Parallel NEC

Kaddy Marindi (1416262)
School of Electrical and Information Engineering
University of Witwatersrand, Johannesburg
Subject : ‘Laboratory Project
Course Code : ELEN4012A
Supervisor : Prof. Alan Clark

***Abstract*—This paper presents project to parallelize ELectromagnetic Numerical Code (NEC). The objective of is to enhance the computational performance of NEC2++ (an open-source implementation of NEC writeen in C++) by utilizing a distributed multicore architecture consisting of a 16-node cluster, with each node equipped with a four-core processor. The code is parallelized in a hybrid approach using OpenMP libray for shared memory, multithreading, parallelization and MPI library for distributed memory parallelization. NEC2++ computational-intesive operations—namely, the matrix filling, the LU decomposition, and the matrix solution solve—are analayzed and parallelized. The matrix is evenly distributed among the nodes using a cyclical distribution layout. LU decompsition is solved using level 3 BLAS, and it resulted in 4 times performce gain when the number of nodes are increased to 8.**

I. Introduction

The Numerical Electromagnetic Code (NEC) program is commonly used for the design of televisions, radios, and other devices to model the electromagnetic response of surface antennas, wires, and metal structures [5], [6]. While NEC2 was originally designed to run on mainframes, newer versions like NEC2++ have been ported to run on PCs [1]. However, a major challenge when running NEC on a PC is its slow execution speed and the limited size of structures that can be analyzed due to insufficient memory [1]. In the past, NEC has been parallelized using computer clusters with single-core processors [1]–[3]. However, the recent shift to multicore processor architectures has introduced new tools, such as OpenMP, that allow for multithreading parallelization in multicore shared memory systems, which is significantly faster than distributed memory systems due to data locality.

This paper presents an investigation project aimed at improving the computation time of NEC2++ and expanding the size of structures that can be analyzed by parallelizing the code in a distributed multicore architecture with a 16-node cluster. A hybrid approach is employed, utilizing both distributed parallelization among the nodes and multithreading parallelization using multicore processors within the nodes by leveraging the MPI and OpenMP libraries, respectively. Additionally, the MKL BLAS library is used to achieve optimal numerical computation speed.

The first section of this paper provides the background information, followed by an iterative approach to parallelizing NEC2++. The results are then analyzed and discussed, leading to the conclusion of the paper.

II. Background

The first operation NEC does is to calculate the interaction matrix of the wires. The interaction matrix is calculated between the wire segments, N making the structure, which results in an interaction matrix of size $N \times N$ [1]. Where N is the sum of all the segments in each wire. The excitation vector $\vec{E}$ is calculated as a function of the sources wires [1].

$$\begin{bmatrix} A_{11} & A_{12} & ... & A_{1N} \\ A_{21} & A_{22} & ... & A_{2N} \\ . & . & . & . \\ . & . & . & . \\ A_{N1} & A_{N2} & ... & A_{NN} \end{bmatrix} \times \begin{bmatrix} I_1 \\ I_2 \\ . \\ . \\ I_N \end{bmatrix} = \begin{bmatrix} E_1 \\ E_2 \\ . \\ . \\ E_N \end{bmatrix} \tag{1}$$

Where A_{ij} is the interaction between segment i and j, I_i is the current on the segment i which can be calculated by solving matrix equ. (1) and E_i is the excitation on segment i excitation [1].

A. Structure memory usage

An approximation of the memory in bytes required to analyze the structure with N segments is given by

$$Mem = N^2 \cdot 16 \tag{2}$$

According to equ. (2) the memory required for N segments has a quadratic growth with the number of segments [2]. For a large number of segments, the RAM size will be the limiting factor. A structure with 14746 segments will require approximately 3.5GB of RAM. Assuming that we are working with a commonly available 4GB RAM PC, the number of segments that can fit in memory cannot exceed 14746 segments.

B. Intensive operations

NEC has three intensive operations that are worth parallelizing:

1) The A-matrix by calculating the interaction between segments. It requires $O(N^2)$ operations.

2) LU decomposition of the A-matrix factorization into permutation, lower and upper triangle (PLU) is the most time-consuming operation which requires $O(N^3)$ simple operations.
3) Solving of $\vec{I}$, requires $O(N^2)$ operations using forward and backward substitution.

III. Parallelizing NEC2++ code

A. Code Structure

NEC2++ code is written in C++ object orientation style, which makes it easy to modify and add new functionality [8]. NEC2++ has a lot of files and functions that are hard to navigate. Table 1 below shows functions modified when parallelizing NEC2++ and their file location.

TABLE I
THE FILES AND FUNCTION MODIFIED WHEN PARALLELIZING THE CODE.

File	functions
nec_context.cpp/h	`cmset(..)`
nec_context.cpp/h	`cmwww(..)`
matrix_algebra.cpp/h	`Lu_decompose(..)`
matrix_algebra.cpp.h	`solve_ge(..)`
test_cpp.cpp	`main(..)`

`cmset(..)` and `cmwww(..)` functions are responsible for matrix filling. `cmwww(..)` is called within `cmset(..)`. `Lu_decompose(..)` is responsible for LU factorization. The solving of matrix equ. (1) in order to find I_i is done in `solve_ge(..)` function. test_cpp.cpp is used for initializing NEC, adding the wire structure and for compiling the code.

B. Matrix distribution layouts

There are different layouts that can be used for distributing a dense matrix in the cluster nodes P_n. The layout used for the distribution of the matrix has a major impact on load balance and communication amongst the processors which critically affects the scalability of parallel code [7]. In addition, the layout chosen should allow the use of level 3 BLAS operations in a single processor. Level 3 Blas operations give processor pick performance for basic matrix-matrix algebra and are exceedingly faster than level 1 and 2 BLAS operations.

There are four layouts considered for data distribution, namely: the one-dimension block, cyclical column, the one-dimension block cyclical column, and the two-dimension block cyclical distribution. The two-dimension cyclical block layout depicted in fig.1 gives the best even distribution between the processor, which avoid processor idling and allow the use of level 3 BLAS operation when doing Gaussian elimination (LU factorization).

A 2-D block cyclical distribution is described by four variables, P, Q, β. Where $P \times Q$ describes the processor grid as shown inf fig. 1 and $\beta \times \beta$ is the block size. The global indexing of the block is $m \to \langle p, b, i \rangle$, where p, b, i is the processor index, block number within the processor and index within the block respectively.

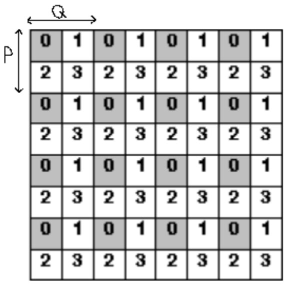

Fig. 1. cyclical block distribution over $P \times Q$ processor grid.

$$m \to \langle s \mod P, \frac{s}{P}, m \mod \beta \rangle \tag{3}$$

where s is the number of blocks defined as the number of rows or columns divide by β. We deduced that the processor to which the block belongs can simple be calculated using equ. (4).

$$rank = (i \mod Q) \times P + j \mod Q \tag{4}$$

where processor rank is from 0 to $(T-1)$. i and j are the global row and column block index. T is the number of processors and $T = Q \times P$.

Cyclical distribution allows analysis of a large structure with many segments that would not fit in a single machine. The local portion of the global matrix distributed to each processor has a size $\frac{N}{P} \times \frac{N}{Q}$.

C. Parallel matrix filling.

Parallel matrix filling was implemented in this solution using algo. 1 does not significantly change the original NEC2++ matrix filling code. There are two basic operations added to the original solution which are also illustrated in the flow diagram in fig. 2.

Algorithm 1 Matrix filling

```
procedure CMSET($*A, nrow$)
  for $i \leftarrow 0, row$ do
    for $j \leftarrow 0, row$ do
      $b_i \leftarrow i/blockSize$
      $b_j \leftarrow j/blockSize$
      $i' \leftarrow$ offset i index
      $j' = offsetjindex$
      $A[j' * (nrow/Q) + i'] \leftarrow$ calculate the interaction
    end for
  end for=0
```

The first step is checking if the interaction element $A_{i,j}$ belongs to the local processor by checking if the processor rank is equal to equ. (4). If it does, then $A_{i,j}$ is offset to fit in the local interaction matrix using equ. (5).

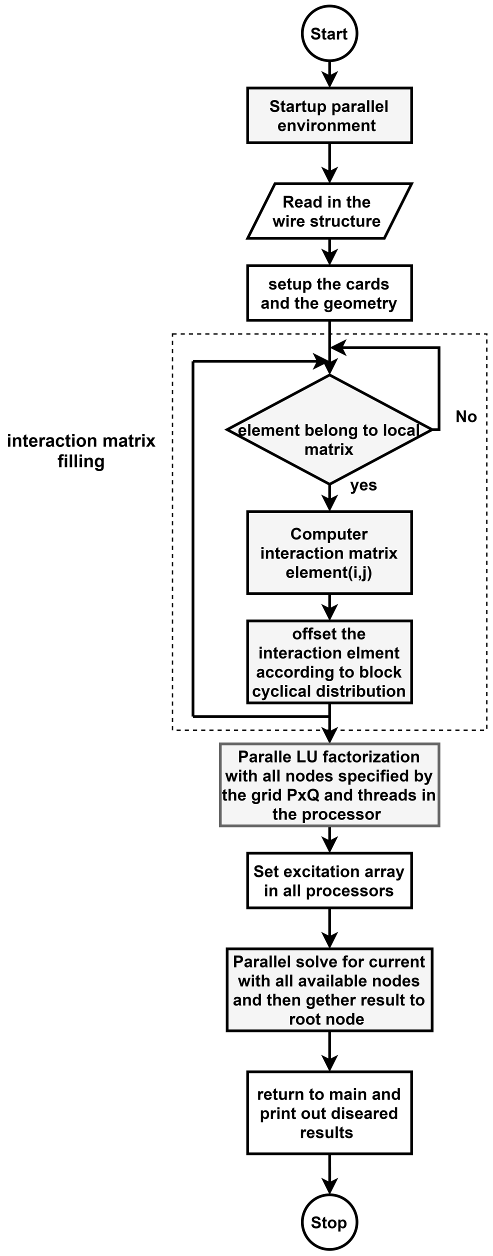
Start
Startup parallel environment
Read in the wire structure
setup the cards and the geometry
interaction matrix filling
element belong to local matrix
No
yes
Computer interaction matrix element(i,j)
offset the interaction elment according to block cyclical distribution
Paralle LU factorization with all nodes specified by the grid PxQ and threads in the processor
Set excitation array in all processors
Parallel solve for current with all available nodes and then gether result to root node
return to main and print out diseared results
Stop

Fig. 2. Parallelizing NEC flow diagram.

$$\begin{aligned} i' &= \frac{i}{\beta b \cdot P} + i \mod n_b \\ j' &= \frac{i}{\beta b \cdot P} + i \mod n_b \end{aligned} \tag{5}$$

Where i and j are a row and column global index of the interaction matrix. i' and i' is the local index in the processor that the interaction element is stored in.

D. LU decomposition

Computer spent most of the time finding the solution of a matrix of equations, in order to do that effectively, a matrix is first factorized into a lower (L) and upper (U) triangular matrix such that $A = LU$. Parallelizing the LU factorization matrix is quite a complex process, thus an iterative process was taken to parallelize LU factorization. The first thing was to implement simple serial LU factorization in a single machine shown in algo. 2. Parallel LU is compared with serial LU solution for correctness.

Algorithm 2 Simple LU factorization

```
procedure LUDECOMPSE($*A, n$)
  for $i \leftarrow 0, n$ do
    for $j \leftarrow i+1, n$ do
      $A[j \times n + i] /= A[i \times n + i]$
      for $k \leftarrow 1+1, n$ do
        $A[j][k] -= A[j \times n + i] * A[i \times n + k]$
      end for
    end for
  end for=0
```

1) Derivation of a Block Algo. for LU factorization: Provided $N \times N$ matrix A partitioned in blocks which is factorized to L and U matrix also partitioned in blocks as shown in fig. 3 below. Where a size of A_{00} is $\beta \times \beta$, A_{10} and A_{01} are $\beta \times (N - \beta)$ and $(N - \beta) \times \beta$ matrix, and A_{11} is $(N - \beta) \times (N - \beta)$ matrix.

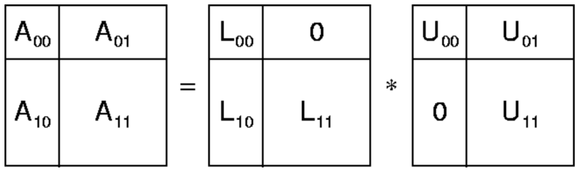

Fig. 3. Block LU factorization for partitioned matrix-A

$$L_{00}U_{00} = A00 \tag{6}$$

$$L_{10}U_{00} = A10 \tag{7}$$

$$L_{00}U_{01} = A01 \tag{8}$$

$$L_{10}U_{01} + L_{11}U_{11} = A11 \tag{9}$$

Equ. 6 is solved using regular LU factorization in Alg. 1. Once L_{00} and U_{00} are determined, they can be used to solve L_{10} and U_{01} in equ. 7 using BLAS 3 triangular matrix solve operation `ztrs`.

$$L_{11}U_{11} = A_{11} - L_{10}U_{01} = A'_{11} \tag{10}$$

Equ. (10) is obtained from equ. (9) by moving the right hand side to the left. Where A'_{11} is computed using level 3 Blass matrix-matrix multiplication operation `cblas_zgemm`.

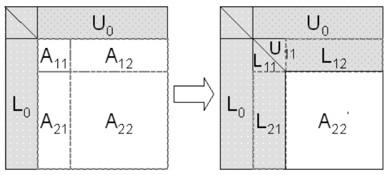

Fig. 4. Iterative LU decomposition steps

Once done computing equ 9, then A'_{11} partition into A_{11}, A_{11}, A_{11} and A_{11} matrix as illustrated in fig 4. The shaded are in fig. 4 are the block done factorized. Repeatedly compute equ. 6 to 9 10 for k times, where $k = N/\beta$.

2) Parallel LU factorization: Alg. 3 demonstrates the implementation of parallel LU factorization in distributed memo using four main functions: `Diag_func`, `Diag_func`, `Row_func`, and `Inner_func`. `Diag_func` computes the diagonal a single LU block using algo. 2 in a single machine while the other machines are waiting. Once the diagonal block $A_{i,i}$ LU is done, it is broadcasted to all other nodes (The diagonal block LU is the critical path). All the nodes have receive $A_{i,i}$ LU before `Diag_func` and `Row_func` computes column and row block as done in equ. 7 and 8 respectively using mkl level 3 BLAS triangular solve operation called `cblas_xTRSM`. The L_0 and U_0 row and column block must be sent to all the machine before computing equ. 10 using `Inner_func`. The `Inner_func` function calls BLAS level 3 `cblas_zgemm` for matrix multiplication. The four main functions `Diag_func`, `Diag_func`, `Row_func`, and `Inner_func` are iteratively called until the whole matrix is LU factorized.

Algorithm 3 Parallel: block-partitioned dense LU

```
procedure P_LU_DEC($*A, n, num\_blocks$)
 1: $set\_mkl\_num\_of\_threads$
 2: $set\_omp\_num\_of\_threads$
 3: for $i \leftarrow 0, num\_blocks$ do
 4:     if $(i \mod P) \times Q + i \mod Q == myrank$ then
 5:         diag_op($i$)
 6:         copy_diag_block($i$)
 7:     end if
 8:     Brodact_diag_block
 9:     #pragma omp parallel for
10:     for $j \leftarrow i + 1, num\_blocks$ do
11:         if $(i \mod P) \times Q + j \mod Q == myrank$ then
12:             row_op($i$)
13:         end if
14:         if $(j \mod P) * Q + i \mod Q == myrank$ then
15:             col_op($i$)
16:         end if
17:     end for
18:     for $j \leftarrow i + 1, num\_blocks$ do
19:         Brodact_row_blocks
20:         Brodact_col_blocks
21:     end for
22:     for $j \leftarrow 1 + i, num\_blocks$ do
23:         #pragma omp parallel for
24:         for $k \leftarrow 1 + i, num\_blocks$ do
25:             if $(j \mod P) \times Q + k \mod Q == myrank$
    then
26:                 inner_op($i$)
27:             end if
28:         end for
29:     end for
30: end for=0
```

E. Data communication

Communication overheads between the nodes can significantly diminish parallel code in distributed memory if not done efficiently. It is recommended to use MPI collective communication where possible. It is essential to send a large message instead of many small messages which introduce node synchronization overheads [9].

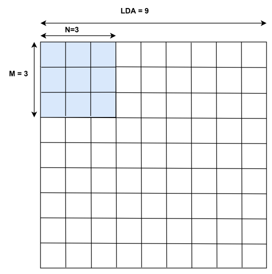

Fig. 5. Memory layout of a row and column of a matrix in row-major storage

For the instance where you want to send a block of data shaded by blue in row-major in fig.9, MPI user defined datatype MPI_Type_vector efficiently sends the whole block in one message. MPI_Type_vector for sending the block size shaded by blue in fig. 9 is illustrated in fig. 6

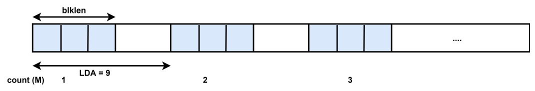

Fig. 6. A vector datatype is built up out of blocks strides of elements of a constituent type

The large the block size the better. The computation time speed for different block size is investigated and plotted in fig. **??**.

F. Matrix equation solve

The matrix solve implemented in NEC2++ function `solve_ge()` is not giving the correct result when solving LU matrix found using serial LU in algo. 1. Thus we decide to re-implement this function using standard algo. 4 which resulted in the same $\vec{i}$ as the original NEC2++ solution.

1) Serial `solve_ge()` function re-implementation: $A\vec{x} = \vec{b}$ system linear equation can also be expressed as $LU\vec{x} = \vec{b}$ after applying LU factorization. Then first do forward substitution:

$$L\vec{x} = \vec{y} \tag{11}$$

using forward substitution. Then solve $U\vec{x} = \vec{y}$ using backward substitution.

2) Solving Solution Matrix in parallel: Parallel solution solve is done using algo. 5 which parallelize the new implementation of serial solve above. The solution solve does not consider permutation vector because LU decomposition is done without pivoting.

The first thing that must be done is to gather all the diagonal block in the global interaction matrix presentation from every node. All the processors have vector b. OpenMP is used to parallize the loop. The parallel solving of the interaction

Algorithm 4 Serial: solution vector of $A * \vec{I} = \vec{b}$

```
procedure SOLVE_GE($*A, *P, *b, *x, n$)
 1: for $i \leftarrow 0, n$ do
 2:     $x[i] \leftarrow b[P[i]]$
 3:     for $j \leftarrow 0, i$ do
 4:         $A[i] -= A[i \times n + j] \times x[j]$
 5:     end for
 6: end for
 7: for $i \leftarrow n - 1, 0$ do
 8:     for $j \leftarrow i + 1, n$ do
 9:         $x[i] += A[i \times n + j] \times x[j]$
10:     end for
11:     $x[i] \leftarrow x[i]/A[i \times n + i]$
12: end for=0
```

Algorithm 5 Parallel: solution vector of $A * \vec{I} = \vec{b}$

```
procedure SOLVE_GE($**A, *P, *b, *x, n$)
 1: Gather all A diagonal blocks
 2: pragma omp parallel for
 3: for $i \leftarrow 0, n$ do
 4:     $x[i] \leftarrow b[P[i]]$
 5:     for $j \leftarrow 0, i$ do
 6:         if $doIHaveA$ then
 7:             compute $i_{local}$ and $j_{local}$
 8:             $x += A[i \times n + j] \times x[j]$
 9:         end if
10:     end for
11:     Gather all x from all nodes
12:     $x[i] -= x$
13: end for
14: pragma omp parallel for
15: for $i \leftarrow n - 1, 0$ do
16:     for $j \leftarrow i + 1, n$ do
17:         if $doIHaveA$ then
18:             compute $i_{local}$ and $j_{local}$
19:             $x += A[i_{local} \times n + j +_l ocal] \times x[j]$
20:         end if
21:     end for
22:     Gather all x from all nodes
23:     $x[i] += x$
24:     compute $i_{local}$
25:     $x[i] \leftarrow x[i]/A[i_{local} \times n + i_{local}]$
26: end for=0
```

matrix didn't give the correct result but it gives a good measure of computation speed improvement.

IV. Parallel performance

When assessing the performance of a parallel code is importance to determine the speed-up and the efficiency ' [1]. The efficiency speaks to how well is the code scaling up as the number of processor increase. The speed up and efficiency are calculated using equ. (**??**) and (12).

$$Efficiency = \frac{timeTakenOnOneProcessor}{timeTakenOnpprocessor \times p} \tag{12}$$

$$SpeedUp = \frac{timeTakenOnOneProcessor}{timeTakenOpprocessor} \tag{13}$$

Fig. 7 shows the time taken to compute LU as the number of nodes changes.

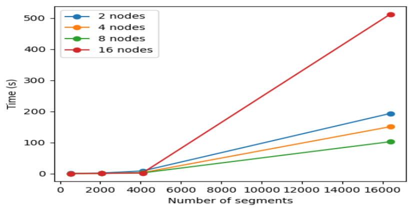

Fig. 7. Time taken to computer LU when varying the segments

From Fig. 7 shows that increasing number of nodes for small number of segments do not give a significant speed. As the number of number of nodes increases, the performance gain from using many number of nodes become more apparently.

Fig. **??** shows the speed up and the time taken as the number of nodes increase. for large matrix size of 16385, the speed

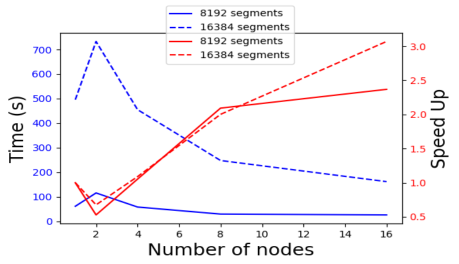

Fig. 8. LU speed up vs number of nodes

up for 16 nodes goes to 3. This graph show that the might be more speed up as the matrix increase.

Fig 1. below shows the matrix filling speed up and efficiency using maximum maximum number of threads and changing the number of nodes.

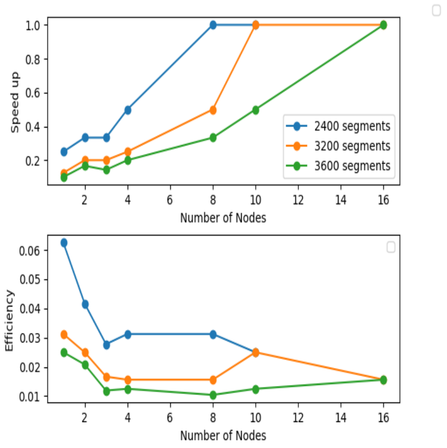

Fig. 9. Column Block distribution Time vs No. of Nodes

The matrix filling using cyclical distribution was not giving any speed up as the number of nodes increase. and changing the thread were giving inconsistent results because of race conditions. Thus we decided to experiment with column block fill so that we can convert it to cyclical block distribution after the matrix filling.

Fig . 10 shows the speed up and the efficiency of the whole for different number of machines and segments. the code is running with maximum number of threads per node.

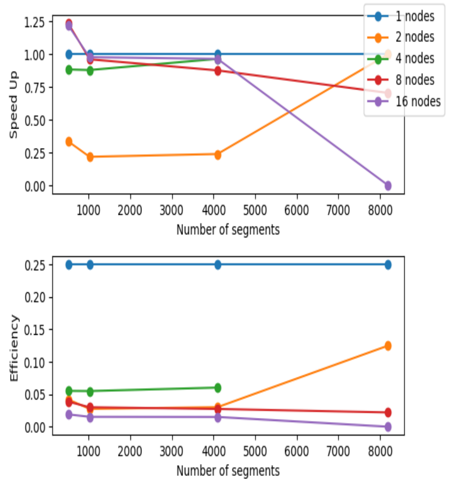

Fig. 10. The Whole NEC2++ speed up

From the graph, the whole code performace did not get improved when increasing the number of machine. The speed up is always below 1 for all different number of nodes and the efficiency is quite low. provided that the cyclical block distribution used when running the whole code gives the same computation time for different number of the machine. The bottle neck is coming from LU solve because it has a lot of communication. Few machines will have less communication compared to many nodes which result in relatively machining computation speed between single node and 16 nodes.

V. Future recommendation

Compute forward substitution while doing LU decomposition to reduce the communication bottleneck of solution solve since it would only have to do backward substitution. LU decomposition is done without pivoting, add pivoting to ensure LU stability.

VI. Conclusion

NEC2++ is parallelized in a distrubuted multicore architecture cluster by parallizing three intensive operation, namely: LU decomposition taking $O(N^3)$ operations, matrix filling and solving solution matrix which are both taking $O(N^2)$ operations. Matrix filling is parallelized using a cyclical distribution did improve the computation time. LU is parallelized using recursive block which allows a use of BLAS3 operations. LU decomposition showed a speed a good scaling when increasing number of processors. solution matrix solve parallezation has communication bottle neck which is deminishing the whole code perfomace.

VII. Project sustainability of the solution

VIII. Recommendation for future work

IX. Conclusion

References

[1] D.C. Nitch and A.P.C. Fourie. Parallel implementation of NEC. Applied Computational Electromagnetics Society Journal, 9(1):5157, 1994
[2] Rubinstein, Abraham, et al. ”A parallel implementation of NEC for the analysis of large structures.” IEEE Transactions on Electromagnetic Compatibility 45.2 (2003): 177-188.
[3]
[4] J. DU CROZ AND N. J. HIGHAM, Stability of methods for matrix inversion, IMA J. Numer. Anal., 12 (1992), pp. 1-19. (Also LAPACK Working Note 27)
[5] nec2.org, ”Numerical Electromagnics Code.” https://www.nec2.org/
[6] Wikipedia, ”Numerical Electromagnetics Code.” https://en.wikipedia.org/wiki/Numerical_Electromagnetics_Code/
[7] Dongarra, J J, Oak Ridge National Lab., TN, van de Geijn, R, and Walker, D W. A look at scalable dense linear algebra libraries. United States: N. p., 1992. Web. doi:10.2172/10164371.
[8] Timothy C.A. Molteno, ”NEC2++: An NEC-2 compatible Numerical Electromagnetics Code”, Electronics Technical Reports No. 2014-3, ISSN 1172-496X, October 2014.
[9] Almási, George, et al. ”Optimization of MPI collective communication on BlueGene/L systems.” Proceedings of the 19th annual international conference on Supercomputing. 2005.

X. Acknowledgements

References