NEC's parallel code with input order 5000.

## 6 Conclusions

With the emergence of multi-core processors, parallel architecture is readily available on almost every computer. To take advantage of this architecture, the legacy code must be analyzed to discover the *bottlenecks* or regions where the parallelization effort can be more rewarding. In this work the NEC2C software, used for simulation of electromagnetic response of antennas and metal structures, was profiled and a proposal for parallelization was implemented.

The results of the experiments show that by implementing a fairly simple modification of 60 lines of code, based on the shared-memory model of OpenMP, a considerable increase in the performance of the system can be obtained (Figures 3 and 4). The use of Intel's profiling tool, Intel® VTune Amplifier XE, simplified the process of discovering the regions of code where the parallelization can be done.

Moreover, this paper provides a simple example that can be used for educational purposes on the software parallelization process.

As a second step in the parallelization process, we suggest the use of the distributed-memory model (based on MPI) in order to avoid the storage of the complete input array in main memory.

Another future work could be the use of the Intel® Math Kernel Library (MKL). It is a computing math library of highly optimized, extensively threaded math routines for applications that require maximum performance. Core math functions include sparse solvers and vector math, which can obviously result in a better algorithm implementation.

## References

- [1] Roger F. Harrington, *Field Computation by Moment Methods*, IEEE Press Series on Electromagnetic Wave Theory, Wiley-IEEE, ISBN 0-78-031014-4, 1993.
- [2] F. Gisin, *Using the method of moment NEC code to solve EMC problems*, IEEE International Symposium on Electromagnetic Compatibility, Dallas, TX, USA, 9-13 August 1993.
- [3] H. Sormann, *Numerische Methoden in der Physik*, Institut für Theoretische Physik - Computational Physics, TU Graz, Graz, Austria, 2011.
- [4] G. J. Burke, A. J. Poggio, *Numerical Electromagnetics Code (NEC-2)*, Public Domain Code, 1980.
- [5] OpenMP Architecture, *OpenMP Application Program Interface*, version 3.0, May 2008, <http://openmp.org/>.
- [6] Intel Corporation, *Intel Vtune Amplifier XE 2011*, 2011.
- [7] N. Kyriazis, *Translation of the NEC-2 FORTRAN source code to the C language*, General Public License, 2003.
- [8] Free Software Foundation Inc., *GCC Version 4.4.4*, Red Hat 4.4.4-13, 26th July 2010, <http://gcc.gnu.org/>.
- [9] Community Enterprise Operating System, *CentOS 5.4 release (Final)*, 3rd September 2009, <http://www.centos.org/>.
- [10] Blaise Barney, *POSIX Threads Programming*, Lawrence Livermore National Laboratory, 24th August 2011, <https://computing.llnl.gov/tutorials/pthreads/>.
- [11] Intel Corporation, *Intel® Cilk™ Plus v1.1*, 2011.
- [12] Intel Corporation, *Intel® Threading Building Blocks 4.0*, 2011.
- [13] MPI, *A Message-Passing Interface Standard*, version 2.2, 4th September 2009, <http://www.mpi-forum.org/docs/mpi-2.2/mpi22-report.pdf>.
- [14] Brian W. Kernighan, Dennis M. Ritchie, *The C Programming Language, second edition*, Englewood Cliffs, NJ: Prentice Hall. ISBN 0-13-110362-8. March 1988.
- [15] Michael T. Heath, *Parallel Numerical Algorithms - Chapter 6: LU Factorization*, Department of Computer Science, University of Illinois at Urbana-Champaign, 2011.