Naval Research Laboratory
Washington, DC 20375-5320

U.S. NAVAL RESEARCH LABORATORY

NRL/5310/MR—2022/1

Development of Efficient Matrix Solution Algorithms Applicable to Electromagnetic Scattering and Radiation Problems

SADASIVA RAO

CHRISTOPHER WAHL

Radar Analysis Branch
Radar Division

March 15, 2022

DISTRIBUTION STATEMENT A: Approved for public release; distribution is unlimited.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing this collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden to Department of Defense, Washington Headquarters Services, Directorate for Information Operations and Reports (0704-0188), 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302. Respondents should be aware that notwithstanding any other provision of law, no person shall be subject to any penalty for failing to comply with a collection of information if it does not display a currently valid OMB control number. **PLEASE DO NOT RETURN YOUR FORM TO THE ABOVE ADDRESS.**

1. REPORT DATE *(DD-MM-YYYY)*
15-03-2022

2. REPORT TYPE
NRL Memorandum Report

3. DATES COVERED *(From - To)*
January 2021 – December 2021

4. TITLE AND SUBTITLE

Development of Efficient Matrix Solution Algorithms Applicable to Electromagnetic Scattering and Radiation Problems

5a. CONTRACT NUMBER

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER
PE-61153N

5d. PROJECT NUMBER

5e. TASK NUMBER
EW021-05-43

5f. WORK UNIT NUMBER
1P78

6. AUTHOR(S)

Sadasiva M. Rao and Christopher Wahl

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Naval Research Laboratory
4555 Overlook Avenue, SW
Washington, DC 20375-5320

8. PERFORMING ORGANIZATION REPORT NUMBER

NRL/5310/MR--2022/1

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Naval Research Laboratory
4555 Overlook Avenue, SW
Washington, DC 20375-5320

10. SPONSOR / MONITOR'S ACRONYM(S)

NRL

11. SPONSOR / MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION / AVAILABILITY STATEMENT

DISTRIBUTION STATEMENT A: Approved for public release; distribution is unlimited.

13. SUPPLEMENTARY NOTES

14. ABSTRACT

In this work, we present a set of new algorithms to efficiently solve matrix equation problems encountered in the application of the method of moments (MOM) to electromagnetic scattering and radiation problems. The new algorithms are based on parallel programing and innovative matrix partitioning schemes. Several numerical examples are presented and compared with conventional solution methods for validation purposes.

15. SUBJECT TERMS

Matrix solutions   Method of moments
Integral equations   Numerical methods

16. SECURITY CLASSIFICATION OF:			**17. LIMITATION OF ABSTRACT**	**18. NUMBER OF PAGES**	**19a. NAME OF RESPONSIBLE PERSON**
					Sadasiva Rao
a. REPORT	**b. ABSTRACT**	**c. THIS PAGE**	U	100	**19b. TELEPHONE NUMBER** *(include area code)*
U	U	U			(202) 404-1985

Standard Form 298 (Rev. 8-98)
Prescribed by ANSI Std. Z39.18

This page intentionally left blank.

CONTENTS

1. INTRODUCTION 1

2. OUTLINE OF THE METHOD OF MOMENTS 2

3. PARALLEL LU-DECOMPOSITION ALGORITHM 4
   3.1 Overview of the LU-Decomposition Algorithm 4
   3.2 Parallelization of LU Decomposition Algorithm 5
   3.3 Implementation on Parallel Processors 6
   3.4 Results 8
   3.5 Summary and Future Work 12

4. MULTI-LEVEL SOLUTION ALGORITHM 13
   4.1 Outline of the Power-Series Solution Method 13
   4.2 Development of Multi-Level Solution Algorithm 17
   4.3 An Alternate Procedure to Obtain the Multi-Level Solution 21
   4.4 Numerical Results 22

5. CONCLUSIONS 29

Appendices A1

REFERENCES

FIGURES

1 Electrically large conducting body illuminated by an X- polarized, Z-traveling plane wave. 2

2 Time versus vector length for arithmetic division. 8

3 Time versus vector length for arithmetic multiplication. 9

4 Time versus vector length for branching. 10

5 Time versus vector length for new branching. 11

6 RCS of a PEC sphere, radius= 3m, and illuminated by an X- polarized, Z-traveling 300 MHz plane wave. Number of levels =6, N=29,400. 23

7 RCS of a three-dimensional PEC cylinder, L = 50m and radius a = 0.1m, placed along the Z-axis and illuminated by an X-polarized, Z-traveling 300 MHz plane wave. Number of levels =5, N=18,018. 24

8 RCS of a three-dimensional PEC cylinder, L = 100m and radius a = 0.1m, placed along the Z-axis and illuminated by an X-polarized, Z-traveling 300 MHz plane wave. Number of levels =6, N=36,018. 25

9 An aircraft-like model. The dimensions of the model along the X, Y, and Z axes are 5.85m, 3.5m, and 1.76m, respectively. 26

10 RCS of an aircraft-like model. The dimensions of the model along the X, Y, and Z axes are 5.85m, 3.5m, and 1.76m, respectively. The model is placed such that the geometrical center roughly coincides with the center of the coordinate system. The model is illuminated by an X-polarized, Z-traveling 300 MHz plane wave. Number of levels =5, N=10,692. 27

11 RCS of a ship-like model. The dimensions of the model along the X, Y, and Z axes are 30m, 2.6m, and 4.0m, respectively. The model is placed such that the geometrical center roughly coincides with the center of the coordinate system. The model is illuminated by an X-polarized, Z-traveling 300 MHz plane wave. Number of levels =7, N=54,408. 28

DEVELOPMENT OF EFFICIENT MATRIX SOLUTION ALGORITHMS APPLICABLE TO ELECTROMAGNETIC SCATTERING AND RADIATION PROBLEMS

1. INTRODUCTION

The method of moments (MOM) solution technique [1–5] , one of the most popular methods for solving electromagnetic scattering and radiation problems, is limited by excessive computational and memory requirements if applied in a conventional way to electrically large scattering problems. As described in the next chapter, the MOM transforms a mathematical equation into a matrix equation to be solved on a digital computer. Unfortunately, for many practical applications, the dimension of the resulting matrix is very large, sometimes running into millions, imposing a technological limitation by requiring excessive memory and computation time. The objective of the present work is to develop methods to minimize this limitation and apply the procedure to possibly more complex and practical problems.

There exist several new algorithms, which seem to overcome the limitation of the MOM. Notable among those are: a) recently developed fast multipole method (FMM) using matrix approximations to reduce the memory and computational time requirements [6], and adaptive cross approximation (ACA) method [7]. However, both are approximate methods and the accuracy of the overall solution depends on the approximation process and may become expensive when high accuracy solutions are desired. Further, FMM also sacrifices one of the most important advantages of the MOM, the ability to handle multiple right hand sides in a simple manner.

In this work, we describe two algorithms; a) Parallel LU-Decomposition Algorithm and b) the Multi-Level Solution Algorithm; we have developed to solve excessively large matrices. The first algorithm is based on using parallel processing schemes and may be an e˙ective way to solve a very large matrix equation. The present day technology in the computer industry allows thousands of cores on standard workstations, and the the new algorithm tries to exploit this technological advancement.

The second algorithm is an improvement on the recently developed power series solution method [8], and results in a multi-level scheme, wherein at each level, only a small portion of the matrix is utilized.

This report is organized as follows: In the following chapter, we present an overview of the mathematical steps to transform the standard electromagnetic integral equations, derived by applying the boundary conditions for a conducting body, into a matrix equation. In Chapter 3, we present the detailed description of the Parallel LU-Decomposition Algorithm along with some representative numerical results. We also include the source code of the algorithm. In Chapter 4, we present the Multi-Level Solution Algorithm along with representative numerical results. Finally, Chapter 5 discusses important conclusions along with possible improvements and future work to be undertaken in this area.

Manuscript approved February 24, 2022.

2. OUTLINE OF THE METHOD OF MOMENTS

Consider an electrically large, perfectly electric conductor (PEC) problem excited by an incoming plane wave as shown in Fig. 1. Here, the term “electrically large” implies that the physical dimension of the scatterer is large compared to the wavelength of the incident field.

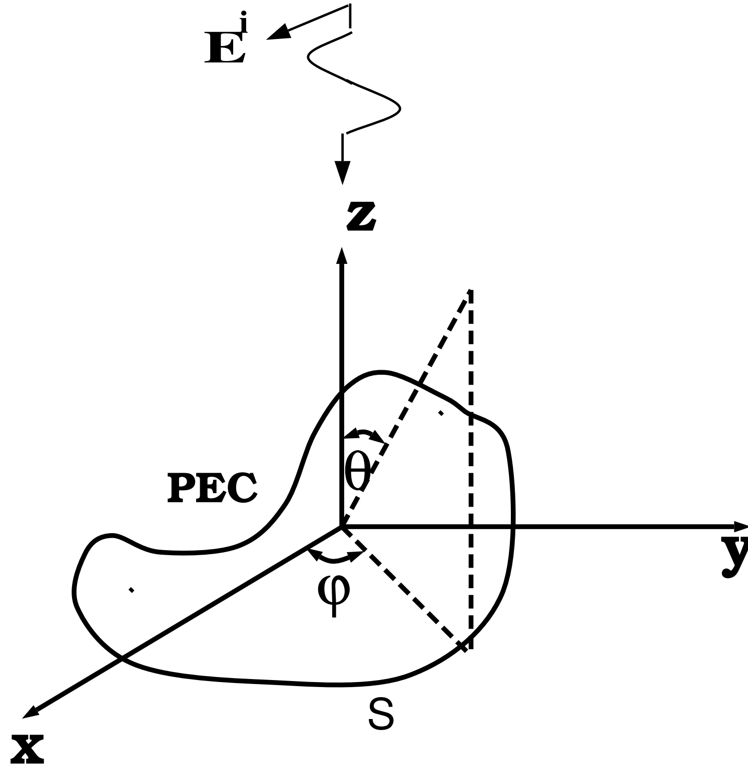

Fig. 1—Electrically large conducting body illuminated by an X- polarized, Z-traveling plane wave.

By enforcing the boundary conditions on electric and magnetic fields, the following integral equations are derived in a standard manner, given by

$$\mathbf{E}^{i}_{tan}(\mathbf{r}) + \mathbf{E}^{s}_{tan}(\mathbf{r}) = 0, \ \mathbf{r} \in S \tag{1}$$

and

$$\mathbf{J}(\mathbf{r}) = \hat{n} \times (\mathbf{H}^{i}(\mathbf{r}) + \mathbf{H}^{s}(\mathbf{r})), \ \mathbf{r} \in S \tag{2}$$

where $(\mathbf{E}^i, \mathbf{H}^i)$ and $(\mathbf{E}^s, \mathbf{H}^s)$ are the incident and scattered fields, respectively, $\mathbf{J}$ is the induced current, S is the conductor surface, $\hat{n}$ represents the unique outward normal to S, and the subscript ”*tan*” represents the tangential component. The electromagnetic scattering problem represented by Eqs. (1) and (2) , in general terms, are symbolically written as,

$$\mathbf{LI} = \mathbf{V}, \tag{3}$$

where **L**, **I**, and **V**, are the integral equation operator, unknown vector describing the induced currents, and known excitation operator representing the incident field, respectively. The integral equation operator adopted in this work is the combined field operator given by,

$$\mathbf{L}_{CFIE} = \gamma\,\mathbf{L}_{EFIE} + \eta(1-\gamma)\,\mathbf{L}_{MFIE}, \tag{4}$$

$$\mathbf{Y}_{CFIE} = \gamma\,\mathbf{Y}_{EFIE} + \eta(1-\gamma)\,\mathbf{Y}_{MFIE}, \tag{5}$$

where $0 \leq \gamma \leq 1$ is a constant and η is the intrinsic impedance of the medium. For open body problems, we let $\gamma = 1$, and for closed bodies γ is typically 0.5. In Eq. (4), the quantities $\mathbf{L}_{EFIE}$, $\mathbf{L}_{MFIE}$ represent the mathematical operators obtained by enforcing Eqs. (1) and (2), respectively. Also, $\mathbf{Y}_{EFIE}$ and $\mathbf{Y}_{MFIE}$ denote the mathematical representations of the incident electric and magnetic fields, respectively.

To begin the numerical procedure, we follow the conventional MOM and approximate the current distribution on the given PEC structure using standard subdomain functions. The first step involves approximating the given structure via standard geometrical discretization, *viz.* triangular patches, and defining the conventional Rao-Wilton-Glisson (RWG) basis functions to approximate the induced current [9].

Next, we order the basis functions using a distance criterion measured from a reference point, either from one end of the scatterer or from a convenient point on the scatterer. This arrangement makes the first basis function closest to the reference point and the last one the farthest. This is a very important step for both algorithms as presented in [8]. We also note that the actual location of the reference point is of no importance and does not affect the final results in any way.

Using the RWG basis functions as testing functions and following the standard MOM procedure, the operator equations in (4) and (5) are transformed into a matrix equation given by,

$$\mathbf{ZX} = \mathbf{Y}. \tag{6}$$

where **Z** is an $N \times N$ matrix, and **X** and **Y** are the column vectors of dimension N, respectively. Note that, because of the re-ordering of the basis functions as mentioned before, in each row of the **Z**-matrix, the diagonal element is the largest element magnitude-wise, and off-diagonal elements progressively decrease away from the diagonal element.

The next step in the application of the MOM is to solve the matrix equation (6) to obtain the unknown vector, **X**. For simple and electrically small problems, the dimension of the **Z**-matrix is on the order of $10^3 - 10^4$ and readily solved by any standard linear equation solver such as LU-Decomposition [10], or by any suitable iteration solver such as Conjugate Gradient method [11] on a modern digital computer. However, for complex problems of Navy interest, the same matrix may run on the order of $10^5 - 10^8$ and imposes a heavy burden on the available computational resources. In the next two chapters, we explore ways to mitigate this problem and develop solution algorithms that attempt to overcome the limitation.

3. PARALLEL LU-DECOMPOSITION ALGORITHM

In this chapter, we present a parallel version of the LU-Decomposition algorithm applicable to the solution of simultaneous equations. Note that the LU-Decomposition algorithm is a widely used algorithm to solve the matrix equation. However, it is effective and efficient for serial processing systems and inefficient for the new generation of parallel processors. Even for the serial system, the LU method requires the full storage of the matrix and the computation time is proportional to $\bigcirc(N^3)$ where N is the size of the matrix. Obviously, developing more efficient schemes reduces the computational burden enabling solutions to more complex problems. In the next section, we present a short overview of the LU-Decomposition algorithm and, in the subsequent section, we describe the parallelization scheme along with numerical results.

3.1 Overview of the LU-Decomposition Algorithm

In numerical analysis and linear algebra, the LU-Decomposition algorithm is a very useful tool to solve a system of equations. Initially, the system of equations are cast into a matrix equation form, $\mathbf{ZX} = \mathbf{Y}$ where $\mathbf{Z}$ is known as the system matrix. The term LU- Decomposition stands for resolving the system matrix, $\mathbf{Z}$, into a product of two matrices where one matrix, $\mathbf{L}$, is strictly zero above the diagonal, whereas the other matrix, $\mathbf{U}$, is strictly zero below the diagonal. Thus, we have

$$\mathbf{Z} = \mathbf{L}\,\mathbf{U} \tag{7}$$

where the elements in the $\mathbf{L}$-matrix above the diagonal are strictly zero and in $\mathbf{U}$, the elements below the diagonal are strictly zero. A simple decomposition of a 3×3 matrix is written as

$$\begin{bmatrix} \mathbf{Z}_{11} & \mathbf{Z}_{12} & \mathbf{Z}_{13} \\ \mathbf{Z}_{21} & \mathbf{Z}_{22} & \mathbf{Z}_{23} \\ \mathbf{Z}_{31} & \mathbf{Z}_{32} & \mathbf{Z}_{33} \end{bmatrix} = \begin{bmatrix} \mathbf{L}_{11} & \mathbf{O} & \mathbf{O} \\ \mathbf{L}_{21} & \mathbf{L}_{22} & \mathbf{O} \\ \mathbf{L}_{31} & \mathbf{L}_{32} & \mathbf{L}_{33} \end{bmatrix} \begin{bmatrix} \mathbf{U}_{11} & \mathbf{U}_{12} & \mathbf{U}_{13} \\ \mathbf{O} & \mathbf{U}_{22} & \mathbf{U}_{23} \\ \mathbf{O} & \mathbf{O} & \mathbf{U}_{33} \end{bmatrix} \tag{8}$$

Before generating this product, the rows and columns of the $\mathbf{Z}$ matrix are interchanged invariably, searching for the pivot element. We define the pivot element as the largest term, magnitude wise, in the given row or column and, finally, in the whole matrix. We view LU decomposition as the matrix form of Gaussian elimination. Computers usually solve square systems of linear equations using LU decomposition, and it is also a key step when inverting a matrix or computing the determinant of a matrix. The LU decomposition is a standard algorithm and widely available in research papers [10], textbooks [12], and standard computer software packages (http://www.netlib.org/lapack/). However, for the sake of completeness, we provide a short description of the process of decomposition.

As a first step, partition $\mathbf{Z}$, $\mathbf{L}$, and $\mathbf{U}$ as follows:

Let

$$\mathbf{Z} = \begin{bmatrix} z_{11} & \mathbf{Z}_{12}^T \\ \mathbf{Z}_{21} & \mathbf{Z}_{22} \end{bmatrix}, \quad \mathbf{L} = \begin{bmatrix} 1 & 0 \\ \mathbf{L}_{21} & \mathbf{L}_{22} \end{bmatrix}, \quad \text{and} \quad \mathbf{U} = \begin{bmatrix} u_{11} & \mathbf{U}_{12}^T \\ 0 & \mathbf{U}_{22} \end{bmatrix} \tag{9}$$

Now, setting $\mathbf{Z} = \mathbf{LU}$ we have

$$\begin{bmatrix} z_{11} & \mathbf{Z}_{12}^T \\ \mathbf{Z}_{21} & \mathbf{Z}_{22} \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ \mathbf{L}_{21} & \mathbf{L}_{22} \end{bmatrix} \begin{bmatrix} u_{11} & \mathbf{U}_{12}^T \\ 0 & \mathbf{U}_{22} \end{bmatrix} \tag{10}$$

$$= \begin{bmatrix} u_{11} & \mathbf{U}_{12}^T \\ \mathbf{L}_{21} u_{11} & \mathbf{L}_{21}\mathbf{U}_{12}^T + \mathbf{L}_{22}\mathbf{U}_{22} \end{bmatrix} \tag{11}$$

Hence, we have

$$\begin{aligned} z_{11} &= u_{11} & \mathbf{Z}_{12}^T &= \mathbf{U}_{12}^T \\ \mathbf{Z}_{21} &= u_{11}\mathbf{L}_{21} & \mathbf{Z}_{22} &= \mathbf{L}_{21}\mathbf{U}_{12}^T + \mathbf{L}_{22}\mathbf{U}_{22} \end{aligned} \tag{12}$$

or equivalently

$$\begin{aligned} z_{11} &= u_{11} & \mathbf{Z}_{12}^T &= \mathbf{U}_{12}^T \\ \mathbf{Z}_{21} &= u_{11}\mathbf{L}_{21} & \mathbf{Z}_{22} - \mathbf{L}_{21}\mathbf{U}_{12}^T &= \mathbf{L}_{22}\mathbf{U}_{22} \end{aligned} \tag{13}$$

If we overwrite the upper triangular part of $\mathbf{Z}$ with $\mathbf{U}$ and the strictly lower triangular part of $\mathbf{Z}$ with the strictly lower triangular part of $\mathbf{L}$ (since we know that the diagonal consists of ones), we deduce that we must perform the computations $\mathbf{Z}_{21} = \mathbf{Z}_{21}/z_{11}$ and $\mathbf{Z}_{22} = \mathbf{Z}_{22} - \mathbf{Z}_{21}\mathbf{Z}_{12}^T$.

Now, we continue the same process for $\mathbf{Z}_{22}$ until we reach z_{nn}.

3.2 Parallel LU Decomposition Algorithm

To efficiently adopt the algorithm to parallel processing, we adopt the right looking block form of the LU decomposition algorithm, and applied timing tests to each block. An optimization was found for the row-wise partial pivoting by comparing the magnitudes of each element in a given row and selecting the largest element. Obviously, the same process can be adopted for column-wise pivoting. In the following, we present the details:

3.2.1 Message Passing Interface(MPI)

The parallelization scheme looked at in this work is the Message Passing Interface, referred to as MPI. MPI is a single program, multiple data (SPMD) parallelization with data sharing done via network message passing. A manager/worker scheme was utilized where the manager processes the loads and distributes data across the worker processes.

3.2.2 Right Looking Block LU Decomposition

The right looking block LU decomposition is a separated form of the normal LU decomposition where the algorithm is split into its fundamental, same operation components as shown in the following:

$$\mathbf{Z} = \begin{bmatrix} \mathbf{U}_{m,n} & \mathbf{U}_m \\ \mathbf{L}_n & \mathbf{Z}'_x \end{bmatrix} \tag{14}$$

where $\mathbf{Z} = \mathbf{Z}'_{x-1}$. Each part of the above matrix refers to a block in the algorithm that was parallelized. Furthermore, before each iteration of the decomposition, a partial pivot was run to find the local maximum in the first column of $\mathbf{Z}$ and switch the max row with the top r ow. Also, a right looking block LU-Parallel (LUP) algorithm was written where instead of permuting the rows during the partial pivots, the row indexes are stored in a new vector, P.

3.3 Implementation on Parallel Processors

We developed several algorithms for testing and analysis and describe the implementation via a simple numerical example.

To test our algorithms, we developed a matrix equation $\mathbf{AX} = \mathbf{Y}$ where $\mathbf{A}$ is an $N \times N$ matrix, and $\mathbf{X}$ and $\mathbf{Y}$ are unknown and known vectors of dimension N, respectively. The elements of the $\mathbf{A}$-matrix and $\mathbf{Y}$-vector are

$$a_{m,n} = \frac{1.0}{(|m-n|+1) + j\,(|m-n|+1)} \tag{15}$$

$$y_m = \sum_{n=1}^{N} a_{m,n}. \tag{16}$$

Note that the entries of the matrix are complex in nature, which makes the matrix $\mathbf{A}$ a complex matrix. To obtain a real matrix, we set the imaginary part to zero for each entry.

The exact solution of this system of equations is $x_m = 1.0$, for $m = 1, 2, \cdots, N$. Since we already know the solution, this system of equations will serve as a way to check our algorithm and perform the efficiency tests.

3.3.1 Description of Algorithms

- LU Modules: All the **LU**-modules have entry functions and their respective manager/worker functions. Entry functions take in the **A** and **L** matrices, and N, describing the dimensionality. Only the manager must have the memory/matrices allocated and set up. The workers just receive information from the manager. The final **U** matrix is stored in the **A** matrix given as a parameter to save memory. Note that calling the entry function without MPI (or only 1 process) will simply run the algorithm in serial.
- Matrix Tests: To quickly test the LU functions, there are a group of matrix set up functions inside the "matrixTests.f90" file. These functions take in unallocated **A** and **L** matrices, an unallocated integer **P**-matrix (for use with the Pivot vector LUP algorithm), and an integer, N, for the dimensionality. The **P**-matrix is filled inside the function.
- To check if the LU algorithm is working, the test matrices use Eqs. (15) and (16) to generate **A** and **Y.** A simple program to perform the matrix fill for 25,000 unknowns is as follows:

```
complex, allocatable :: A(:, :), L(:, :)
integer, allocatable :: P(:)
integer :: N
call fillMatrixTest9_25000x25000(A, L, P, N)
```

- Solving the LU system with the Z_vec given from "zVectorComplex" inside the "matrixMakers.f90" file should return the X_vec as a 1s vector. The solvers, "lowerTriangularMatVecSolve" and "upperTriangularMatVecSolve", are found in the "matrixSolvers.f90" file and the function to make sure the X_vec is all 1s, "vecConstantCheck", is found in the "matrixUtil.f90" file. This check set up is found inside the "MPILUComplex()" and "OptimizedLUTest()" functions inside the "main.f90" file.
- The main.f90 File: This file contains subroutines that run the different LU tests, namely:
  1. serialLUPTest() -> The serial LUP algorithm that uses a pivot vector, P, to store permutations
  2. MPILUTest() -> The MPI capable LU algorithm using real numbers
  3. MPILUComplexTest() -> The MPI capable LU algorithm using complex numbers
  4. OptimizedLUTest() -> The MPI capable LU algorithm using complex numbers with only the pivot using MPI

  These functions have several fillMatrixTest_ functions set up that are exchanged by commenting in/out the test that needs to be run. The timed or not timed version of the LU algorithms are selected using the correct entry functions from the files listed above.

3.3.2 Timing Tests

First, two forms of the LU decomposition algorithm were written: a) Right looking block LUP decomposition and b) MPI Right looking block LU decomposition. Timing tests were run between the LUP, 1 core MPI LU (or serial without the P vector), 4 cores MPI LU, and 12 cores MPI LU.

Second, after investigating the initial tests, we found more cores did not directly result in faster run times with a 25k by 25k matrix. A more thorough look at each block of the algorithm, including the pivot, was needed.

To do this, different functions were used to represent the major parts of the algorithm. The left block is a scalar division, the bottom right block is a scalar multiplication, and the pivoting uses branching/if-statements on every value. Functions representing each block were written that iterated over a vector. Furthermore, due to the bottom left and bottom right blocks utilizing more than one memory array, each operation accesses two different vectors as shown below.

- nums(:), *some complex vector that is transmitted by the Manager processor*.
- vec(:), *a complex vector pre-stored on each processor*.
- const, *a complex variable pre-stored on each processor*.
- Division function: nums(i) = nums(i) −vec(i) / const
- Multiply function: nums(i) = nums(i)− vec(i) * const
- Branch function: Max(nums)

Two tests with different size vectors were conducted. Each test ran the functions 1000 times, recorded the times to run the computation, and then averaged them. Also, the multiplication represents an $\bigcirc(1)$ time operation, division is an iterative operation, and branching is an unpredictable operation that hinders certain pipeline optimizations.

3.4 Results

Initial results showed serial computation was faster for all scalar division and multiplication. However, performing an operation that requires branching on every value, such as finding the maximum value in a vector by checking if the current index is larger than the previous largest value, could be accelerated with MPI.

3.4.1 Arithmetic Division

The graph in Fig. 2 compares the serial and 12-core MPI parallelization of using division between two vectors of complex values. The parallelization takes longer than the main processor computing the entire vector. We note that the division is an iterative operation.

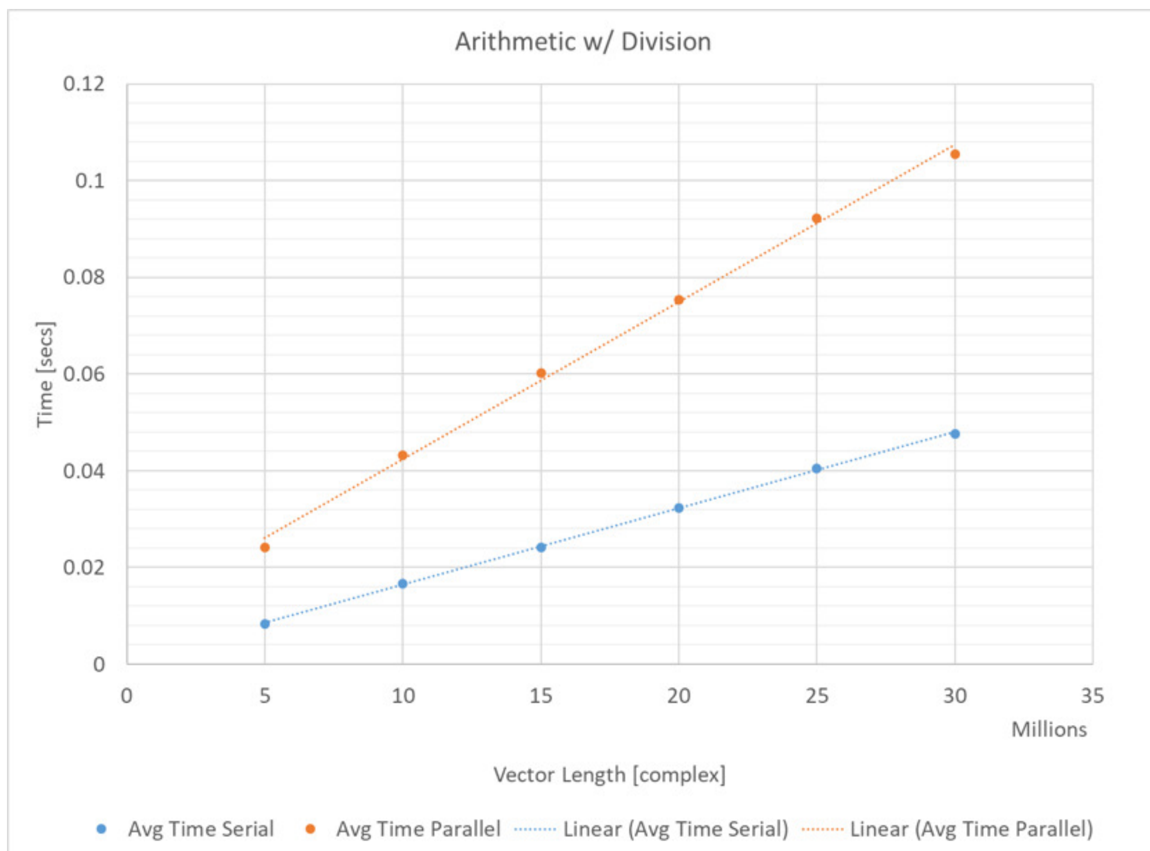

Fig. 2—Time versus vector length for arithmetic division.

3.4.2 Arithmetic Multiplication

The graph in Fig. 3 compares the serial and 12-core MPI parallelization of using multiplication between two vectors of complex values. The serial computation is always faster than the parallelization computation. Multiplication is an O(1) operation as demonstrated by the significantly flatter "Avg Time Serial" line. We note that the parallelized division and multiplication lines are very close, suggesting most of the time spent is used sending the vector to the other cores.

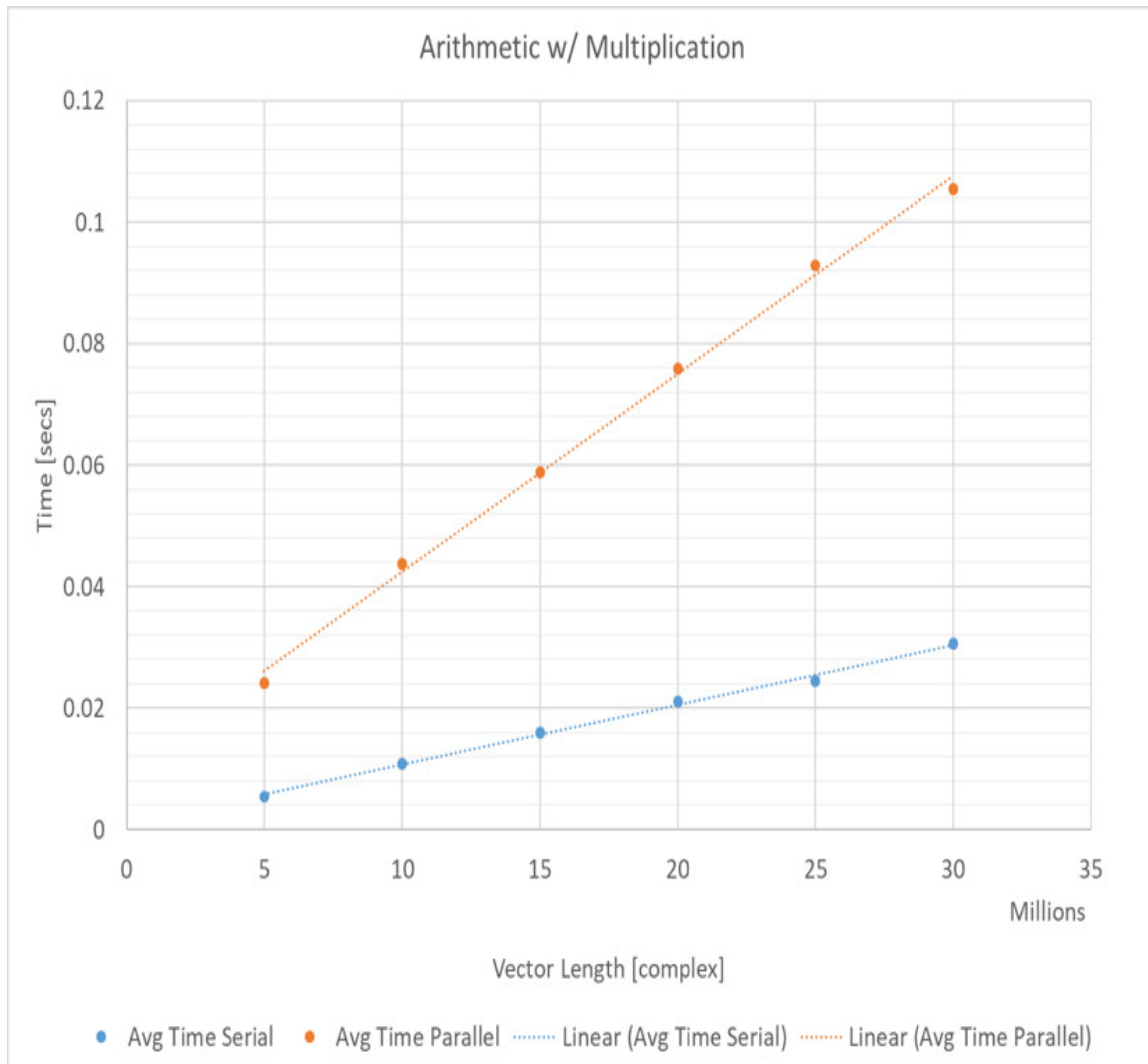

Fig. 3—Time versus vector length for arithmetic multiplication.

3.4.3 Branching

The graph in Fig. 4 compares the time necessary to find the maximum absolute value of a complex vector in serial and 12-core MPI parallelization. While serial is initially faster, parallelization surpasses it with a vector of nominally 10 million values. Finding the maximum requires using an if-statement branch on every value, an operation that pipelining has a hard time optimizing, causing a significantly slower runtime despite sharing an O(n) vector operation time complexity like the other two tests. We note that, while the serial timing is very linear, the parallelized time curves at higher values.

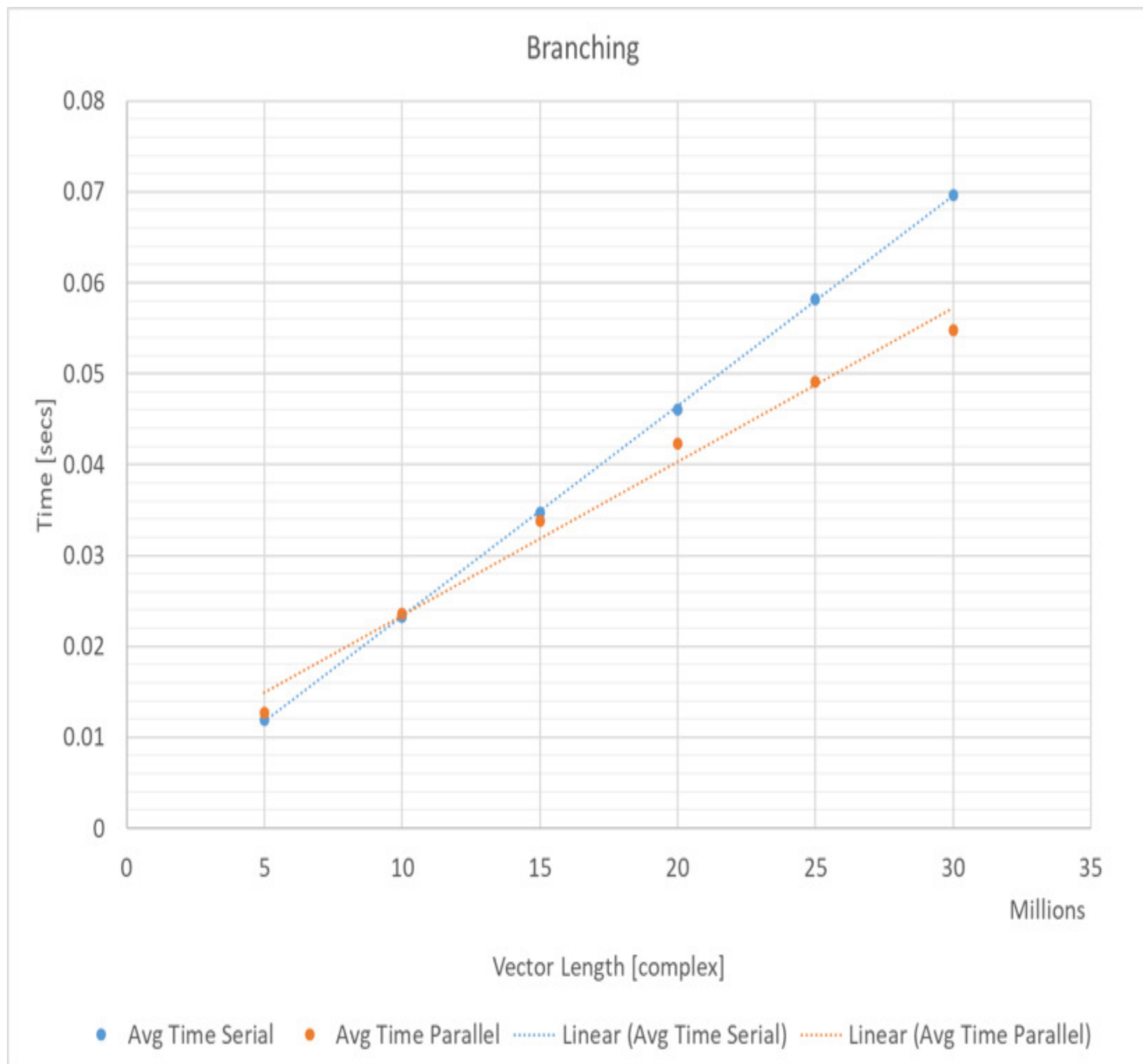

Fig. 4—Time versus vector length for branching.

The parallelized branching starts out slower, but as more values need to be branched, it becomes faster. The switch between serial to parallel happens around 10M values. However, in an attempt to find a more accurate optimization point, several more tests were done running each vector length 10 thousand times and averaging the total compute time. In the second round of tests, both the parallel and series branching ran

faster. The parallel branching ran in a similar time span, just a couple milliseconds faster. However, the series branching was running 25% faster, exceeding the parallel branching.

We note that in the multiplication and division cases the MPI transmission seems to be a very large portion of the operation time due to the significantly steeper slope between the parallel and series cases. But for the branching, the slopes between the parallelized operations and series operations are almost parallel. We believe that MPI can be used to accelerate other algorithms with a slightly higher degree of parallelism or with non-one-step operations (as opposed to LU's single multiplication, single division, or single branch).

After seeing the first branching results showing a significant increase in speed in finding the maximum, a more granular test was run to find exactly when it becomes beneficial to implement the parallelized tournament pivot as shown in Fig. 5. However, the results could not be recreated and running the whole function in serial was faster. Though the parallel scheme is near parallel to the serial time and even curves at larger vectors.

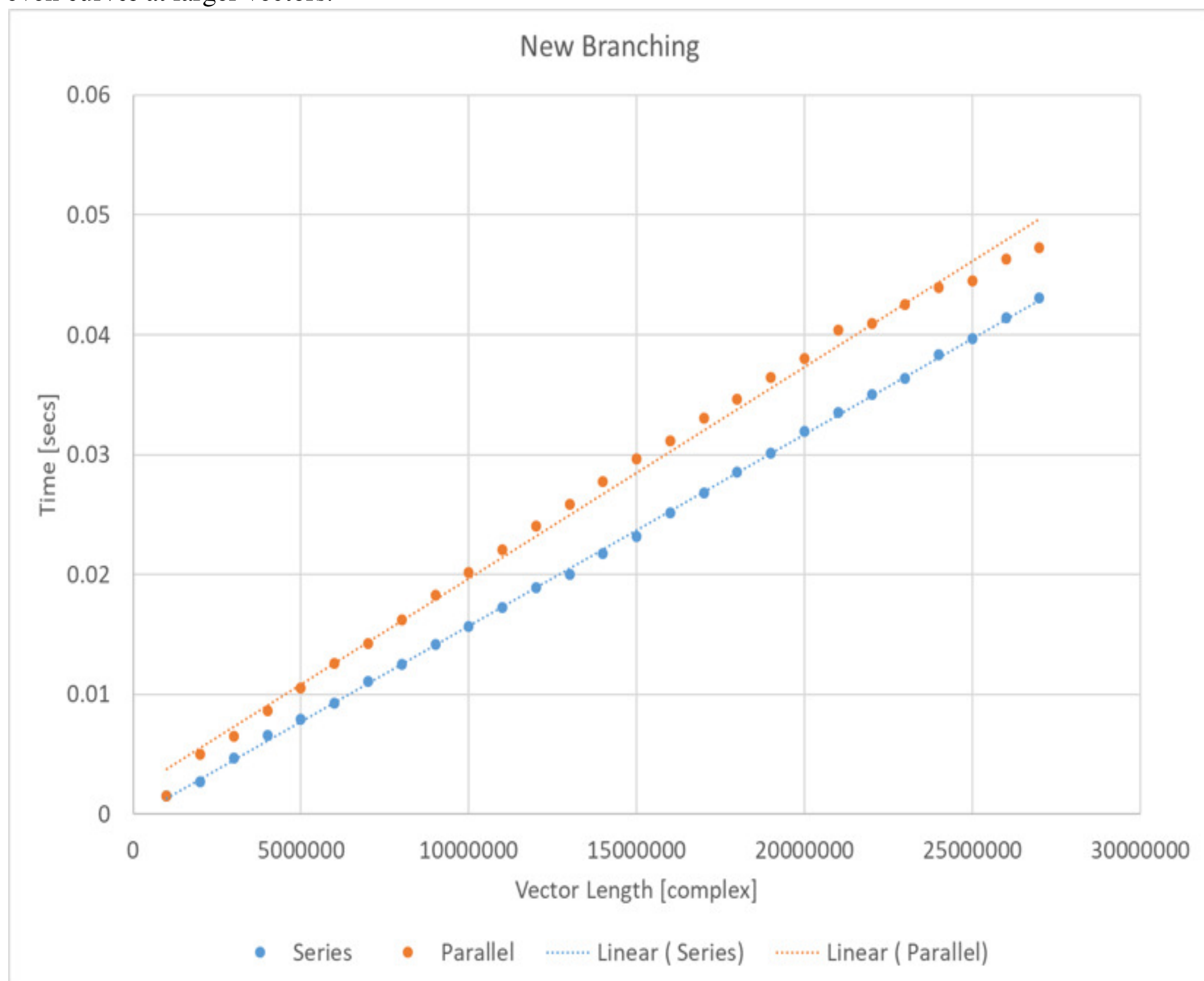

Fig. 5—Time versus vector length for new branching.

3.5 Summary and Future Work

We used MPI to try to accelerate the LU decomposition algorithm. MPI's most time consuming parallel block is the partial pivot where a branch operation must be conducted over every value. While in initial testing, MPI was able to accelerate the partial pivot additional testing showed there was no increase in speed. We need to examine this with more examples and further testing to see whether striding over the processors final max values saves additional time.

Analysis of the parallelization of the different blocks in the Right Looking LU Algorithm shows a potential speed increase of the partial pivot using a tournament pivoting scheme. Initial tests showed a speed for over 10 million values while secondary testing showed a slower time shift from the serial computation. We believe that functions containing a branch can be potentially sped up using the MPI and further testing is needed to determine if a different memory sharing scheme increases the speed of the tournament pivot. We intend to carry out this work further and report the results in due course.

4. MULTI-LEVEL SOLUTION ALGORITHM

In this chapter, we present our Multi-Level Solution Algorithm that appears to reduce the computational burden while solving the matrix equation. The advantage is that the whole matrix is never stored in the computer and only computed as required. Hence, the algorithm requires fewer resources as compared to the standard matrix inversion algorithms. We advise storing the matrix to reduce the computation time considerably.

The Multi-Level Solution Algorithm is based on the previously developed power-series solution method [8] and, in a broad sense, may be considered as an improvement to that method. Hence, in the next section, we present an outline of the power-series method for the sake of completeness, and in the subsequent section, we discuss the development of the Multi-Level Solution Algorithm.

4.1 Outline of the Power-Series Solution Method

In the development of the power-series solution method, initially we perform the same mathematical steps as the MOM and develop the matrix equation. However, before we begin the development of the matrix equation, we assemble the total number of basis functions into a fixed number of groups with each group containing a fixed number of micro-basis functions. Let us divide the N basis functions into P groups with $M = N/P$ elements in each group. Here, we note that each group corresponds to a block of elements in the global MOM matrix. Then, the **Z**-matrix may be symbolically written as:

$$\mathbf{Z} = \begin{bmatrix} \mathbf{Z}_{11} & \mathbf{Z}_{12} & \mathbf{Z}_{13} & \cdots & \mathbf{Z}_{1P} \\ \mathbf{Z}_{21} & \mathbf{Z}_{22} & \mathbf{Z}_{23} & \cdots & \mathbf{Z}_{2P} \\ \mathbf{Z}_{31} & \mathbf{Z}_{32} & \mathbf{Z}_{33} & \cdots & \mathbf{Z}_{3P} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ \mathbf{Z}_{P1} & \mathbf{Z}_{P2} & \mathbf{Z}_{P3} & \cdots & \mathbf{Z}_{PP} \end{bmatrix}, \tag{17}$$

where each $\mathbf{Z}_{ij}, i = 1, 2, \cdots, P, j = 1, 2, \cdots, P$ represents a submatrix of $M \times M$. In a similar manner, we express the column vector **Y** as:

$$\mathbf{Y} \quad = \quad [\mathbf{Y}_1, \mathbf{Y}_2, \mathbf{Y}_3, \cdots, \mathbf{Y}_P]^T, \tag{18}$$

where the superscript “T” represents the transpose.

Next, we transform Eq. (6) to

$$\tilde{\mathbf{Z}}\mathbf{X} = \tilde{\mathbf{Y}} \tag{19}$$

where $\tilde{\mathbf{Z}} = \mathbf{R}_1\mathbf{Z}$, $\tilde{\mathbf{Y}} = \mathbf{R}_1\mathbf{Y}$, and

$$\mathbf{R}_1 \quad = \quad \begin{bmatrix} \mathbf{I} & \mathbf{R}_{12} & \mathbf{R}_{13} & \mathbf{O} & \mathbf{O} & \mathbf{O} \\ \mathbf{O} & \mathbf{I} & \mathbf{O} & \mathbf{O} & \mathbf{O} & \mathbf{O} \\ \mathbf{O} & \mathbf{O} & \mathbf{I} & \mathbf{O} & \mathbf{O} & \mathbf{O} \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ \mathbf{O} & \mathbf{O} & \mathbf{O} & \mathbf{O} & \cdots & \mathbf{I} \end{bmatrix}. \tag{20}$$

$\mathbf{R}_{12}$ and $\mathbf{R}_{13}$ are $M \times M$ matrices with unknown coefficients, and $\mathbf{I}$ and $\mathbf{O}$ are $M \times M$ identity and null matrices, respectively.

Considering the first row of the $\tilde{\mathbf{Z}}$-matrix, using standard matrix multiplication procedures, we have:

$$\begin{aligned}
\tilde{\mathbf{Z}}_{11} &= \mathbf{Z}_{11} + \mathbf{R}_{12}\mathbf{Z}_{21} + \mathbf{R}_{13}\mathbf{Z}_{31} \\
\tilde{\mathbf{Z}}_{12} &= \mathbf{Z}_{12} + \mathbf{R}_{12}\mathbf{Z}_{22} + \mathbf{R}_{13}\mathbf{Z}_{32} \\
\tilde{\mathbf{Z}}_{13} &= \mathbf{Z}_{13} + \mathbf{R}_{12}\mathbf{Z}_{23} + \mathbf{R}_{13}\mathbf{Z}_{33} \\
\vdots &\ \vdots\ \vdots \\
\tilde{\mathbf{Z}}_{1P} &= \mathbf{Z}_{1P} + \mathbf{R}_{12}\mathbf{Z}_{2P} + \mathbf{R}_{13}\mathbf{Z}_{3P}.
\end{aligned}$$

Next, we solve for $\mathbf{R}_{12}$ and $\mathbf{R}_{13}$ by forcing the elements of $\tilde{\mathbf{Z}}_{12}$ and $\tilde{\mathbf{Z}}_{13}$ to zero. Thus, we solve

$$\mathbf{Z}_{12} + \mathbf{R}_{12}\mathbf{Z}_{22} + \mathbf{R}_{13}\mathbf{Z}_{32} = 0, \tag{21}$$

$$\mathbf{Z}_{13} + \mathbf{R}_{12}\mathbf{Z}_{23} + \mathbf{R}_{13}\mathbf{Z}_{33} = 0, \tag{22}$$

simultaneously, which results in a solution of a $2M \times 2M$ matrix with M right hand sides. Once $\mathbf{R}_{12}$ and $\mathbf{R}_{13}$ are known, obtaining $\tilde{\mathbf{Y}}$ is trivial. Note that the procedure described so far sets the interaction between groups 1 and 2 ($\tilde{\mathbf{Z}}_{12}$) and between groups 1 and 3 ($\tilde{\mathbf{Z}}_{13}$) to zero, and makes $\tilde{\mathbf{Z}}_{11}$ dominant block in the row.

By applying a similar procedure to rows $2, 3, \cdots, P$ and each time solving a $2M \times 2M$ matrix, we generate a new matrix equation

$$\overline{\mathbf{Z}}\mathbf{X} = \overline{\mathbf{Y}}, \tag{23}$$

where the new $\overline{\mathbf{Z}}$-matrix is given by

$$\begin{bmatrix}
\tilde{\mathbf{Z}}_{11} & \mathbf{O} & \mathbf{O} & \cdots & \tilde{\mathbf{Z}}_{1,P-2} & \tilde{\mathbf{Z}}_{1,P-1} & \tilde{\mathbf{Z}}_{1P} \\
\mathbf{O} & \tilde{\mathbf{Z}}_{22} & \mathbf{O} & \cdots & \tilde{\mathbf{Z}}_{2,P-2} & \tilde{\mathbf{Z}}_{2,P-1} & \tilde{\mathbf{Z}}_{2P} \\
\tilde{\mathbf{Z}}_{31} & \mathbf{O} & \tilde{\mathbf{Z}}_{33} & \cdots & \tilde{\mathbf{Z}}_{3,P-2} & \tilde{\mathbf{Z}}_{3,P-1} & \tilde{\mathbf{Z}}_{3P} \\
\vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\
\tilde{\mathbf{Z}}_{P1} & \tilde{\mathbf{Z}}_{P2} & \tilde{\mathbf{Z}}_{P3} & \cdots & \mathbf{O} & \mathbf{O} & \tilde{\mathbf{Z}}_{PP}
\end{bmatrix}$$

and

$$\overline{\mathbf{Y}} = \left[\tilde{\mathbf{Y}}_1, \tilde{\mathbf{Y}}_2, \tilde{\mathbf{Y}}_3, \tilde{\mathbf{Y}}_4, \cdots, \tilde{\mathbf{Y}}_P\right]^T.$$

We note that the $\overline{\mathbf{Z}}$-matrix in Eq. (23) represents a diagonally-dominant matrix assuming a sufficient number of basis functions are collected in a group.

Next, let $\overline{\mathbf{Z}} = \overline{\mathbf{Z}}_d + \overline{\mathbf{Z}}_{off}$ where $\overline{\mathbf{Z}}_d$ includes only diagonal blocks of the $\overline{\mathbf{Z}}$-matrix, and $\overline{\mathbf{Z}}_{off}$-matrix includes the remaining blocks. We now have

$$\begin{aligned} \left[\overline{\mathbf{Z}}_d + \overline{\mathbf{Z}}_{off}\right]\mathbf{X} &= \overline{\mathbf{Y}} \\ \Rightarrow \overline{\mathbf{Z}}_d\left[\mathbf{I} + \overline{\mathbf{Z}}_d^{-1}\overline{\mathbf{Z}}_{off}\right]\mathbf{X} &= \overline{\mathbf{Y}} \\ \Rightarrow \left[\mathbf{I} + \overline{\mathbf{Z}}_d^{-1}\overline{\mathbf{Z}}_{off}\right]\mathbf{X} &= \overline{\mathbf{Z}}_d^{-1}\overline{\mathbf{Y}} \\ \Rightarrow [\mathbf{I} + \mathbf{U}]\,\mathbf{X} &= \mathbf{X}_0. \end{aligned} \tag{24}$$

In Eq. (24), $\mathbf{X}_0 = \overline{\mathbf{Z}}_d^{-1}\overline{\mathbf{Y}}$ and $\mathbf{U} = \overline{\mathbf{Z}}_d^{-1}\overline{\mathbf{Z}}_{off}$.

Finally, the solution, $\mathbf{X}$, is obtained by expanding Eq. (24) in a power series as,

$$\begin{aligned} \mathbf{X} &= [\mathbf{I} + \mathbf{U}]^{-1}\,\mathbf{X}_0 \\ &= \left[\mathbf{I} - \mathbf{U} + \mathbf{U}^2 - \mathbf{U}^3 + \cdots\right]\mathbf{X}_0 \\ &= \mathbf{X}_0 - \mathbf{U}\mathbf{X}_0 + \mathbf{U}[\mathbf{U}\mathbf{X}_0] \\ &\quad -\mathbf{U}(\mathbf{U}[\mathbf{U}\mathbf{X}_0]) + \cdots . \end{aligned} \tag{25}$$

The necessary and sufficient condition for the power series in Eq. (25) to converge is the Frobenius norm $||\mathbf{U}|| \leq 1$ [12]. To achieve this condition, it may be easier to enforce $||\overline{\mathbf{Z}}_d^{-1}|| \cdot ||\overline{\mathbf{Z}}_{off}|| \leq 1$. The norms of $\overline{\mathbf{Z}}_d^{-1}$ and $\overline{\mathbf{Z}}_{off}$ are computed while generating these terms and ensure that the necessary condition is satisfied. If the condition is not satisfied, the group size must be increased to the necessary value. Alternatively, one may adopt the following procedure.

We note that

$$\begin{aligned} ||\overline{\mathbf{Z}}_d^{-1}|| &= \frac{||\overline{\mathbf{Z}}_d^{-1}|| \cdot ||\overline{\mathbf{Z}}_d\mathbf{X}_0||}{||\overline{\mathbf{Z}}_d\mathbf{X}_0||} \\ &\leq \frac{||\overline{\mathbf{Z}}_d^{-1}|| \cdot ||\overline{\mathbf{Z}}_d|| \cdot ||\mathbf{X}_0||}{||\overline{\mathbf{Y}}||} \\ &= \kappa_d\frac{||\mathbf{X}_0||}{||\overline{\mathbf{Y}}||}, \end{aligned} \tag{26}$$

where $\kappa_d = ||\overline{\mathbf{Z}}_d^{-1}|| \cdot ||\overline{\mathbf{Z}}_d||$ represents the condition number of $\overline{\mathbf{Z}}_d$.

Next, we define $\overline{\mathbf{Y}}_e = \overline{\mathbf{Z}}\mathbf{X}_0 - \overline{\mathbf{Z}}_d$ and $\overline{\mathbf{X}_0} = \mathbf{Z}_{off}\mathbf{X}_0$ and we have

$$\begin{aligned} ||\overline{\mathbf{Z}}_{off}|| &= \frac{||\overline{\mathbf{Z}}_{off}|| \cdot ||\overline{\mathbf{Z}}_{off}^{-1}\overline{\mathbf{Y}}_e||}{||\overline{\mathbf{Z}}_{off}^{-1}\overline{\mathbf{Y}}_e||} \\ &\leq \frac{||\overline{\mathbf{Z}}_{off}|| \cdot ||\overline{\mathbf{Z}}_{off}^{-1}|| \cdot ||\overline{\mathbf{Y}}_e||}{||\mathbf{X}_0||} \\ &= \kappa_{off}\frac{||\overline{\mathbf{Y}}_e||}{||\mathbf{X}_0||}, \end{aligned} \tag{27}$$

where $\kappa_{off} = ||\overline{\mathbf{Z}}_{off}^{-1}|| \cdot ||\overline{\mathbf{Z}}_{off}||$ represents the condition number of $\overline{\mathbf{Z}}_{off}$.

Combining Eqs. (26) and (27), we have

$$||\overline{\mathbf{Z}}_d^{-1}|| \cdot ||\overline{\mathbf{Z}}_{off}|| \leq \kappa_d\kappa_{off}\frac{||\overline{\mathbf{Y}}_e||}{||\overline{\mathbf{Y}}||} \tag{28}$$

and, to satisfy the condition $||\mathbf{U}|| \leq 1$, we must ensure that

$$\frac{||\overline{\mathbf{Y}}_e||}{||\overline{\mathbf{Y}}||} \leq \frac{1}{\kappa_d\kappa_{off}}. \tag{29}$$

We note that computing the condition numbers κ_d and κ_{off} is not needed but must ensure that the fraction $\frac{||\overline{\mathbf{Y}}_e||}{||\overline{\mathbf{Y}}||}$ is a small number. Our various numerical experiments suggest that this number must be less than 0.4. This is because the described numerical implementation ensures that the matrices, $\overline{\mathbf{Z}}_d$ and $\overline{\mathbf{Z}}_{off}$, are well-conditioned matrices. If the $\frac{||\overline{\mathbf{Y}}_e||}{||\overline{\mathbf{Y}}||}$ is not less than 0.4, then the solution may diverge and the procedure needs to be reimplemented by increasing the group size.

Alternatively, we solved Eq. (23) by carrying out the following mathematical and numerical operations:

1. Obtain $\mathbf{X}_0$ as before by solving the equation $\overline{\mathbf{Z}}_d\mathbf{X}_0 = \overline{\mathbf{Y}}$.
2. Obtain $\overline{\mathbf{Y}}_0$ by performing $\overline{\mathbf{Y}}_0 = \overline{\mathbf{Z}}\mathbf{X}_0$. Then, we have $\overline{\mathbf{Z}}\mathbf{X} = \overline{\mathbf{Y}}$ and $\overline{\mathbf{Z}}\mathbf{X}_0 = \overline{\mathbf{Y}}_0$.
3. Thus, we have $\overline{\mathbf{Z}}(\mathbf{X} - \mathbf{X}_0) = (\overline{\mathbf{Y}} - \overline{\mathbf{Y}}_0)$, which is in the same form as Eq. (23). Hence, the process is repeated until we have convergence. Note that, for the iterative process to converge, we must satisfy Eq. (29).

We note that the matrix, $\mathbf{Z}_{off}$, contains the whole matrix except for the diagonal blocks. The development of $\mathbf{Z}_{off}$ could be time consuming when the number of unknowns is very high. A worthwhile improvement to this approach would be to avoid evaluating the whole matrix and, to this end, we developed a new Multi-Level Solution Algorithm as described in the following section.

4.2 Development of Multi-Lev Solutionel Algorithm

As a first step, define the total number of levels, L, where the the number of groups, P, $= 2^L$. We note that the number of groups should be a multiple of 2, which is simple to achieve and does not restrict the algorithm in any sense.

Next, for the sake of clarity and ease of explaining the algorithm development, we select $L = 4$, which implies that we have $2^4 = 16$ groups in the problem. Thus, following the numerical procedures of the previous section, we develop the matrix equation

$$\overline{\mathbf{Z}}^0 \mathbf{X} = \overline{\mathbf{Y}}, \tag{30}$$

where the new $\overline{\mathbf{Z}}^0$-matrix is same as $\overline{\mathbf{Z}}$, and written, for the sake of clarity, as

$$\overline{\mathbf{Z}}^0 = \begin{bmatrix} \tilde{\mathbf{Z}}^0_{1,1} & \mathbf{O} & \mathbf{O} & \cdots & \tilde{\mathbf{Z}}^0_{1,14} & \tilde{\mathbf{Z}}^0_{1,15} & \tilde{\mathbf{Z}}^0_{1,16} \\ \mathbf{O} & \tilde{\mathbf{Z}}^0_{2,2} & \mathbf{O} & \cdots & \tilde{\mathbf{Z}}^0_{2,14} & \tilde{\mathbf{Z}}^0_{2,15} & \tilde{\mathbf{Z}}^0_{2,16} \\ \tilde{\mathbf{Z}}^0_{3,1} & \mathbf{O} & \tilde{\mathbf{Z}}^0_{3,3} & \cdots & \tilde{\mathbf{Z}}^0_{3,14} & \tilde{\mathbf{Z}}^0_{3,15} & \tilde{\mathbf{Z}}^0_{3,16} \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ \tilde{\mathbf{Z}}^0_{16,1} & \tilde{\mathbf{Z}}^0_{16,2} & \tilde{\mathbf{Z}}^0_{16,3} & \cdots & \mathbf{O} & \mathbf{O} & \tilde{\mathbf{Z}}^0_{16,16} \end{bmatrix} \tag{31}$$

The solution of Eq. (30) is obtained by developing the solution at each level as described in the following:

4.2.1 Level-Zero Solution

The Level-Zero solution, $\mathbf{X}^0$, is obtained by solving the following equation,

$$\overline{\mathbf{Z}}^0_d \mathbf{X}^0 = \overline{\mathbf{Y}}, \tag{32}$$

where the new $\overline{\mathbf{Z}}^0_d$-matrix is obtained by retaining only the diagonal blocks and neglecting all other blocks. Thus, we have

$$\overline{\mathbf{Z}}^0_d = \begin{bmatrix} \tilde{\mathbf{Z}}^0_{1,1} & \mathbf{O} & \mathbf{O} & \cdots & \mathbf{O} & \mathbf{O} & \mathbf{O} \\ \mathbf{O} & \tilde{\mathbf{Z}}^0_{2,2} & \mathbf{O} & \cdots & \mathbf{O} & \mathbf{O} & \mathbf{O} \\ \mathbf{O} & \mathbf{O} & \tilde{\mathbf{Z}}^0_{3,3} & \cdots & \mathbf{O} & \mathbf{O} & \mathbf{O} \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ \mathbf{O} & \mathbf{O} & \mathbf{O} & \cdots & \mathbf{O} & \mathbf{O} & \tilde{\mathbf{Z}}^0_{16,16} \end{bmatrix}.$$

We rewrite Eq. (32) as

$$\overline{\mathbf{Z}}^0_{i,i} \mathbf{X}^0_i = \overline{\mathbf{Y}}_i, \tag{33}$$

for $i = 1, 2, \cdots, 16$ because each block is decoupled from the other blocks. The solution of Eq. (33) requires inverting an $M \times M$ matrix, and we have to perform this inversion for each block. This completes the zero-level solution.

Once we obtain the zero-level solution, we proceed to the next level as explained in the following:

4.2.2 Level-One Solution

We proceed to the first-level by increasing the block size to $2M$ and reducing the number blocks to half of the previous level. Thus, for the selected example, we now have an 8×8 matrix with each block of size $2M \times 2M$. The Level-One matrix equation is

$$\overline{\mathbf{Z}}^1 \mathbf{X} = \overline{\mathbf{Y}}, \tag{34}$$

where

$$\overline{\mathbf{Z}}^1 = \begin{bmatrix} \tilde{\mathbf{Z}}^1_{1,1} & \tilde{\mathbf{Z}}^1_{1,2} & \tilde{\mathbf{Z}}^1_{1,3} & \cdots & \tilde{\mathbf{Z}}^1_{1,6} & \tilde{\mathbf{Z}}^1_{1,7} & \tilde{\mathbf{Z}}^1_{1,8} \\ \tilde{\mathbf{Z}}^1_{2,1} & \tilde{\mathbf{Z}}^1_{2,2} & \tilde{\mathbf{Z}}^1_{2,3} & \cdots & \tilde{\mathbf{Z}}^1_{2,6} & \tilde{\mathbf{Z}}^1_{2,7} & \tilde{\mathbf{Z}}^1_{2,8} \\ \tilde{\mathbf{Z}}^1_{3,1} & \tilde{\mathbf{Z}}^1_{3,2} & \tilde{\mathbf{Z}}^1_{3,3} & \cdots & \tilde{\mathbf{Z}}^1_{3,6} & \tilde{\mathbf{Z}}^1_{3,7} & \tilde{\mathbf{Z}}^1_{3,8} \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ \tilde{\mathbf{Z}}^1_{8,1} & \tilde{\mathbf{Z}}^1_{8,2} & \tilde{\mathbf{Z}}^1_{8,3} & \cdots & \tilde{\mathbf{Z}}^1_{8,6} & \tilde{\mathbf{Z}}^1_{8,7} & \tilde{\mathbf{Z}}^1_{8,8} \end{bmatrix} \tag{35}$$

The Level-One solution, $\mathbf{X}^1$, is obtained by solving the following equation,

$$\overline{\mathbf{Z}}^1_d \mathbf{X}^1 = \overline{\mathbf{Y}}, \tag{36}$$

where $\overline{\mathbf{Z}}^1_d$-matrix is obtained by retaining only the diagonal blocks in Eq. (35) and neglecting all other blocks. Thus, we have

$$\overline{\mathbf{Z}}^1_d = \begin{bmatrix} \tilde{\mathbf{Z}}^1_{1,1} & \mathbf{O} & \mathbf{O} & \cdots & \mathbf{O} & \mathbf{O} & \mathbf{O} \\ \mathbf{O} & \tilde{\mathbf{Z}}^1_{2,2} & O & \cdots & \mathbf{O} & \mathbf{O} & \mathbf{O} \\ \mathbf{O} & \mathbf{O} & \tilde{\mathbf{Z}}^1_{3,3} & \cdots & \mathbf{O} & \mathbf{O} & \mathbf{O} \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ \mathbf{O} & \mathbf{O} & \mathbf{O} & \cdots & \mathbf{O} & \mathbf{O} & \tilde{\mathbf{Z}}^1_{8,8} \end{bmatrix} \tag{37}$$

We rewrite Eq. (37) as

$$\overline{\mathbf{Z}}^1_{i,i} \mathbf{X}^1_i = \overline{\mathbf{Y}}_i \tag{38}$$

for $i = 1, 2, \cdots, 8$ because each block is decoupled from the other blocks.

We note that $\mathbf{Z}^1_{i,i}$ in Eq. (38) can be written as

$$\overline{\mathbf{Z}}^1_{i,i} = \begin{bmatrix} \mathbf{A} & \mathbf{O} \\ \mathbf{O} & \mathbf{B} \end{bmatrix} \tag{39}$$

where $\mathbf{A}$ and $\mathbf{B}$ are zero level diagonal matrices. Since the off-diagonal blocks of Eq. (39) are zero, the inverse of $\mathbf{Z}^1_{t,i}$, for $i = 1, 2, \cdots, 8$ is simply the inverse of $\mathbf{A}$ and $\mathbf{B}$. Hence, the Level-One solution is the same as the Level-Zero solution. Now, we consider the second level solution.

4.2.3 Level-Two Solution

Proceeding to the Level-Two solution, we double the block size and reduce the group size to half of the previous level. Thus, for the selected example, we now have a 4×4 matrix with each block of size $4M \times 4M$. The Level-Two matrix equation is

$$\overline{\mathbf{Z}}^2 \mathbf{X} = \overline{\mathbf{Y}}, \tag{40}$$

where

$$\overline{\mathbf{Z}}^2 = \begin{bmatrix} \tilde{\mathbf{Z}}^2_{1,1} & \tilde{\mathbf{Z}}^2_{1,2} & \tilde{\mathbf{Z}}^2_{1,3} & \tilde{\mathbf{Z}}^2_{1,4} \\ \tilde{\mathbf{Z}}^2_{2,1} & \tilde{\mathbf{Z}}^2_{2,2} & \tilde{\mathbf{Z}}^2_{2,3} & \tilde{\mathbf{Z}}^2_{2,4} \\ \tilde{\mathbf{Z}}^2_{3,1} & \tilde{\mathbf{Z}}^2_{3,2} & \tilde{\mathbf{Z}}^2_{3,3} & \tilde{\mathbf{Z}}^2_{3,4} \\ \tilde{\mathbf{Z}}^2_{4,1} & \tilde{\mathbf{Z}}^2_{4,2} & \tilde{\mathbf{Z}}^2_{4,3} & \tilde{\mathbf{Z}}^2_{4,4} \end{bmatrix} \tag{41}$$

The Level-Two solution, $\mathbf{X}^2$, is obtained by solving the following equation,

$$\overline{\mathbf{Z}}^2_d \mathbf{X}^2 = \overline{\mathbf{Y}}, \tag{42}$$

where the $\overline{\mathbf{Z}}^2_d$-matrix is obtained by retaining only the diagonal blocks in Eq. (41) and neglecting all other blocks. Thus, we have

$$\overline{\mathbf{Z}}^2_d = \begin{bmatrix} \tilde{\mathbf{Z}}^2_{1,1} & \mathbf{O} & \mathbf{O} & \mathbf{O} \\ \mathbf{O} & \tilde{\mathbf{Z}}^2_{2,2} & O & \mathbf{O} \\ \mathbf{O} & \mathbf{O} & \tilde{\mathbf{Z}}^2_{3,3} & \mathbf{O} \\ \mathbf{O} & \mathbf{O} & \mathbf{O} & \tilde{\mathbf{Z}}^2_{4,4} \end{bmatrix} \tag{43}$$

We rewrite Eq. (43) as

$$\overline{\mathbf{Z}}^2_{i,i} \mathbf{X}^2_i = \overline{\mathbf{Y}}_i, \tag{44}$$

for $i = 1, 2, 3, 4$ because each block is decoupled from the other blocks.

Next, we note that we can write $\mathbf{Z}^2_{i,i}$ in Eq. (44) as

$$\overline{\mathbf{Z}}^2_{i,i} = \begin{bmatrix} \mathbf{A} & \mathbf{C} \\ \mathbf{D} & \mathbf{B} \end{bmatrix} \tag{45}$$

and to obtain the Level-Two solution, we must solve an equation

$$\begin{bmatrix} \mathbf{A} & \mathbf{C} \\ \mathbf{D} & \mathbf{B} \end{bmatrix} \begin{bmatrix} \mathbf{X}_1 \\ \mathbf{X}_2 \end{bmatrix} = \begin{bmatrix} \mathbf{Y}_1 \\ \mathbf{Y}_2 \end{bmatrix} \tag{46}$$

for each diagonal block $i = 1, 2, 3, 4$. Note that the matrices $\mathbf{A}$ and $\mathbf{B}$ are actually the diagonal matrices of the previous level. The solution of Eq. (46) is obtained using the power-series solution method described in the previous section. We also note that, in order to obtain the power series solution, we use the previous level solution as the diagonal solution as required in Eq. (24).

4.2.4 Level-Three Solution

Proceeding to the Level-Three solution, we double the block size and reduce the group size to half of the previous level. Thus, for the selected example, we now have a 2×2 matrix with each block of size $8M \times 8M$. The Level-Three matrix equation is

$$\overline{\mathbf{Z}}^3 \mathbf{X} = \overline{\mathbf{Y}}, \tag{47}$$

where

$$\overline{\mathbf{Z}}^3 = \begin{bmatrix} \tilde{\mathbf{Z}}^3_{1,1} & \tilde{\mathbf{Z}}^3_{1,2} \\ \tilde{\mathbf{Z}}^3_{2,1} & \tilde{\mathbf{Z}}^3_{2,2} \end{bmatrix}. \tag{48}$$

The Level-Three solution, $\mathbf{X}^3$, is obtained by solving the following equation,

$$\overline{\mathbf{Z}}^3_d \mathbf{X}^3 = \overline{\mathbf{Y}}, \tag{49}$$

where the $\overline{\mathbf{Z}}^3_d$-matrix is obtained by retaining only the diagonal blocks in Eq. (48) and neglecting the off-diagonal blocks. Thus, we have

$$\overline{\mathbf{Z}}^3_d = \begin{bmatrix} \tilde{\mathbf{Z}}^3_{1,1} & \mathbf{O} \\ \mathbf{O} & \tilde{\mathbf{Z}}^3_{2,2} \end{bmatrix} \tag{50}$$

We rewrite Eq. (50) as

$$\overline{\mathbf{Z}^3_{i,i}} \mathbf{X}^3_i = \overline{\mathbf{Y}_i}, \tag{51}$$

for $i = 1$ and $i = 2$ because each block is decoupled from the other blocks.

Next, we note that we can write $\mathbf{Z}^3_{i,i}$ in Eq. (51) as

$$\overline{\mathbf{Z}}^3_{i,i} = \begin{bmatrix} \mathbf{A} & \mathbf{C} \\ \mathbf{D} & \mathbf{B} \end{bmatrix} \tag{52}$$

and to obtain the Level-Three solution, we must solve an equation

$$\begin{bmatrix} \mathbf{A} & \mathbf{C} \\ \mathbf{D} & \mathbf{B} \end{bmatrix} \begin{bmatrix} \mathbf{X}_1 \\ \mathbf{X}_2 \end{bmatrix} = \begin{bmatrix} \mathbf{Y}_1 \\ \mathbf{Y}_2 \end{bmatrix} \tag{53}$$

for diagonal blocks $i = 1$ and $i = 2$. Again, we note that the matrices $\mathbf{A}$ and $\mathbf{B}$ are the diagonal blocks of the previous level. The solution of Eq. (53) is obtained using the power-series solution method described in the previous section. We also note that, in order to obtain the power series solution, we can use the previous level solution as the diagonal solution as required in Eq. (24).

4.2.5 Level-Four Solution

Finally, we proceed to the last level, *i.e*. Level-Four. Once again, we double the block size and reduce the group size to half of the previous level. Thus, for the selected example, we now have the whole matrix with a single block of size $16M \times 16M$. The Level-Four matrix equation is

$$\overline{\mathbf{Z}}^4\mathbf{X} = \overline{\mathbf{Y}}, \tag{54}$$

where

$$\overline{\mathbf{Z}}^4_{i,i} = \begin{bmatrix} \mathbf{A} & \mathbf{C} \\ \mathbf{D} & \mathbf{B} \end{bmatrix}. \tag{55}$$

The solution of Eq. (54) is obtained using the power-series solution method described in the previous section. We also note that, in order to obtain the power series solution, we can use the previous level solution as the diagonal solution as required in Eq. (24). Thus, we complete the Multi-Level Solution Algorithm.

4.3 An Alternate Procedure to Obtain the Multi-Level Solution

By examining the important steps of the previous section, we note that the total solution is accurately obtained if we solve the problem described in the following:

4.3.1 Problem Statement

Let us consider the solution of a 2X2 matrix equation that is useful for our Multi-Level Solution Algorithm. The matrix equation is

$$\mathbf{ZX} = \mathbf{Y}, \tag{56}$$

where

$$\mathbf{Z} = \begin{bmatrix} \mathbf{A} & \mathbf{C} \\ \mathbf{D} & \mathbf{B} \end{bmatrix}, \tag{57}$$

$$\mathbf{X} = \begin{bmatrix} \mathbf{X}_1 \\ \mathbf{X}_2 \end{bmatrix}, \tag{58}$$

$$\mathbf{Y} = \begin{bmatrix} \mathbf{Y}_1 \\ \mathbf{Y}_2 \end{bmatrix}, \tag{59}$$

where **A**, **B**, **C**, and **D** are matrices of dimension, N. Given **Y**, we need to find **X**. The solution to the problem is obtained by solving the matrix equation. We also have

$$\mathbf{X}_p = \mathbf{A}^{-1}\mathbf{Y}_1 \quad \text{and} \quad \mathbf{X}_q = \mathbf{B}^{-1}\mathbf{Y}_2 \tag{60}$$

We want to get an expression for the solution, **X,** in terms of **A**, **B**, **C**, **D**, $\mathbf{X}_p$, and $\mathbf{X}_q$.

4.3.2 Solution Procedure

We write the inverse of the **Z**-matrix in Eq. (56) as

$$\mathbf{Z}^{-1} = \begin{bmatrix} \mathbf{Q}^{-1}\mathbf{A}^{-1} & -\mathbf{Q}^{-1}\mathbf{A}^{-1}\mathbf{C}\mathbf{B}^{-1} \\ -\mathbf{S}^{-1}\mathbf{B}^{-1}\mathbf{D}\mathbf{A}^{-1} & \mathbf{S}^{-1}\mathbf{B}^{-1} \end{bmatrix}, \tag{61}$$

where

$$\mathbf{Q} = \mathbf{I} - \mathbf{A}^{-1}\mathbf{C}\mathbf{B}^{-1}\mathbf{D} \tag{62}$$

$$\mathbf{S} = \mathbf{I} - \mathbf{B}^{-1}\mathbf{D}\mathbf{A}^{-1}\mathbf{C} \tag{63}$$

Finally, the solution is written as

$$\begin{aligned} \mathbf{X} &= \mathbf{Z}^{-1}\mathbf{Y} \\ &= \begin{bmatrix} \mathbf{Q}^{-1} & \mathbf{O} \\ \mathbf{O} & \mathbf{S}^{-1} \end{bmatrix} \begin{bmatrix} \mathbf{X}_p - \mathbf{A}^{-1}\mathbf{C}\mathbf{X}_q \\ \mathbf{X}_q - \mathbf{B}^{-1}\mathbf{D}\mathbf{X}_p \end{bmatrix} \end{aligned} \tag{64}$$

Now, approximating $\mathbf{Q}^{-1}$ and $\mathbf{S}^{-1}$ as

$$\mathbf{Q}^{-1} = \left[\mathbf{I} - \mathbf{A}^{-1}\mathbf{C}\mathbf{B}^{-1}\mathbf{D}\right]^{-1} \approx \mathbf{I} + \mathbf{A}^{-1}\mathbf{C}\mathbf{B}^{-1}\mathbf{D} \tag{65}$$

$$\mathbf{S}^{-1} = \left[\mathbf{I} - \mathbf{B}^{-1}\mathbf{D}\mathbf{A}^{-1}\mathbf{C}\right]^{-1} \approx \mathbf{I} + \mathbf{B}^{-1}\mathbf{D}\mathbf{A}^{-1}\mathbf{C} \tag{66}$$

we have

$$\mathbf{X}_1 = \left[\mathbf{I} + \mathbf{A}^{-1}\mathbf{C}\mathbf{B}^{-1}\mathbf{D}\right]\left[\mathbf{X}_p - \mathbf{A}^{-1}\mathbf{C}\mathbf{X}_q\right] \tag{67}$$

$$\mathbf{X}_2 = \left[\mathbf{I} + \mathbf{B}^{-1}\mathbf{D}\mathbf{A}^{-1}\mathbf{C}\ \right]\left]\mathbf{X}_q - \mathbf{B}^{-1}\mathbf{D}\mathbf{X}_p\right] \tag{68}$$

Using expressions (65) and (66) appropriately, we obtain the total solution.

4.4 Numerical Results

In this section, we present a few representative numerical results involving three-dimensional, perfectly electric conducting bodies for validation purposes. We present the bistatic radar cross section (RCS) calculations of several objects, and compare these with the standard MoM solution or with an exact solution where available. For all the examples presented in this section, the body is placed such that the body center approximately coincides with the center of the coordinate system. Also, for all examples, the incident wave is a 300 MHz plane wave polarized along the X-axis and traveling along the negative Z-direction.

As a first example, consider a conducting sphere, radius= 3.0m, placed with center coinciding with the center of the coordinate system. The sphere is approximated by 29,400 basis functions. The Multi-Level Solution Algorithm adopts 6 levels with a group size at the lowest level equal to 460. The bistatic RCS for this problem is compared with the exact solution obtained by the Mie-Series solution [13] and presented in Fig. 6. We note excellent agreement between the two results.

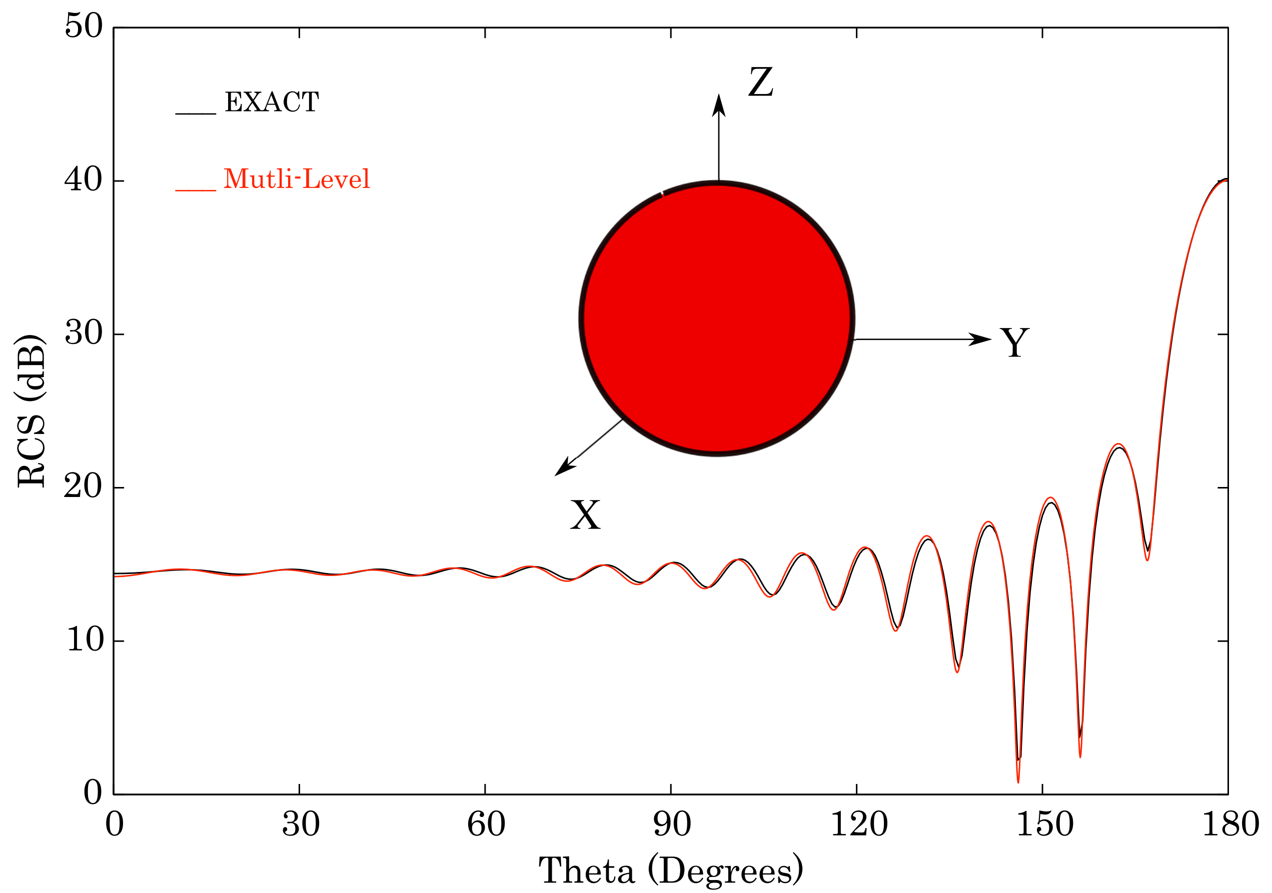

Fig. 6—RCS of a PEC sphere, radius= 3m, and illuminated by an X- polarized, Z-traveling 300 MHz plane wave. Number of levels =6, N=29,400.

Now, let us consider a long, PEC cylinder closed at both ends with L = 50m and radius a = 0.1m. The cylinder is placed along the Z-axis and illuminated by a plane wave as described earlier. Because the cylinder is thin and long, the RCS varies considerably along the length of the cylinder depicting several maxima and minima. Our intention with this example is to demonstrate that our new, Multi-Level Solution Algorithm captures this effect accurately. For the MOM solution, the cylinder is approximated by 18,018 basis functions. For the Multi-Level Solution Algorithm, the total number of basis functions are divided into 32 groups with each group consisting of 564 basis functions. In Fig. 7, we plot the bistatic RCS for this case and compare it with the conventional MOM solution. We note an excellent comparison of the Multi-Level Solution Algorithm with the MOM solution.

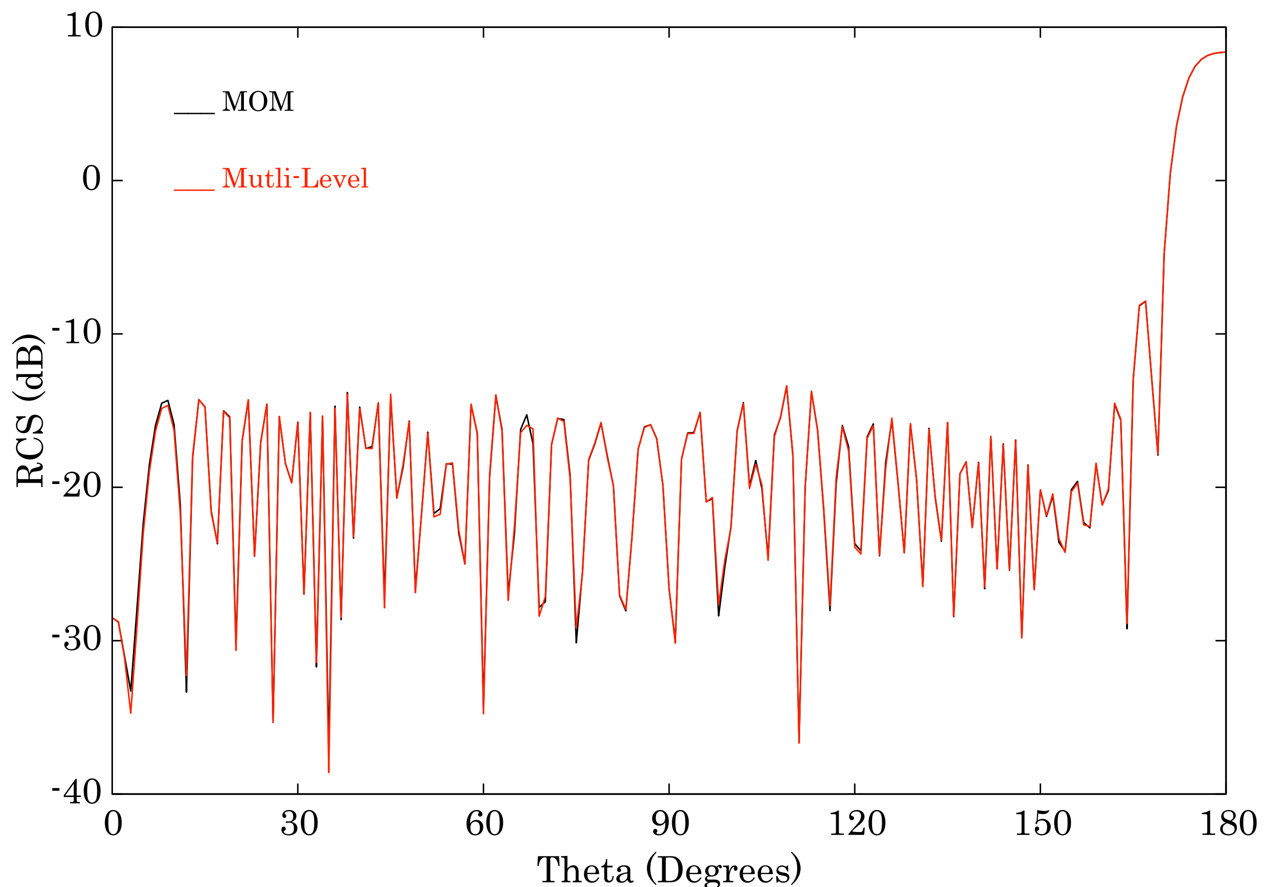

Fig. 7—RCS of a three-dimensional PEC cylinder, L = 50m and radius a = 0.1m, placed along the Z-axis and illuminated by an X-polarized, Z-traveling 300 MHz plane wave. Number of levels =5, N=18,018.

Next, we consider a 100m long cylinder closed at both ends with radius a = 0.1m. The cylinder is placed along the Z-axis and illuminated by a plane wave as described earlier. The cylinder is approximated by 36,018 basis functions. For the Multi-Level Solution Algorithm, the total number of basis functions are divided into 64 groups with each group consisting of 564 basis functions. In Fig. 8 we plot the bistatic RCS for this case and compare it with the conventional MOM solution. Once again, we note an excellent comparison between the two solutions.

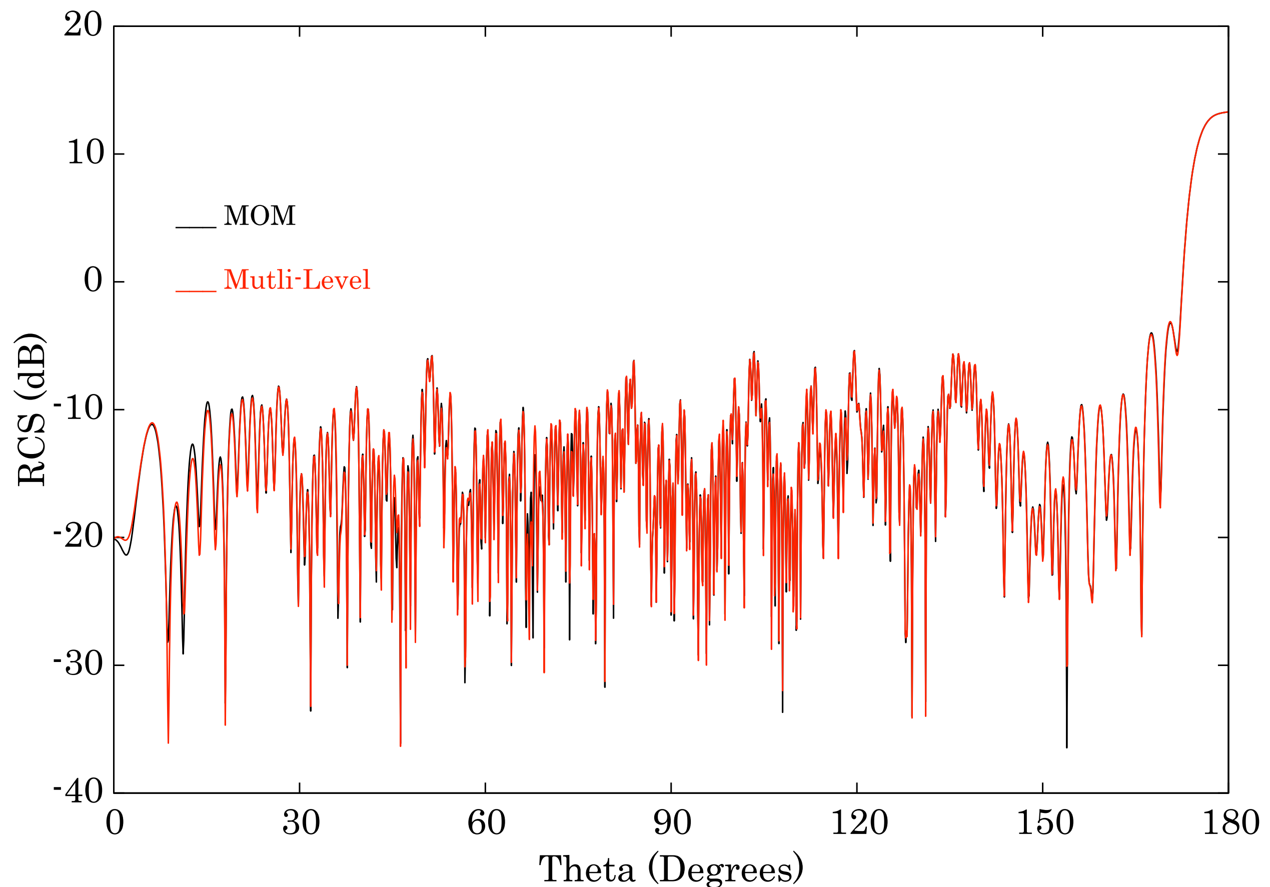

Fig. 8—RCS of a three-dimensional PEC cylinder, L = 100m and radius a = 0.1m, placed along the Z-axis and illuminated by an X-polarized, Z-traveling 300 MHz plane wave. Number of levels =6, N=36,018.

Now, we consider a more complex, non-canonical object to illustrate the applicability of the Multi-Level Solution Algorithm for complex problems. Consider an aircraft model illuminated by an X-polarized plane wave traveling along the Z-axis. The model is located in the XY-plane with nose along the X-axis. The largest dimensions of the model along the X, Y, and Z axes are 5.85m, 3.5m, and 1.76m, respectively. The model is approximated by 10,692 basis functions shown as facets in Fig. 9. For the Multi-Level Solution Algorithm, the total number of basis functions is divided into 32 groups (5 levels), with 335 basis functions in each group. The bistatic RCS along the XZ-plane is shown in Fig. 10. The Multi-Level Solution Algorithm result is compared with the MOM solution, and we note a reasonably good comparison for this complex structure.

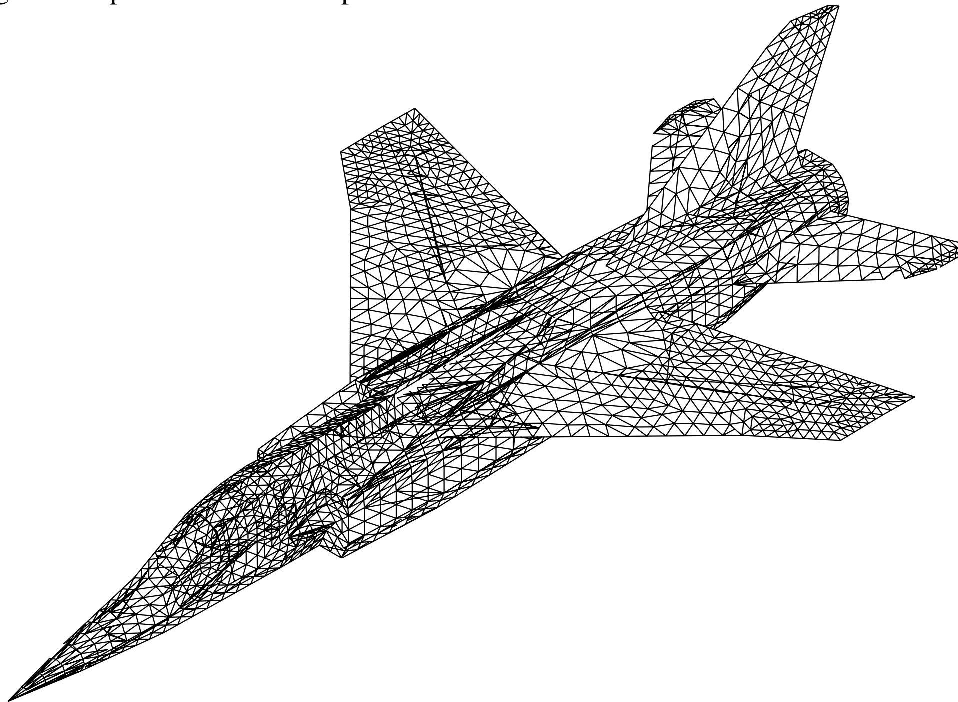

Fig. 9—An aircraft-like model. The dimensions of the model along the X, Y, and Z axes are 5.85m, 3.5m, and 1.76m, respectively.

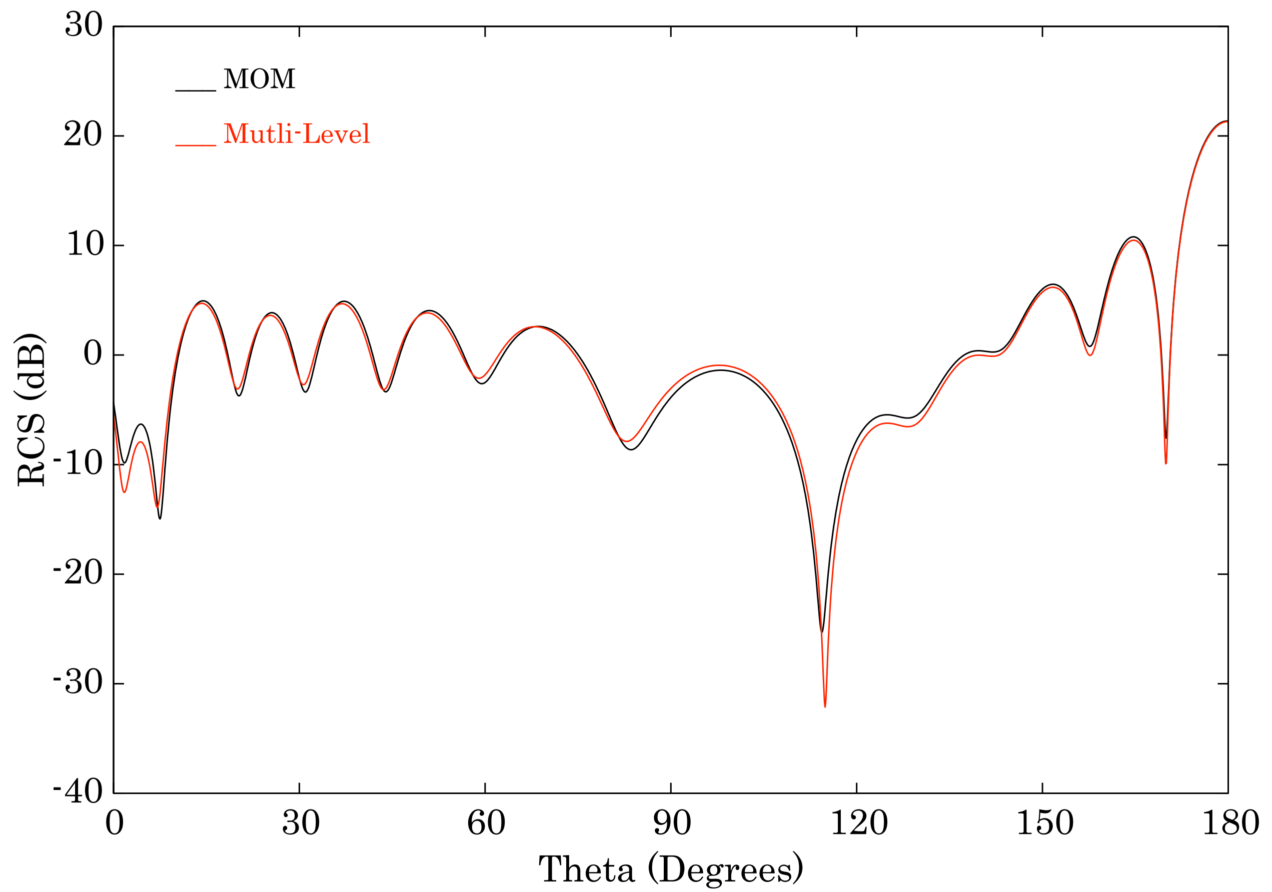

Fig. 10—RCS of an aircraft-like model. The dimensions of the model along the X, Y, and Z axes are 5.85m, 3.5m, and 1.76m, respectively. The model is placed such that the geometrical center roughly coincides with the center of the coordinate system. The model is illuminated by an X-polarized, Z-traveling 300 MHz plane wave. Number of levels =5, N=10,692.

As a final example, consider the 30-meter, ship-like model as shown in Fig. 1 1 . The model is illuminated by a 300 MHz, X-polarized plane wave traveling along the Z-axis. The model is approximated by 54,408 basis functions. For the Multi-Level Solution Algorithm, the total number of basis functions is divided into 128 groups (7 levels), with 426 basis functions in each group. The bistatic RCS along the YZ-plane is shown in Fig. 11 and compared with the conventional MOM solution. Once again, we note excellent agreement between the two methods.

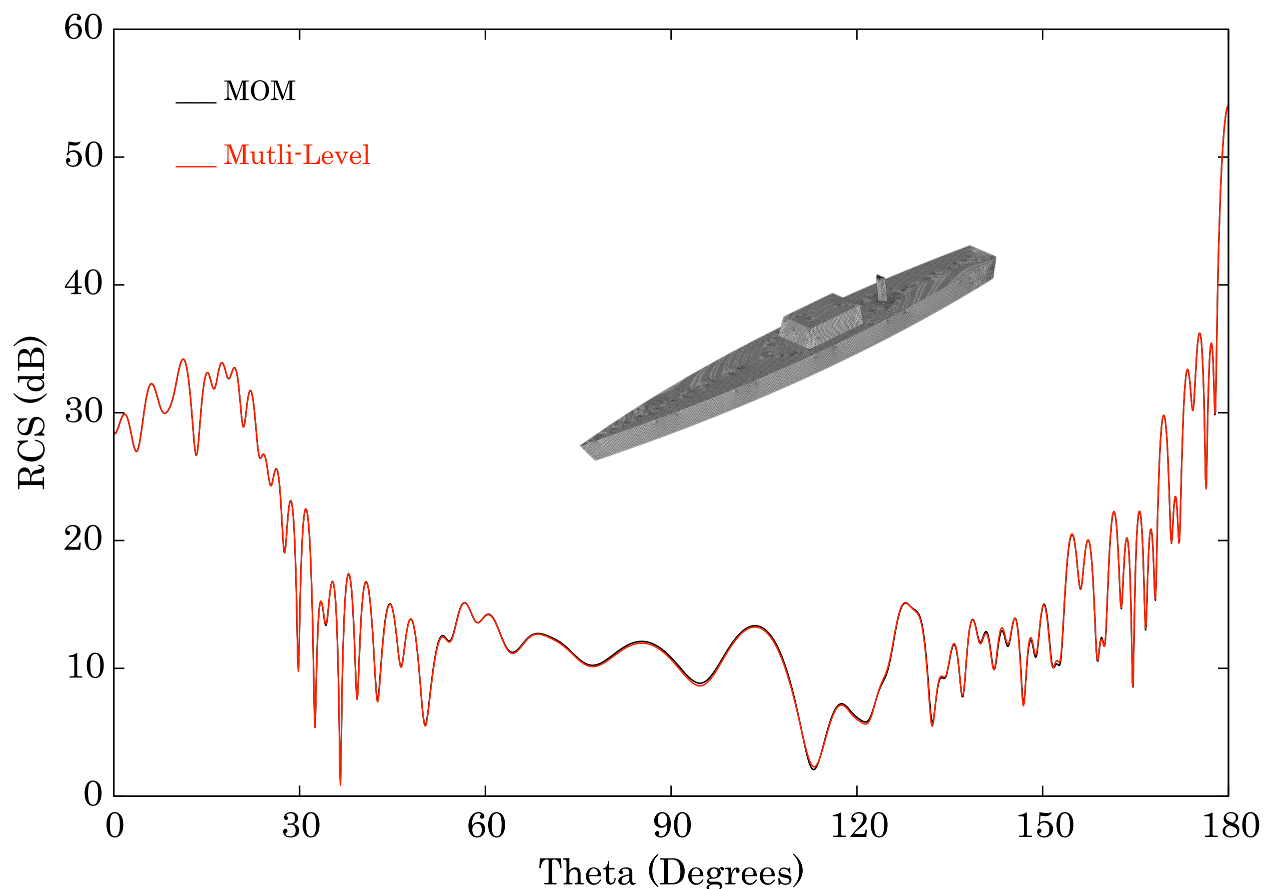

Fig. 11—RCS of a ship-like model. The dimensions of the model along the X, Y, and Z axes are 30m, 2.6m, and 4.0m, respectively. The model is placed such that the geometrical center roughly coincides with the center of the coordinate system. The model is illuminated by an X-polarized, Z-traveling 300 MHz plane wave. Number of levels =7, N=54,408.

5. CONCLUSIONS

In this work, we present two distinct algorithms to efficiently solve matrix equations, and apply the solution procedures to electromagnetic scattering and radiation problems. In the first part, we attempted to develop an efficient solution by parallelizing the widely popular LU-Decomposition method taking into consideration the availability of the parallel processor systems. We have only started this work recently and reported a few preliminary results. Much more work needs to be done in this area and we intend to continue this work in the near future.

In the second part, we present a new and efficient Multi-Level Solution Algorithm to solve electrically large electromagnetic scattering problems utilizing the MOM formulation. The Multi-Level Solution Algorithm retains all of the conventional advantages of MOM, *viz.* accuracy of solution, ability to handle multiple excitation vectors, applicable to scattering and radiation problems, and ability to calculate both near and far-field patterns while handling electrically large PEC bodies. Also, since the method is a purely algebraic method, we expect that the method may be applied to any MOM problem with minor modifications.

The algorithms presented could be more efficient using GPU's, more efficient parallel processing, and dedicated computer hardware implementation via new processors and field programmable gate arrays (FPGAs). However, we have not made such improvements yet though we may report on this in the near future.

Appendix A
Source Code for Parallel Processing

We list the source code for the parallel programming algorithm discussed in Chapter 3. The source code is provided for guidance only. It is expressly stated that although sufficient care was taken to remove programming errors, it is possible that errors remain, and the user is advised to check for errors while coding. Finally, the program may perform less efficiently on non-WINDOWS platforms.

```
!  FortranLUPBlockSolvers.f90
!
!  FUNCTIONS:
!  FortranLUPBlockSolvers - Entry point of console application.
!

!*********************************************************************
****
!
!  PROGRAM: FortranLUPBlockSolvers
!
!  PURPOSE:  Used for testing and developing the LU Decomposition code.
Select which test below to run. Then go down to that subroutine and edit
which matrix test to use.
!               (It should be noted that this file is relatively messy
and was used primarily for testing purposes. However, the
LUPBlocksMPIComplex and LUPOptimized have
!               a clean version of the functions without timing
functionality.) Each processor calls the generic function -> gets sorted
into a manager or worker role ->
!               runs the function. If you run this application with "-n
1", it will run like a normal serial application. Further breakdown of
the LU files is in LUPOptimized.f90
!               Various different matrix tests are stored in
matrixTests.f90 for convenient use. (Most use the matrix that results in
all 1's for post LU algorithm testing using
!               the matrixSolvers.f90 codes)
!*********************************************************************
****

program FortranLUPBlockSolvers

    use mpi

    use matrixTests
    use matrixHelpers
    use matrixUtil
    use matrixSolvers
    use LUOptimized

    implicit none

    !call serialLUPTest()
    ! call MPILUComplexTest()
    call OptimizedLUTest()

end program FortranLUPBlockSolvers


subroutine serialLUPTest() ! Slow, uses P pivot vector

    use matrixTests
    use matrixHelpers
    use LUPBlocksV3
```

```
    implicit none


    ! Variables
    complex, allocatable :: A(:, :), L(:, :), U(:, :)
    integer, allocatable :: P(:)
    integer :: N, pivotRow

    ! Variables Setup
    call fillMatrixTest9_25000x25000Complex(A, L, P, N)

    ! Body of FortranLUPBlockSolvers
    print *, 'Running...'
    call LUPSolver(L, A, P, N)

    deallocate(A)
    deallocate(L)
    deallocate(P)
    write(*,*) "Press enter key to exit.."
    read(*,*)

end subroutine

subroutine MPILUTest()

    use mpi
    use matrixTests
    use matrixHelpers
    use LUPBlocksMPIV1

    ! Variables
    real, allocatable :: A(:, :), L(:, :)
    integer, allocatable :: P(:)
    integer :: N, pivotRow, sizeOfCluster, processRank, maxIndex, ierror,
primary

    call MPI_INIT(ierror)

    call MPI_COMM_RANK(MPI_COMM_WORLD, processRank, ierror)
    call MPI_COMM_SIZE(MPI_COMM_WORLD, sizeOfCluster, ierror)

    N = 0
    primary = 1


    ! Set up matrices
    if(processRank .eq. 0) then
        call fillMatrixTest2_10x10(A, L, P, N)

    end if

    call matRealPrint(A)
```

```
    ! Run LUP
    call LUMPI(A, L, N)


    ! Read output wait
    if(processRank .eq. 0) then
        print*
        print*, "Post A"
        call matRealPrint(A)
        print*, "Post L"
        call matRealPrint(L)
        write(*,*) "Press enter key to exit.."
        read(*,*)
    endif


    ! Clean up
    if(processRank .eq. 0) then
        deallocate(A)
        deallocate(L)
        deallocate(P)
    endif

    call MPI_FINALIZE(ierror)

    end subroutine


subroutine MPILUComplexTest()

    use mpi

    use matrixTests
    use matrixHelpers
    use LUBlocksMPIComplexV1
    use matrixSolvers

    implicit none

    ! Variables
    complex, allocatable :: A(:, :), L(:, :)
    integer, allocatable :: P(:)
    complex, allocatable :: vec(:), sol(:)
    integer :: outcome
    integer :: N, pivotRow, sizeOfCluster, processRank, maxIndex, ierror,
primary

    call MPI_INIT(ierror)

    call MPI_COMM_RANK(MPI_COMM_WORLD, processRank, ierror)
    call MPI_COMM_SIZE(MPI_COMM_WORLD, sizeOfCluster, ierror)

    N = 0
    primary = 1
```

```
! Set up matrices
if(processRank .eq. 0) then
    !call fillMatrixTest6_100x100Complex(A, L, P, N)
    !call fillMatrixTest7_1000x1000Complex(A, L, P, N)
    !call fillMatrixTest8_10000x10000Complex(A, L, P, N)
    call fillMatrixTest9_25000x25000Complex(A, L, P, N)
    !call matComplexPrint(A)
end if

! Run LUP
call LUMPIComplexTiming(A, L, N)

! Check if resulting vector is all 1's
if(processRank .eq. 0) then
    allocate(vec(N))
    allocate(sol(N))

    call zVectorComplex(vec, N)
    sol(:) = 0

    call lowerTriangularMatVecSolve(L, vec, sol, N)
    call upperTriangularMatVecSolve(A, sol, vec, N)

    print*,"SOLVED VEC:"
    !call vecComplexPrint(vec)
    print*, "IS ALL 1's?"
    call vecConstantCheck(vec, (1, 0), 0.05, N, outcome)
    print*, outcome
endif


print*, processRank, "REACHED FINALIZE"
call MPI_FINALIZE(ierror)
print*, "Hit FINALIZE:", processRank

! Read output wait
if(processRank .eq. 0) then
    print*
    write(*,*) "Press enter key to exit.."
    read(*,*)
endif


! Clean up
if(processRank .eq. 0) then
    deallocate(A)
    deallocate(L)
    deallocate(P)
endif

endsubroutine
```

```
subroutine OptimizedLUTest()

    use mpi

    use matrixTests
    use matrixHelpers
    use LUOptimized
    use matrixSolvers

    implicit none

    ! Variables
    complex, allocatable :: A(:, :), L(:, :)
    integer, allocatable :: P(:)
    complex, allocatable :: vec(:), sol(:)
    integer :: outcome
    integer :: N, pivotRow, sizeOfCluster, processRank, maxIndex, ierror,
primary

    call MPI_INIT(ierror)

    call MPI_COMM_RANK(MPI_COMM_WORLD, processRank, ierror)
    call MPI_COMM_SIZE(MPI_COMM_WORLD, sizeOfCluster, ierror)

    N = 0
    primary = 1


    ! Set up matrices
    if(processRank .eq. 0) then
        !call fillMatrixTest6_100x100Complex(A, L, P, N)
        !call fillMatrixTest7_1000x1000Complex(A, L, P, N)
        !call fillMatrixTest8_10000x10000Complex(A, L, P, N)
        !call fillMatrixTest9_25000x25000Complex(A, L, P, N)
        call fillMatrixTest10_50000x50000Complex(A, L, P, N)
        !call matComplexPrint(A)
    end if

    ! Run LU
    call LUOptimizationTiming(A, L, N)

    ! Check if resulting vector is all 1's
    if(processRank .eq. 0) then
        allocate(vec(N))
        allocate(sol(N))

        call zVectorComplex(vec, N)
        sol(:) = 0

        call lowerTriangularMatVecSolve(L, vec, sol, N)
        call upperTriangularMatVecSolve(A, sol, vec, N)

        print*,"SOLVED VEC:"
        !call vecComplexPrint(vec)
```

```
        print*, "IS ALL 1's?"
        call vecConstantCheck(vec, (1, 0), 0.05, N, outcome)
        print*, outcome
    endif


    print*, processRank, "REACHED FINALIZE"
    call MPI_FINALIZE(ierror)
    print*, "Hit FINALIZE:", processRank

    ! Read output wait
    if(processRank .eq. 0) then
        print*
        write(*,*) "Press enter key to exit.."
        read(*,*)
    endif


    ! Clean up
    if(processRank .eq. 0) then
        deallocate(A)
        deallocate(L)
        deallocate(P)
    endif

endsubroutine
```

```
module matrixHelpers

use matrixPrint
use matrixIsAllocated
use matrixMakers

end module
```

```
module matrixIsAllocated

contains
subroutine IsAllocated(mat)
    real, allocatable, intent(in) :: mat(:, :)

    if(allocated(mat)) then
        print *, "Matrix is allocated"
    end if

end subroutine

end module
```

```
module matrixPrint
implicit none

contains

subroutine arrayIntegerPrint(mat)
    integer, allocatable, intent(in) :: mat(:)
    integer :: i
    integer :: low, up

    if(allocated(mat)) then
        low = lbound(mat, 1)
        up = ubound(mat, 1)

        do i = low, up
            write(*, fmt="(i0,a2)", advance="no") mat(i), ", "

        end do
        print *
    endif
end subroutine

subroutine matIntegerPrint(mat)
    integer, allocatable, intent(in) :: mat(:, :)
    integer :: i, j
    integer :: low, up

    if(allocated(mat)) then
        low = lbound(mat, 2)
        up = ubound(mat, 2)

        do i = low, up
            do j = low, up
                write(*, fmt="(i0,a2)", advance="no") mat(i, j), ", "
            end do
            print *
        end do
    endif
end subroutine

subroutine matRealPrint(mat)
    real, allocatable, intent(in) :: mat(:, :)
    integer :: i, j
    integer :: low, up

    if(allocated(mat)) then
        low = lbound(mat, 2)
        up = ubound(mat, 2)

        do i = low, up
            do j = low, up
```

```
                    write(*, fmt="(f0.3,a2)", advance="no") mat(i, j), ",
"
                end do
                print *
            end do
        endif
    end subroutine

    subroutine matComplexPrint(mat)
        complex, allocatable, intent(in) :: mat(:, :)
        integer :: i, j
        integer :: low, up

        if(allocated(mat)) then
            low = lbound(mat, 2)
            up = ubound(mat, 2)

            do i = low, up
                do j = low, up
                    if(imag(mat(i, j)) < 0) then
                        write(*, fmt="(f0.3f0.3a3)", advance="no"),
real(mat(i, j)), imag(mat(i, j)), "j, "
                    else
                        write(*, fmt="(f0.3a1f0.3a3)", advance="no"),
real(mat(i, j)), "+", imag(mat(i, j)), "j, "
                    endif
                end do
                print *
            end do
        endif
    end subroutine

    subroutine vecComplexPrint(vec)
        complex, allocatable, intent(in) :: vec(:)
        integer :: i, j
        integer :: low, up

        if(allocated(vec)) then
            low = lbound(vec, 1)
            up = ubound(vec, 1)

            do i = low, up
                if(imag(vec(i)) < 0) then
                    write(*, fmt="(f0.3f0.3a3)", advance="no"),
real(vec(i)), imag(vec(i)), "j, "
                else
                    write(*, fmt="(f0.3a1f0.3a3)", advance="no"),
real(vec(i)), "+", imag(vec(i)), "j, "
                endif
            end do
        endif
    end subroutine
```

```
    subroutine matRealPrintPivot(mat, pivot)
        real, allocatable, intent(in) :: mat(:, :)
        integer, allocatable, intent(in) :: pivot(:)
        integer :: i, j
        integer :: low, up

        if(allocated(mat) .and. allocated(pivot)) then
            low = lbound(mat, 2)
            up = ubound(mat, 2)

            do i = low, up
                do j = low, up
                    write(*, fmt="(f0.3,a2)", advance="no") mat(pivot(i),
j), ", "
                end do
                print *
            end do
        endif
    end subroutine

    end module
```

```
module matrixSolvers

    implicit none

    contains

    subroutine matrixVectorMulComplex(mat, vec, sol, N)

        implicit none

        complex, allocatable, intent(inout) :: mat(:, :), vec(:)
        complex, allocatable, intent(inout) :: sol(:)
        integer, intent(in) :: N
        integer :: i, j

        do i = 1, N
            do j = 1, N
                sol(j) = sol(j) + vec(i) * mat(j, i)

            enddo
        enddo

    end subroutine

    subroutine lowerTriangularMatVecSolve(mat, vec, sol, N)
        ! mat * sol = vec, solves for sol
        complex, allocatable, intent(in) :: mat(:, :), vec(:)
        complex, allocatable, intent(inout) :: sol(:)
        integer, intent(in) :: N
        integer :: row, col

        do row = 1, N

            sol(row) = vec(row)
            ! Move already known values
            do col = 1, row - 1

                sol(row) = sol(row) - sol(col) * mat(row, col)

            enddo

            ! Divide out the matrix val to get sol val
            sol(row) = sol(row) / mat(row, row)

        enddo

    endsubroutine

    subroutine upperTriangularMatVecSolve(mat, vec, sol, N)

        complex, allocatable, intent(in) :: mat(:, :), vec(:)
        complex, allocatable, intent(inout) :: sol(:)
        integer, intent(in) :: N
```

```
        integer :: row, col, rowd, cold

        do rowd = 1, N

            row = N - rowd + 1

            sol(row) = vec(row)

            ! Subtract the already known values
            do col = row + 1, N

                sol(row) = sol(row) - sol(col) * mat(row, col)

            enddo

            ! Divide out to get the sol value
            sol(row) = sol(row) / mat(row, row)

        enddo


    endsubroutine

endmodule
```

```
module matrixTest1

use matrixMakers

contains
subroutine fillMatrixTest1_3x3(A, L, U, P, N)
    real, allocatable, intent(inout) :: A(:, :), L(:, :), U(:, :)
    integer, allocatable, intent(inout) :: P(:)
    integer, intent(out) :: N

    if(.not. allocated(A)) then
        allocate(A(3, 3))
    endif

    if(.not. allocated(L)) then
        allocate(L(3, 3))
    endif

    if(.not. allocated(U)) then
        allocate(U(3, 3))
    endif

    if(.not. allocated(P)) then
        allocate(P(3))
    endif

    ! Set up L, U, P
    N = 3
    call countMatrix(P, N)
    call zeroMatrix(L, N, N)
    call zeroMatrix(U, N, N)


    ! Set up A
    A(1, 1) = 3
    A(2, 1) = -6
    A(3, 1) = 0
    A(1, 2) = 1
    A(2, 2) = 0
    A(3, 2) = 8
    A(1, 3) = 6
    A(2, 3) = -16
    A(3, 3) = -17

    P = (/1, 2, 3/)

end subroutine


end module
```

```
module matrixTests

    use matrixMakers

    contains
        subroutine fillMatrixTest1_3x3(A, L, P, N)
            real, allocatable, intent(inout) :: A(:, :), L(:, :)
            integer, allocatable, intent(inout) :: P(:)
            integer, intent(out) :: N

            if(.not. allocated(A)) then
                allocate(A(3, 3))
            endif

            if(.not. allocated(L)) then
                allocate(L(3, 3))
            endif

            if(.not. allocated(P)) then
                allocate(P(3))
            endif

            ! Set up L, U, P
            N = 3
            call countMatrix(P, N)
            call zeroMatrix(L, N, N)

            ! Set up A
            A(1, 1) = 3
            A(2, 1) = -6
            A(3, 1) = 0
            A(1, 2) = 1
            A(2, 2) = 0
            A(3, 2) = 8
            A(1, 3) = 6
            A(2, 3) = -16
            A(3, 3) = -17

            P = (/1, 2, 3/)

        end subroutine

        subroutine fillMatrixTest2_10x10(A, L, P, N)
            real, allocatable, intent(inout) :: A(:, :), L(:, :)
            integer, allocatable, intent(inout) :: P(:)
            integer, intent(out) :: N

            N = 10

            if(.not. allocated(A)) then
                allocate(A(N, N))
```

```
endif

if(.not. allocated(L)) then
    allocate(L(N, N))
endif


if(.not. allocated(P)) then
    allocate(P(N))
endif

! Set up L, U, P
call countMatrix(P, N)
call zeroMatrix(L, N, N)

! Set up A
A(1, 1) = 59
A(1, 2) = 39
A(1, 3) = 21
A(1, 4) = 34
A(1, 5) = 38
A(1, 6) = 7
A(1, 7) = 68
A(1, 8) = 46
A(1, 9) = 39
A(1, 10) = 51
A(2, 1) = 28
A(2, 2) = 50
A(2, 3) = 46
A(2, 4) = 45
A(2, 5) = 29
A(2, 6) = 47
A(2, 7) = 27
A(2, 8) = 48
A(2, 9) = 3
A(2, 10) = 53
A(3, 1) = 14
A(3, 2) = 26
A(3, 3) = 7
A(3, 4) = 25
A(3, 5) = 49
A(3, 6) = 24
A(3, 7) = 14
A(3, 8) = 39
A(3, 9) = 56
A(3, 10) = 68
A(4, 1) = 63
A(4, 2) = 36
A(4, 3) = 45
A(4, 4) = 15
A(4, 5) = 26
A(4, 6) = 5
A(4, 7) = 42
A(4, 8) = 18
```

```
A(4, 9) = 1
A(4, 10) = 40
A(5, 1) = 62
A(5, 2) = 35
A(5, 3) = 66
A(5, 4) = 30
A(5, 5) = 37
A(5, 6) = 63
A(5, 7) = 35
A(5, 8) = 5
A(5, 9) = 20
A(5, 10) = 36
A(6, 1) = 61
A(6, 2) = 10
A(6, 3) = 64
A(6, 4) = 41
A(6, 5) = 66
A(6, 6) = 14
A(6, 7) = 41
A(6, 8) = 16
A(6, 9) = 38
A(6, 10) = 61
A(7, 1) = 2
A(7, 2) = 6
A(7, 3) = 55
A(7, 4) = 36
A(7, 5) = 35
A(7, 6) = 6
A(7, 7) = 14
A(7, 8) = 17
A(7, 9) = 16
A(7, 10) = 41
A(8, 1) = 55
A(8, 2) = 30
A(8, 3) = 31
A(8, 4) = 42
A(8, 5) = 4
A(8, 6) = 39
A(8, 7) = 31
A(8, 8) = 49
A(8, 9) = 44
A(8, 10) = 42
A(9, 1) = 50
A(9, 2) = 2
A(9, 3) = 9
A(9, 4) = 40
A(9, 5) = 23
A(9, 6) = 66
A(9, 7) = 54
A(9, 8) = 50
A(9, 9) = 64
A(9, 10) = 28
A(10, 1) = 5
A(10, 2) = 13
```

```
    A(10, 3) = 16
    A(10, 4) = 31
    A(10, 5) = 61
    A(10, 6) = 57
    A(10, 7) = 28
    A(10, 8) = 55
    A(10, 9) = 64
    A(10, 10) = 3

    P = (/1, 2, 3, 4, 5, 6, 7, 8, 9, 10/)

end subroutine

subroutine fillMatrixTest3_10x10(A, L, P, N)
    real, allocatable, intent(inout) :: A(:, :), L(:, :)
    integer, allocatable, intent(inout) :: P(:)
    integer, intent(out) :: N

    N = 10

    if(.not. allocated(A)) then
        allocate(A(N, N))
    endif

    if(.not. allocated(L)) then
        allocate(L(N, N))
    endif


    if(.not. allocated(P)) then
        allocate(P(N))
    endif

    ! Set up L, U, P
    call countMatrix(P, N)
    call zeroMatrix(L, N, N)

    ! Set up A
    A(1, 1) = 59
    A(1, 2) = 39
    A(1, 3) = 21
    A(1, 4) = 34
    A(1, 5) = 38
    A(1, 6) = 7
    A(1, 7) = -68
    A(1, 8) = 46
    A(1, 9) = 39
    A(1, 10) = 51
    A(2, 1) = 28
    A(2, 2) = 50
    A(2, 3) = 46
    A(2, 4) = 45
    A(2, 5) = 29
    A(2, 6) = 47
```

```
A(2, 7) = 27
A(2, 8) = 48
A(2, 9) = 3
A(2, 10) = 53
A(3, 1) = 14
A(3, 2) = 26
A(3, 3) = 7
A(3, 4) = 25
A(3, 5) = 49
A(3, 6) = 24
A(3, 7) = 14
A(3, 8) = 39
A(3, 9) = 56
A(3, 10) = 68
A(4, 1) = 63
A(4, 2) = 36
A(4, 3) = 45
A(4, 4) = 15
A(4, 5) = 26
A(4, 6) = 5
A(4, 7) = 42
A(4, 8) = 18
A(4, 9) = 1
A(4, 10) = 40
A(5, 1) = 62
A(5, 2) = 35
A(5, 3) = -66
A(5, 4) = 30
A(5, 5) = 37
A(5, 6) = 63
A(5, 7) = 35
A(5, 8) = 5
A(5, 9) = 20
A(5, 10) = 36
A(6, 1) = 61
A(6, 2) = 10
A(6, 3) = 64
A(6, 4) = 41
A(6, 5) = -66
A(6, 6) = 14
A(6, 7) = 41
A(6, 8) = 16
A(6, 9) = 38
A(6, 10) = 61
A(7, 1) = 2
A(7, 2) = 6
A(7, 3) = 55
A(7, 4) = 36
A(7, 5) = 35
A(7, 6) = 6
A(7, 7) = 14
A(7, 8) = 17
A(7, 9) = 16
A(7, 10) = 41
```

```
    A(8, 1) = 55
    A(8, 2) = 30
    A(8, 3) = 31
    A(8, 4) = 42
    A(8, 5) = 4
    A(8, 6) = 39
    A(8, 7) = 31
    A(8, 8) = 49
    A(8, 9) = 44
    A(8, 10) = 42
    A(9, 1) = 50
    A(9, 2) = 2
    A(9, 3) = 9
    A(9, 4) = 40
    A(9, 5) = 23
    A(9, 6) = 66
    A(9, 7) = 54
    A(9, 8) = 50
    A(9, 9) = 64
    A(9, 10) = 28
    A(10, 1) = 5
    A(10, 2) = 13
    A(10, 3) = 16
    A(10, 4) = 31
    A(10, 5) = 61
    A(10, 6) = 57
    A(10, 7) = 28
    A(10, 8) = 55
    A(10, 9) = 64
    A(10, 10) = 3

    P = (/1, 2, 3, 4, 5, 6, 7, 8, 9, 10/)

end subroutine


subroutine fillMatrixTest2_10x10Complex(A, L, P, N)
    complex, allocatable, intent(inout) :: A(:, :), L(:, :)
    integer, allocatable, intent(inout) :: P(:)
    integer, intent(out) :: N

    N = 10

    if(.not. allocated(A)) then
        allocate(A(N, N))
    endif

    if(.not. allocated(L)) then
        allocate(L(N, N))
    endif


    if(.not. allocated(P)) then
        allocate(P(N))
```

```
endif

! Set up L, U, P
call countMatrix(P, N)
call zeroMatrixComplex(L, N, N)

! Set up A
A(1, 1) = 59
A(1, 2) = 39
A(1, 3) = 21
A(1, 4) = 34
A(1, 5) = 38
A(1, 6) = 7
A(1, 7) = 68
A(1, 8) = 46
A(1, 9) = 39
A(1, 10) = 51
A(2, 1) = 28
A(2, 2) = 50
A(2, 3) = 46
A(2, 4) = 45
A(2, 5) = 29
A(2, 6) = 47
A(2, 7) = 27
A(2, 8) = 48
A(2, 9) = 3
A(2, 10) = 53
A(3, 1) = 14
A(3, 2) = 26
A(3, 3) = 7
A(3, 4) = 25
A(3, 5) = 49
A(3, 6) = 24
A(3, 7) = 14
A(3, 8) = 39
A(3, 9) = 56
A(3, 10) = 68
A(4, 1) = 63
A(4, 2) = 36
A(4, 3) = 45
A(4, 4) = 15
A(4, 5) = 26
A(4, 6) = 5
A(4, 7) = 42
A(4, 8) = 18
A(4, 9) = 1
A(4, 10) = 40
A(5, 1) = 62
A(5, 2) = 35
A(5, 3) = 66
A(5, 4) = 30
A(5, 5) = 37
A(5, 6) = 63
A(5, 7) = 35
```

```
A(5, 8) = 5
A(5, 9) = 20
A(5, 10) = 36
A(6, 1) = 61
A(6, 2) = 10
A(6, 3) = 64
A(6, 4) = 41
A(6, 5) = 66
A(6, 6) = 14
A(6, 7) = 41
A(6, 8) = 16
A(6, 9) = 38
A(6, 10) = 61
A(7, 1) = 2
A(7, 2) = 6
A(7, 3) = 55
A(7, 4) = 36
A(7, 5) = 35
A(7, 6) = 6
A(7, 7) = 14
A(7, 8) = 17
A(7, 9) = 16
A(7, 10) = 41
A(8, 1) = 55
A(8, 2) = 30
A(8, 3) = 31
A(8, 4) = 42
A(8, 5) = 4
A(8, 6) = 39
A(8, 7) = 31
A(8, 8) = 49
A(8, 9) = 44
A(8, 10) = 42
A(9, 1) = 50
A(9, 2) = 2
A(9, 3) = 9
A(9, 4) = 40
A(9, 5) = 23
A(9, 6) = 66
A(9, 7) = 54
A(9, 8) = 50
A(9, 9) = 64
A(9, 10) = 28
A(10, 1) = 5
A(10, 2) = 13
A(10, 3) = 16
A(10, 4) = 31
A(10, 5) = 61
A(10, 6) = 57
A(10, 7) = 28
A(10, 8) = 55
A(10, 9) = 64
A(10, 10) = 3
```

```
    P = (/1, 2, 3, 4, 5, 6, 7, 8, 9, 10/)

end subroutine

subroutine fillMatrixTest3_10x10Complex(A, L, P, N)
    complex, allocatable, intent(inout) :: A(:, :), L(:, :)
    integer, allocatable, intent(inout) :: P(:)
    integer, intent(out) :: N

    N = 10

    if(.not. allocated(A)) then
        allocate(A(N, N))
    endif

    if(.not. allocated(L)) then
        allocate(L(N, N))
    endif


    if(.not. allocated(P)) then
        allocate(P(N))
    endif

    ! Set up L, U, P
    call countMatrix(P, N)
    call zeroMatrixComplex(L, N, N)

    ! Set up A
    A(1, 1) = 59
    A(1, 2) = 39
    A(1, 3) = 21
    A(1, 4) = 34
    A(1, 5) = 38
    A(1, 6) = 7
    A(1, 7) = -68
    A(1, 8) = 46
    A(1, 9) = 39
    A(1, 10) = 51
    A(2, 1) = 28
    A(2, 2) = 50
    A(2, 3) = 46
    A(2, 4) = 45
    A(2, 5) = 29
    A(2, 6) = 47
    A(2, 7) = 27
    A(2, 8) = 48
    A(2, 9) = 3
    A(2, 10) = 53
    A(3, 1) = 14
    A(3, 2) = 26
    A(3, 3) = 7
    A(3, 4) = 25
    A(3, 5) = 49
```

```
A(3, 6) = 24
A(3, 7) = 14
A(3, 8) = 39
A(3, 9) = 56
A(3, 10) = 68
A(4, 1) = 63
A(4, 2) = 36
A(4, 3) = 45
A(4, 4) = 15
A(4, 5) = 26
A(4, 6) = 5
A(4, 7) = 42
A(4, 8) = 18
A(4, 9) = 1
A(4, 10) = 40
A(5, 1) = 62
A(5, 2) = 35
A(5, 3) = -66
A(5, 4) = 30
A(5, 5) = 37
A(5, 6) = 63
A(5, 7) = 35
A(5, 8) = 5
A(5, 9) = 20
A(5, 10) = 36
A(6, 1) = 61
A(6, 2) = 10
A(6, 3) = 64
A(6, 4) = 41
A(6, 5) = -66
A(6, 6) = 14
A(6, 7) = 41
A(6, 8) = 16
A(6, 9) = 38
A(6, 10) = 61
A(7, 1) = 2
A(7, 2) = 6
A(7, 3) = 55
A(7, 4) = 36
A(7, 5) = 35
A(7, 6) = 6
A(7, 7) = 14
A(7, 8) = 17
A(7, 9) = 16
A(7, 10) = 41
A(8, 1) = 55
A(8, 2) = 30
A(8, 3) = 31
A(8, 4) = 42
A(8, 5) = 4
A(8, 6) = 39
A(8, 7) = 31
A(8, 8) = 49
A(8, 9) = 44
```

```
    A(8, 10) = 42
    A(9, 1) = 50
    A(9, 2) = 2
    A(9, 3) = 9
    A(9, 4) = 40
    A(9, 5) = 23
    A(9, 6) = 66
    A(9, 7) = 54
    A(9, 8) = 50
    A(9, 9) = 64
    A(9, 10) = 28
    A(10, 1) = 5
    A(10, 2) = 13
    A(10, 3) = 16
    A(10, 4) = 31
    A(10, 5) = 61
    A(10, 6) = 57
    A(10, 7) = 28
    A(10, 8) = 55
    A(10, 9) = 64
    A(10, 10) = 3

    P = (/1, 2, 3, 4, 5, 6, 7, 8, 9, 10/)

end subroutine

subroutine fillMatrixTest4_10x10Complex(A, L, P, N)
    complex, allocatable, intent(inout) :: A(:, :), L(:, :)
    integer, allocatable, intent(inout) :: P(:)
    integer, intent(out) :: N
    Complex :: cvval, czval, czmat

    N=10

    if(.not. allocated(A)) then
        allocate(A(N, N))
    endif

    if(.not. allocated(L)) then
        allocate(L(N, N))
    endif


    if(.not. allocated(P)) then
        allocate(P(N))
    endif

    ! Set up L, U, P
    call countMatrix(P, N)
    call zeroMatrixComplex(L, N, N)

    Do jx=1,N
        Do ix=1,N
            Call zmatrxMatTest(jx,ix,czval)
```

```
            A(jx,ix)=czval
        End do
    End do

endsubroutine

subroutine fillMatrixTest5_25x25Complex(A, L, P, N)
    complex, allocatable, intent(inout) :: A(:, :), L(:, :)
    integer, allocatable, intent(inout) :: P(:)
    integer, intent(out) :: N
    Complex :: cvval, czval, czmat

    N=25

    if(.not. allocated(A)) then
        allocate(A(N, N))
    endif

    if(.not. allocated(L)) then
        allocate(L(N, N))
    endif


    if(.not. allocated(P)) then
        allocate(P(N))
    endif

    ! Set up L, U, P
    call countMatrix(P, N)
    call zeroMatrixComplex(L, N, N)

    Do jx=1,N
        Do ix=1,N
            Call zmatrxMatTest(jx,ix,czval)
            A(jx,ix)=czval
        End do
    End do

endsubroutine

subroutine fillMatrixTest6_100x100Complex(A, L, P, N)
    complex, allocatable, intent(inout) :: A(:, :), L(:, :)
    integer, allocatable, intent(inout) :: P(:)
    integer, intent(out) :: N
    Complex :: cvval, czval, czmat

    N=100

    if(.not. allocated(A)) then
        allocate(A(N, N))
    endif

    if(.not. allocated(L)) then
        allocate(L(N, N))
```

```
    endif


    if(.not. allocated(P)) then
        allocate(P(N))
    endif

    ! Set up L, U, P
    call countMatrix(P, N)
    call zeroMatrixComplex(L, N, N)

    Do jx=1,N
        Do ix=1,N
            Call zmatrxMatTest(jx,ix,czval)
            A(jx,ix)=czval
        End do
    End do

endsubroutine

subroutine fillMatrixTest7_1000x1000Complex(A, L, P, N)
    complex, allocatable, intent(inout) :: A(:, :), L(:, :)
    integer, allocatable, intent(inout) :: P(:)
    integer, intent(out) :: N
    Complex :: cvval, czval, czmat

    N=1000

    if(.not. allocated(A)) then
        allocate(A(N, N))
    endif

    if(.not. allocated(L)) then
        allocate(L(N, N))
    endif


    if(.not. allocated(P)) then
        allocate(P(N))
    endif

    ! Set up L, U, P
    call countMatrix(P, N)
    call zeroMatrixComplex(L, N, N)

    Do jx=1,N
        Do ix=1,N
            Call zmatrxMatTest(jx,ix,czval)
            A(jx,ix)=czval
        End do
    End do

endsubroutine
```

```
subroutine fillMatrixTest9_25000x25000Complex(A, L, P, N)
    complex, allocatable, intent(inout) :: A(:, :), L(:, :)
    integer, allocatable, intent(inout) :: P(:)
    integer, intent(out) :: N
    Complex :: cvval, czval, czmat

    N=25000

    if(.not. allocated(A)) then
        allocate(A(N, N))
    endif

    if(.not. allocated(L)) then
        allocate(L(N, N))
    endif


    if(.not. allocated(P)) then
        allocate(P(N))
    endif

    ! Set up L, U, P
    call countMatrix(P, N)
    call zeroMatrixComplex(L, N, N)

    Do jx=1,N
        Do ix=1,N
            Call zmatrxMatTest(jx,ix,czval)
            A(jx,ix)=czval
        End do
    End do

endsubroutine

subroutine fillMatrixTest8_10000x10000Complex(A, L, P, N)
    complex, allocatable, intent(inout) :: A(:, :), L(:, :)
    integer, allocatable, intent(inout) :: P(:)
    integer, intent(out) :: N
    Complex :: cvval, czval, czmat

    N=10000

    if(.not. allocated(A)) then
        allocate(A(N, N))
    endif

    if(.not. allocated(L)) then
        allocate(L(N, N))
    endif


    if(.not. allocated(P)) then
        allocate(P(N))
    endif
```

```
    ! Set up L, U, P
    call countMatrix(P, N)
    call zeroMatrixComplex(L, N, N)

    Do jx=1,N
        Do ix=1,N
            Call zmatrxMatTest(jx,ix,czval)
            A(jx,ix)=czval
        End do
    End do

endsubroutine

subroutine fillMatrixTest10_50000x50000Complex(A, L, P, N)

    complex, allocatable, intent(inout) :: A(:, :), L(:, :)
    integer, allocatable, intent(inout) :: P(:)
    integer, intent(out) :: N
    Complex :: cvval, czval, czmat

    N=50000

    if(.not. allocated(A)) then
        allocate(A(N, N))
    endif

    if(.not. allocated(L)) then
        allocate(L(N, N))
    endif


    if(.not. allocated(P)) then
        allocate(P(N))
    endif

    ! Set up L, U, P
    call countMatrix(P, N)
    call zeroMatrixComplex(L, N, N)

    Do jx=1,N
        Do ix=1,N
            Call zmatrxMatTest(jx,ix,czval)
            A(jx,ix)=czval
        End do
    End do

endsubroutine

subroutine vmatrxMatTest(jx,cvval,nbf)
    implicit complex (c)
    cvval=cmplx(0.0,0.0)

    do ix=1,nbf
```

```
            call zmatrxMatTest(jx,ix,czval)
            cvval=cvval+czval
        end do

        return
    end

    subroutine zmatrxMatTest(jx,ix,czval)
        implicit complex (c)
        val=float(iabs(jx-ix)+1)
        valsq=val*val
        cdnm=cmplx(val,val)
        czval=cmplx(1.0,0.0)/cdnm
        return
    end

end module
```

```
module matrixUtil

    implicit none

    contains

    subroutine vecConstantCheck(vec, val, tolerance, N, outcome)
        complex, allocatable, intent(in) :: vec(:)
        complex :: val
        real :: tolerance
        integer, intent(in) :: N
        integer, intent(inout) :: outcome
        integer :: index
        complex :: check

        outcome = 1
        do index = 1, N

            if((real(vec(index)) > real(val) * (1 + tolerance) .or.
real(vec(index)) < real(val) * (1 - tolerance)) .and. (imag(vec(index)) >
imag(val) * (1 + tolerance) .or. imag(vec(index)) < imag(val) * (1 -
tolerance))) then

                outcome = 0
                return

            endif

        enddo
    endsubroutine

end module
```

```
! -*- Mode: F90; -*-
! Copyright(c) Microsoft Corporation.All rights reserved.
! Licensed under the MIT License.
!
!      (C) 2004 by Argonne National Laboratory.
!      (C) 2015 by Microsoft Corporation
!
!                                    MPICH COPYRIGHT
!
!   The following is a notice of limited availability of the code, and
disclaimer
!   which must be included in the prologue of the code and in all source
listings
!   of the code.
!
!   Copyright Notice
!    + 2002 University of Chicago
!
!   Permission is hereby granted to use, reproduce, prepare derivative
works, and
!   to redistribute to others.  This software was authored by:
!
!   Mathematics and Computer Science Division
!   Argonne National Laboratory, Argonne IL 60439
!
!   (and)
!
!   Department of Computer Science
!   University of Illinois at Urbana-Champaign
!
!
!                                   GOVERNMENT LICENSE
!
!   Portions of this material resulted from work developed under a U.S.
!   Government Contract and are subject to the following license: the
Government
!   is granted for itself and others acting on its behalf a paid-up,
nonexclusive,
!   irrevocable worldwide license in this computer software to reproduce,
prepare
!   derivative works, and perform publicly and display publicly.
!
!                                       DISCLAIMER
!
!   This computer code material was prepared, in part, as an account of
work
!   sponsored by an agency of the United States Government.  Neither the
United
!   States, nor the University of Chicago, nor any of their employees,
makes any
!   warranty express or implied, or assumes any legal liability or
responsibility
!   for the accuracy, completeness, or usefulness of any information,
apparatus,
```

```
!   product, or process disclosed, or represents that its use would not
infringe
!   privately owned rights.
!
!
       MODULE MPI_CONSTANTS
       IMPLICIT NONE

       INTEGER MPI_SOURCE, MPI_TAG, MPI_ERROR
       PARAMETER (MPI_SOURCE=3,MPI_TAG=4,MPI_ERROR=5)
       INTEGER MPI_STATUS_SIZE
       PARAMETER (MPI_STATUS_SIZE=5)
       INTEGER MPI_STATUS_IGNORE(MPI_STATUS_SIZE)
       INTEGER MPI_STATUSES_IGNORE(MPI_STATUS_SIZE,1)
       INTEGER MPI_ERRCODES_IGNORE(1)
       CHARACTER*1 MPI_ARGVS_NULL(1,1)
       CHARACTER*1 MPI_ARGV_NULL(1)
       INTEGER MPI_SUCCESS
       PARAMETER (MPI_SUCCESS=0)
       INTEGER MPI_ERR_OTHER
       PARAMETER (MPI_ERR_OTHER=15)
       INTEGER MPI_ERR_WIN
       PARAMETER (MPI_ERR_WIN=45)
       INTEGER MPI_ERR_FILE
       PARAMETER (MPI_ERR_FILE=27)
       INTEGER MPI_ERR_COUNT
       PARAMETER (MPI_ERR_COUNT=2)
       INTEGER MPI_ERR_SPAWN
       PARAMETER (MPI_ERR_SPAWN=42)
       INTEGER MPI_ERR_BASE
       PARAMETER (MPI_ERR_BASE=46)
       INTEGER MPI_ERR_RMA_CONFLICT
       PARAMETER (MPI_ERR_RMA_CONFLICT=49)
       INTEGER MPI_ERR_IN_STATUS
       PARAMETER (MPI_ERR_IN_STATUS=17)
       INTEGER MPI_ERR_INFO_KEY
       PARAMETER (MPI_ERR_INFO_KEY=29)
       INTEGER MPI_ERR_LOCKTYPE
       PARAMETER (MPI_ERR_LOCKTYPE=47)
       INTEGER MPI_ERR_OP
       PARAMETER (MPI_ERR_OP=9)
       INTEGER MPI_ERR_ARG
       PARAMETER (MPI_ERR_ARG=12)
       INTEGER MPI_ERR_READ_ONLY
       PARAMETER (MPI_ERR_READ_ONLY=40)
       INTEGER MPI_ERR_SIZE
       PARAMETER (MPI_ERR_SIZE=51)
       INTEGER MPI_ERR_BUFFER
       PARAMETER (MPI_ERR_BUFFER=1)
       INTEGER MPI_ERR_DUP_DATAREP
       PARAMETER (MPI_ERR_DUP_DATAREP=24)
       INTEGER MPI_ERR_UNSUPPORTED_DATAREP
       PARAMETER (MPI_ERR_UNSUPPORTED_DATAREP=43)
       INTEGER MPI_ERR_LASTCODE
```

```
PARAMETER (MPI_ERR_LASTCODE=1073741823)
INTEGER MPI_ERR_TRUNCATE
PARAMETER (MPI_ERR_TRUNCATE=14)
INTEGER MPI_ERR_DISP
PARAMETER (MPI_ERR_DISP=52)
INTEGER MPI_ERR_PORT
PARAMETER (MPI_ERR_PORT=38)
INTEGER MPI_ERR_INFO_NOKEY
PARAMETER (MPI_ERR_INFO_NOKEY=31)
INTEGER MPI_ERR_ASSERT
PARAMETER (MPI_ERR_ASSERT=53)
INTEGER MPI_ERR_FILE_EXISTS
PARAMETER (MPI_ERR_FILE_EXISTS=25)
INTEGER MPI_ERR_PENDING
PARAMETER (MPI_ERR_PENDING=18)
INTEGER MPI_ERR_COMM
PARAMETER (MPI_ERR_COMM=5)
INTEGER MPI_ERR_KEYVAL
PARAMETER (MPI_ERR_KEYVAL=48)
INTEGER MPI_ERR_NAME
PARAMETER (MPI_ERR_NAME=33)
INTEGER MPI_ERR_REQUEST
PARAMETER (MPI_ERR_REQUEST=19)
INTEGER MPI_ERR_GROUP
PARAMETER (MPI_ERR_GROUP=8)
INTEGER MPI_ERR_TOPOLOGY
PARAMETER (MPI_ERR_TOPOLOGY=10)
INTEGER MPI_ERR_TYPE
PARAMETER (MPI_ERR_TYPE=3)
INTEGER MPI_ERR_TAG
PARAMETER (MPI_ERR_TAG=4)
INTEGER MPI_ERR_INFO_VALUE
PARAMETER (MPI_ERR_INFO_VALUE=30)
INTEGER MPI_ERR_NOT_SAME
PARAMETER (MPI_ERR_NOT_SAME=35)
INTEGER MPI_ERR_RMA_SYNC
PARAMETER (MPI_ERR_RMA_SYNC=50)
INTEGER MPI_ERR_INFO
PARAMETER (MPI_ERR_INFO=28)
INTEGER MPI_ERR_NO_MEM
PARAMETER (MPI_ERR_NO_MEM=34)
INTEGER MPI_ERR_BAD_FILE
PARAMETER (MPI_ERR_BAD_FILE=22)
INTEGER MPI_ERR_FILE_IN_USE
PARAMETER (MPI_ERR_FILE_IN_USE=26)
INTEGER MPI_ERR_UNKNOWN
PARAMETER (MPI_ERR_UNKNOWN=13)
INTEGER MPI_ERR_UNSUPPORTED_OPERATION
PARAMETER (MPI_ERR_UNSUPPORTED_OPERATION=44)
INTEGER MPI_ERR_QUOTA
PARAMETER (MPI_ERR_QUOTA=39)
INTEGER MPI_ERR_AMODE
PARAMETER (MPI_ERR_AMODE=21)
INTEGER MPI_ERR_ROOT
```

```
PARAMETER (MPI_ERR_ROOT=7)
INTEGER MPI_ERR_RANK
PARAMETER (MPI_ERR_RANK=6)
INTEGER MPI_ERR_DIMS
PARAMETER (MPI_ERR_DIMS=11)
INTEGER MPI_ERR_NO_SUCH_FILE
PARAMETER (MPI_ERR_NO_SUCH_FILE=37)
INTEGER MPI_ERR_SERVICE
PARAMETER (MPI_ERR_SERVICE=41)
INTEGER MPI_ERR_INTERN
PARAMETER (MPI_ERR_INTERN=16)
INTEGER MPI_ERR_IO
PARAMETER (MPI_ERR_IO=32)
INTEGER MPI_ERR_ACCESS
PARAMETER (MPI_ERR_ACCESS=20)
INTEGER MPI_ERR_NO_SPACE
PARAMETER (MPI_ERR_NO_SPACE=36)
INTEGER MPI_ERR_CONVERSION
PARAMETER (MPI_ERR_CONVERSION=23)
INTEGER MPI_ERRORS_ARE_FATAL
PARAMETER (MPI_ERRORS_ARE_FATAL=1409286144)
INTEGER MPI_ERRORS_RETURN
PARAMETER (MPI_ERRORS_RETURN=1409286145)
INTEGER MPI_IDENT
PARAMETER (MPI_IDENT=0)
INTEGER MPI_CONGRUENT
PARAMETER (MPI_CONGRUENT=1)
INTEGER MPI_SIMILAR
PARAMETER (MPI_SIMILAR=2)
INTEGER MPI_UNEQUAL
PARAMETER (MPI_UNEQUAL=3)
INTEGER MPI_MAX
PARAMETER (MPI_MAX=1476395009)
INTEGER MPI_MIN
PARAMETER (MPI_MIN=1476395010)
INTEGER MPI_SUM
PARAMETER (MPI_SUM=1476395011)
INTEGER MPI_PROD
PARAMETER (MPI_PROD=1476395012)
INTEGER MPI_LAND
PARAMETER (MPI_LAND=1476395013)
INTEGER MPI_BAND
PARAMETER (MPI_BAND=1476395014)
INTEGER MPI_LOR
PARAMETER (MPI_LOR=1476395015)
INTEGER MPI_BOR
PARAMETER (MPI_BOR=1476395016)
INTEGER MPI_LXOR
PARAMETER (MPI_LXOR=1476395017)
INTEGER MPI_BXOR
PARAMETER (MPI_BXOR=1476395018)
INTEGER MPI_MINLOC
PARAMETER (MPI_MINLOC=1476395019)
INTEGER MPI_MAXLOC
```

```
PARAMETER (MPI_MAXLOC=1476395020)
INTEGER MPI_REPLACE
PARAMETER (MPI_REPLACE=1476395021)
INTEGER MPI_NO_OP
PARAMETER (MPI_NO_OP=1476395022)
INTEGER MPI_COMM_WORLD
PARAMETER (MPI_COMM_WORLD=1140850688)
INTEGER MPI_COMM_SELF
PARAMETER (MPI_COMM_SELF=1140850689)
INTEGER MPI_COMM_TYPE_SHARED
PARAMETER (MPI_COMM_TYPE_SHARED=1)
INTEGER MPI_GROUP_EMPTY
PARAMETER (MPI_GROUP_EMPTY=1207959552)
INTEGER MPI_COMM_NULL
PARAMETER (MPI_COMM_NULL=67108864)
INTEGER MPI_WIN_NULL
PARAMETER (MPI_WIN_NULL=536870912)
INTEGER MPI_FILE_NULL
PARAMETER (MPI_FILE_NULL=0)
INTEGER MPI_GROUP_NULL
PARAMETER (MPI_GROUP_NULL=134217728)
INTEGER MPI_OP_NULL
PARAMETER (MPI_OP_NULL=402653184)
INTEGER MPI_DATATYPE_NULL
PARAMETER (MPI_DATATYPE_NULL=z'0c000000')
INTEGER MPI_REQUEST_NULL
PARAMETER (MPI_REQUEST_NULL=738197504)
INTEGER MPI_ERRHANDLER_NULL
PARAMETER (MPI_ERRHANDLER_NULL=335544320)
INTEGER MPI_INFO_NULL
PARAMETER (MPI_INFO_NULL=469762048)
INTEGER MPI_MESSAGE_NULL
PARAMETER (MPI_MESSAGE_NULL=805306368)
INTEGER MPI_MESSAGE_NO_PROC
PARAMETER (MPI_MESSAGE_NO_PROC=1879048192)
INTEGER MPI_TAG_UB
PARAMETER (MPI_TAG_UB=1681915906)
INTEGER MPI_HOST
PARAMETER (MPI_HOST=1681915908)
INTEGER MPI_IO
PARAMETER (MPI_IO=1681915910)
INTEGER MPI_WTIME_IS_GLOBAL
PARAMETER (MPI_WTIME_IS_GLOBAL=1681915912)
INTEGER MPI_UNIVERSE_SIZE
PARAMETER (MPI_UNIVERSE_SIZE=1681915914)
INTEGER MPI_LASTUSEDCODE
PARAMETER (MPI_LASTUSEDCODE=1681915916)
INTEGER MPI_APPNUM
PARAMETER (MPI_APPNUM=1681915918)
INTEGER MPI_WIN_BASE
PARAMETER (MPI_WIN_BASE=1711276034)
INTEGER MPI_WIN_SIZE
PARAMETER (MPI_WIN_SIZE=1711276036)
INTEGER MPI_WIN_DISP_UNIT
```

```
PARAMETER (MPI_WIN_DISP_UNIT=1711276038)
INTEGER MPI_MAX_ERROR_STRING
PARAMETER (MPI_MAX_ERROR_STRING=511)
INTEGER MPI_MAX_PORT_NAME
PARAMETER (MPI_MAX_PORT_NAME=255)
INTEGER MPI_MAX_OBJECT_NAME
PARAMETER (MPI_MAX_OBJECT_NAME=127)
INTEGER MPI_MAX_INFO_KEY
PARAMETER (MPI_MAX_INFO_KEY=254)
INTEGER MPI_MAX_INFO_VAL
PARAMETER (MPI_MAX_INFO_VAL=1023)
INTEGER MPI_MAX_PROCESSOR_NAME
PARAMETER (MPI_MAX_PROCESSOR_NAME=128-1)
INTEGER MPI_MAX_DATAREP_STRING
PARAMETER (MPI_MAX_DATAREP_STRING=127)
INTEGER MPI_MAX_LIBRARY_VERSION_STRING
PARAMETER (MPI_MAX_LIBRARY_VERSION_STRING=64-1)
INTEGER MPI_UNDEFINED
PARAMETER (MPI_UNDEFINED=(-32766))
INTEGER MPI_KEYVAL_INVALID
PARAMETER (MPI_KEYVAL_INVALID=603979776)
INTEGER MPI_BSEND_OVERHEAD
PARAMETER (MPI_BSEND_OVERHEAD=(95))
INTEGER MPI_PROC_NULL
PARAMETER (MPI_PROC_NULL=-1)
INTEGER MPI_ANY_SOURCE
PARAMETER (MPI_ANY_SOURCE=-2)
INTEGER MPI_ANY_TAG
PARAMETER (MPI_ANY_TAG=-1)
INTEGER MPI_ROOT
PARAMETER (MPI_ROOT=-3)
INTEGER MPI_GRAPH
PARAMETER (MPI_GRAPH=1)
INTEGER MPI_CART
PARAMETER (MPI_CART=2)
INTEGER MPI_DIST_GRAPH
PARAMETER (MPI_DIST_GRAPH=3)
INTEGER MPI_VERSION
PARAMETER (MPI_VERSION=2)
INTEGER MPI_SUBVERSION
PARAMETER (MPI_SUBVERSION=0)
INTEGER MPI_LOCK_EXCLUSIVE
PARAMETER (MPI_LOCK_EXCLUSIVE=234)
INTEGER MPI_LOCK_SHARED
PARAMETER (MPI_LOCK_SHARED=235)
INTEGER MPI_CHAR
PARAMETER (MPI_CHAR=z'4c000101')
INTEGER MPI_UNSIGNED_CHAR
PARAMETER (MPI_UNSIGNED_CHAR=z'4c000102')
INTEGER MPI_SHORT
PARAMETER (MPI_SHORT=z'4c000203')
INTEGER MPI_UNSIGNED_SHORT
PARAMETER (MPI_UNSIGNED_SHORT=z'4c000204')
INTEGER MPI_INT
```

```
PARAMETER (MPI_INT=z'4c000405')
INTEGER MPI_UNSIGNED
PARAMETER (MPI_UNSIGNED=z'4c000406')
INTEGER MPI_LONG
PARAMETER (MPI_LONG=z'4c000407')
INTEGER MPI_UNSIGNED_LONG
PARAMETER (MPI_UNSIGNED_LONG=z'4c000408')
INTEGER MPI_LONG_LONG
PARAMETER (MPI_LONG_LONG=z'4c000809')
INTEGER MPI_LONG_LONG_INT
PARAMETER (MPI_LONG_LONG_INT=z'4c000809')
INTEGER MPI_FLOAT
PARAMETER (MPI_FLOAT=z'4c00040a')
INTEGER MPI_DOUBLE
PARAMETER (MPI_DOUBLE=z'4c00080b')
INTEGER MPI_LONG_DOUBLE
PARAMETER (MPI_LONG_DOUBLE=z'4c00080c')
INTEGER MPI_BYTE
PARAMETER (MPI_BYTE=z'4c00010d')
INTEGER MPI_WCHAR
PARAMETER (MPI_WCHAR=z'4c00020e')
INTEGER MPI_PACKED
PARAMETER (MPI_PACKED=z'4c00010f')
INTEGER MPI_LB
PARAMETER (MPI_LB=z'4c000010')
INTEGER MPI_UB
PARAMETER (MPI_UB=z'4c000011')
INTEGER MPI_2INT
PARAMETER (MPI_2INT=z'4c000816')
INTEGER MPI_SIGNED_CHAR
PARAMETER (MPI_SIGNED_CHAR=z'4c000118')
INTEGER MPI_UNSIGNED_LONG_LONG
PARAMETER (MPI_UNSIGNED_LONG_LONG=z'4c000819')
INTEGER MPI_CHARACTER
PARAMETER (MPI_CHARACTER=z'4c00011a')
INTEGER MPI_INTEGER
PARAMETER (MPI_INTEGER=z'4c00041b')
INTEGER MPI_REAL
PARAMETER (MPI_REAL=z'4c00041c')
INTEGER MPI_LOGICAL
PARAMETER (MPI_LOGICAL=z'4c00041d')
INTEGER MPI_COMPLEX
PARAMETER (MPI_COMPLEX=z'4c00081e')
INTEGER MPI_DOUBLE_PRECISION
PARAMETER (MPI_DOUBLE_PRECISION=z'4c00081f')
INTEGER MPI_2INTEGER
PARAMETER (MPI_2INTEGER=z'4c000820')
INTEGER MPI_2REAL
PARAMETER (MPI_2REAL=z'4c000821')
INTEGER MPI_DOUBLE_COMPLEX
PARAMETER (MPI_DOUBLE_COMPLEX=z'4c001022')
INTEGER MPI_2DOUBLE_PRECISION
PARAMETER (MPI_2DOUBLE_PRECISION=z'4c001023')
INTEGER MPI_2COMPLEX
```

```
PARAMETER (MPI_2COMPLEX=z'4c001024')
INTEGER MPI_2DOUBLE_COMPLEX
PARAMETER (MPI_2DOUBLE_COMPLEX=z'4c002025')
INTEGER MPI_REAL2
PARAMETER (MPI_REAL2=z'0c000000')
INTEGER MPI_REAL4
PARAMETER (MPI_REAL4=z'4c000427')
INTEGER MPI_COMPLEX8
PARAMETER (MPI_COMPLEX8=z'4c000828')
INTEGER MPI_REAL8
PARAMETER (MPI_REAL8=z'4c000829')
INTEGER MPI_COMPLEX16
PARAMETER (MPI_COMPLEX16=z'4c00102a')
INTEGER MPI_REAL16
PARAMETER (MPI_REAL16=z'0c000000')
INTEGER MPI_COMPLEX32
PARAMETER (MPI_COMPLEX32=z'0c000000')
INTEGER MPI_INTEGER1
PARAMETER (MPI_INTEGER1=z'4c00012d')
INTEGER MPI_COMPLEX4
PARAMETER (MPI_COMPLEX4=z'0c000000')
INTEGER MPI_INTEGER2
PARAMETER (MPI_INTEGER2=z'4c00022f')
INTEGER MPI_INTEGER4
PARAMETER (MPI_INTEGER4=z'4c000430')
INTEGER MPI_INTEGER8
PARAMETER (MPI_INTEGER8=z'4c000831')
INTEGER MPI_INTEGER16
PARAMETER (MPI_INTEGER16=z'0c000000')

INCLUDE 'mpifptr.h'

INTEGER MPI_OFFSET
PARAMETER (MPI_OFFSET=z'4c00083c')
INTEGER MPI_COUNT
PARAMETER (MPI_COUNT=z'4c00083d')
INTEGER MPI_FLOAT_INT
PARAMETER (MPI_FLOAT_INT=z'8c000000')
INTEGER MPI_DOUBLE_INT
PARAMETER (MPI_DOUBLE_INT=z'8c000001')
INTEGER MPI_LONG_INT
PARAMETER (MPI_LONG_INT=z'8c000002')
INTEGER MPI_SHORT_INT
PARAMETER (MPI_SHORT_INT=z'8c000003')
INTEGER MPI_LONG_DOUBLE_INT
PARAMETER (MPI_LONG_DOUBLE_INT=z'8c000004')
INTEGER MPI_INTEGER_KIND
PARAMETER (MPI_INTEGER_KIND=4)
INTEGER MPI_OFFSET_KIND
PARAMETER (MPI_OFFSET_KIND=8)
INTEGER MPI_COUNT_KIND
PARAMETER (MPI_COUNT_KIND=8)
INTEGER MPI_COMBINER_NAMED
PARAMETER (MPI_COMBINER_NAMED=1)
```

```
INTEGER MPI_COMBINER_DUP
PARAMETER (MPI_COMBINER_DUP=2)
INTEGER MPI_COMBINER_CONTIGUOUS
PARAMETER (MPI_COMBINER_CONTIGUOUS=3)
INTEGER MPI_COMBINER_VECTOR
PARAMETER (MPI_COMBINER_VECTOR=4)
INTEGER MPI_COMBINER_HVECTOR_INTEGER
PARAMETER (MPI_COMBINER_HVECTOR_INTEGER=5)
INTEGER MPI_COMBINER_HVECTOR
PARAMETER (MPI_COMBINER_HVECTOR=6)
INTEGER MPI_COMBINER_INDEXED
PARAMETER (MPI_COMBINER_INDEXED=7)
INTEGER MPI_COMBINER_HINDEXED_INTEGER
PARAMETER (MPI_COMBINER_HINDEXED_INTEGER=8)
INTEGER MPI_COMBINER_HINDEXED
PARAMETER (MPI_COMBINER_HINDEXED=9)
INTEGER MPI_COMBINER_INDEXED_BLOCK
PARAMETER (MPI_COMBINER_INDEXED_BLOCK=10)
INTEGER MPI_COMBINER_STRUCT_INTEGER
PARAMETER (MPI_COMBINER_STRUCT_INTEGER=11)
INTEGER MPI_COMBINER_STRUCT
PARAMETER (MPI_COMBINER_STRUCT=12)
INTEGER MPI_COMBINER_SUBARRAY
PARAMETER (MPI_COMBINER_SUBARRAY=13)
INTEGER MPI_COMBINER_DARRAY
PARAMETER (MPI_COMBINER_DARRAY=14)
INTEGER MPI_COMBINER_F90_REAL
PARAMETER (MPI_COMBINER_F90_REAL=15)
INTEGER MPI_COMBINER_F90_COMPLEX
PARAMETER (MPI_COMBINER_F90_COMPLEX=16)
INTEGER MPI_COMBINER_F90_INTEGER
PARAMETER (MPI_COMBINER_F90_INTEGER=17)
INTEGER MPI_COMBINER_RESIZED
PARAMETER (MPI_COMBINER_RESIZED=18)
INTEGER MPI_COMBINER_HINDEXED_BLOCK
PARAMETER (MPI_COMBINER_HINDEXED_BLOCK=19)
INTEGER MPI_MODE_NOCHECK
PARAMETER (MPI_MODE_NOCHECK=1024)
INTEGER MPI_MODE_NOSTORE
PARAMETER (MPI_MODE_NOSTORE=2048)
INTEGER MPI_MODE_NOPUT
PARAMETER (MPI_MODE_NOPUT=4096)
INTEGER MPI_MODE_NOPRECEDE
PARAMETER (MPI_MODE_NOPRECEDE=8192)
INTEGER MPI_MODE_NOSUCCEED
PARAMETER (MPI_MODE_NOSUCCEED=16384)
INTEGER MPI_THREAD_SINGLE
PARAMETER (MPI_THREAD_SINGLE=0)
INTEGER MPI_THREAD_FUNNELED
PARAMETER (MPI_THREAD_FUNNELED=1)
INTEGER MPI_THREAD_SERIALIZED
PARAMETER (MPI_THREAD_SERIALIZED=2)
INTEGER MPI_THREAD_MULTIPLE
PARAMETER (MPI_THREAD_MULTIPLE=3)
```

```
       INTEGER MPI_MODE_RDONLY
       PARAMETER (MPI_MODE_RDONLY=2)
       INTEGER MPI_MODE_RDWR
       PARAMETER (MPI_MODE_RDWR=8)
       INTEGER MPI_MODE_WRONLY
       PARAMETER (MPI_MODE_WRONLY=4)
       INTEGER MPI_MODE_DELETE_ON_CLOSE
       PARAMETER (MPI_MODE_DELETE_ON_CLOSE=16)
       INTEGER MPI_MODE_UNIQUE_OPEN
       PARAMETER (MPI_MODE_UNIQUE_OPEN=32)
       INTEGER MPI_MODE_CREATE
       PARAMETER (MPI_MODE_CREATE=1)
       INTEGER MPI_MODE_EXCL
       PARAMETER (MPI_MODE_EXCL=64)
       INTEGER MPI_MODE_APPEND
       PARAMETER (MPI_MODE_APPEND=128)
       INTEGER MPI_MODE_SEQUENTIAL
       PARAMETER (MPI_MODE_SEQUENTIAL=256)
       INTEGER MPI_SEEK_SET
       PARAMETER (MPI_SEEK_SET=600)
       INTEGER MPI_SEEK_CUR
       PARAMETER (MPI_SEEK_CUR=602)
       INTEGER MPI_SEEK_END
       PARAMETER (MPI_SEEK_END=604)
       INTEGER MPI_ORDER_C
       PARAMETER (MPI_ORDER_C=56)
       INTEGER MPI_ORDER_FORTRAN
       PARAMETER (MPI_ORDER_FORTRAN=57)
       INTEGER MPI_DISTRIBUTE_BLOCK
       PARAMETER (MPI_DISTRIBUTE_BLOCK=121)
       INTEGER MPI_DISTRIBUTE_CYCLIC
       PARAMETER (MPI_DISTRIBUTE_CYCLIC=122)
       INTEGER MPI_DISTRIBUTE_NONE
       PARAMETER (MPI_DISTRIBUTE_NONE=123)
       INTEGER MPI_DISTRIBUTE_DFLT_DARG
       PARAMETER (MPI_DISTRIBUTE_DFLT_DARG=-49767)
       INTEGER (KIND=8) MPI_DISPLACEMENT_CURRENT
       PARAMETER (MPI_DISPLACEMENT_CURRENT=-54278278)
       INTEGER MPI_BOTTOM, MPI_IN_PLACE
       INTEGER MPI_UNWEIGHTED, MPI_WEIGHTS_EMPTY

       COMMON /MPIPRIV1/ MPI_BOTTOM, MPI_IN_PLACE, MPI_STATUS_IGNORE

       COMMON /MPIPRIV2/ MPI_STATUSES_IGNORE, MPI_ERRCODES_IGNORE
!DEC$ ATTRIBUTES DLLIMPORT :: /MPIPRIV1/, /MPIPRIV2/

       COMMON /MPIFCMB5/ MPI_UNWEIGHTED
       COMMON /MPIFCMB9/ MPI_WEIGHTS_EMPTY
!DEC$ ATTRIBUTES DLLIMPORT :: /MPIFCMB5/, /MPIFCMB9/

       COMMON /MPIPRIVC/ MPI_ARGVS_NULL, MPI_ARGV_NULL
!DEC$ ATTRIBUTES DLLIMPORT :: /MPIPRIVC/

       END MODULE MPI_CONSTANTS
```

```
      MODULE MPI_BASE
      IMPLICIT NONE
      INTERFACE
      SUBROUTINE
MPI_TYPE_CREATE_DARRAY(v0,v1,v2,v3,v4,v5,v6,v7,v8,v9,ierror)
      INTEGER v0, v1, v2, v3(*), v4(*), v5(*), v6(*), v7, v8, v9
      INTEGER ierror
      END SUBROUTINE MPI_TYPE_CREATE_DARRAY

      SUBROUTINE MPI_COMM_FREE_KEYVAL(v0,ierror)
      INTEGER v0
      INTEGER ierror
      END SUBROUTINE MPI_COMM_FREE_KEYVAL

      SUBROUTINE MPI_TYPE_EXTENT(v0,v1,ierror)
      USE MPI_CONSTANTS,ONLY:MPI_ADDRESS_KIND
      INTEGER v0
      INTEGER(KIND=MPI_ADDRESS_KIND) v1
      INTEGER ierror
      END SUBROUTINE MPI_TYPE_EXTENT

      SUBROUTINE MPI_TYPE_GET_NAME(v0,v1,v2,ierror)
      INTEGER v0
      CHARACTER (LEN=*) v1
      INTEGER v2
      INTEGER ierror
      END SUBROUTINE MPI_TYPE_GET_NAME

      SUBROUTINE MPI_GROUP_INTERSECTION(v0,v1,v2,ierror)
      INTEGER v0, v1, v2
      INTEGER ierror
      END SUBROUTINE MPI_GROUP_INTERSECTION

      SUBROUTINE MPI_WIN_LOCK(v0,v1,v2,v3,ierror)
      INTEGER v0, v1, v2, v3
      INTEGER ierror
      END SUBROUTINE MPI_WIN_LOCK

      SUBROUTINE MPI_CARTDIM_GET(v0,v1,ierror)
      INTEGER v0, v1
      INTEGER ierror
      END SUBROUTINE MPI_CARTDIM_GET

      SUBROUTINE MPI_WIN_GET_ERRHANDLER(v0,v1,ierror)
      INTEGER v0, v1
      INTEGER ierror
      END SUBROUTINE MPI_WIN_GET_ERRHANDLER

      SUBROUTINE MPI_COMM_SPLIT(v0,v1,v2,v3,ierror)
      INTEGER v0, v1, v2, v3
      INTEGER ierror
      END SUBROUTINE MPI_COMM_SPLIT
```

```
SUBROUTINE MPI_COMM_SPLIT_TYPE(v0,v1,v2,v3,v4,ierror)
INTEGER v0, v1, v2, v3, v4
INTEGER ierror
END SUBROUTINE MPI_COMM_SPLIT_TYPE

SUBROUTINE MPI_CANCEL(v0,ierror)
INTEGER v0
INTEGER ierror
END SUBROUTINE MPI_CANCEL

SUBROUTINE MPI_WIN_POST(v0,v1,v2,ierror)
INTEGER v0, v1, v2
INTEGER ierror
END SUBROUTINE MPI_WIN_POST

SUBROUTINE MPI_WIN_COMPLETE(v0,ierror)
INTEGER v0
INTEGER ierror
END SUBROUTINE MPI_WIN_COMPLETE

SUBROUTINE MPI_TEST_CANCELLED(v0,v1,ierror)
USE MPI_CONSTANTS,ONLY:MPI_STATUS_SIZE
INTEGER v0(MPI_STATUS_SIZE)
LOGICAL v1
INTEGER ierror
END SUBROUTINE MPI_TEST_CANCELLED

SUBROUTINE MPI_GROUP_SIZE(v0,v1,ierror)
INTEGER v0, v1
INTEGER ierror
END SUBROUTINE MPI_GROUP_SIZE

SUBROUTINE MPI_ADD_ERROR_STRING(v0,v1,ierror)
INTEGER v0
CHARACTER (LEN=*) v1
INTEGER ierror
END SUBROUTINE MPI_ADD_ERROR_STRING

SUBROUTINE MPI_PACK_SIZE(v0,v1,v2,v3,ierror)
INTEGER v0, v1, v2, v3
INTEGER ierror
END SUBROUTINE MPI_PACK_SIZE

SUBROUTINE MPI_GET_ELEMENTS(v0,v1,v2,ierror)
USE MPI_CONSTANTS,ONLY:MPI_STATUS_SIZE
INTEGER v0(MPI_STATUS_SIZE), v1, v2
INTEGER ierror
END SUBROUTINE MPI_GET_ELEMENTS

SUBROUTINE MPI_GET_ELEMENTS_X(v0,v1,v2,ierror)
USE MPI_CONSTANTS,ONLY:MPI_STATUS_SIZE,MPI_COUNT_KIND
INTEGER v0(MPI_STATUS_SIZE), v1
INTEGER(KIND=MPI_COUNT_KIND) v2
INTEGER ierror
```

```
END SUBROUTINE MPI_GET_ELEMENTS_X

SUBROUTINE MPI_ERRHANDLER_GET(v0,v1,ierror)
INTEGER v0, v1
INTEGER ierror
END SUBROUTINE MPI_ERRHANDLER_GET

SUBROUTINE MPI_FILE_GET_ERRHANDLER(v0,v1,ierror)
INTEGER v0, v1
INTEGER ierror
END SUBROUTINE MPI_FILE_GET_ERRHANDLER

SUBROUTINE MPI_TYPE_LB(v0,v1,ierror)
USE MPI_CONSTANTS,ONLY:MPI_ADDRESS_KIND
INTEGER v0
INTEGER(KIND=MPI_ADDRESS_KIND) v1
INTEGER ierror
END SUBROUTINE MPI_TYPE_LB

SUBROUTINE MPI_REQUEST_FREE(v0,ierror)
INTEGER v0
INTEGER ierror
END SUBROUTINE MPI_REQUEST_FREE

SUBROUTINE MPI_GROUP_RANGE_INCL(v0,v1,v2,v3,ierror)
INTEGER v0, v1, v2(3,*), v3
INTEGER ierror
END SUBROUTINE MPI_GROUP_RANGE_INCL

SUBROUTINE MPI_TYPE_GET_TRUE_EXTENT(v0,v1,v2,ierror)
USE MPI_CONSTANTS,ONLY:MPI_ADDRESS_KIND
INTEGER v0
INTEGER(KIND=MPI_ADDRESS_KIND) v1, v2
INTEGER ierror
END SUBROUTINE MPI_TYPE_GET_TRUE_EXTENT

SUBROUTINE MPI_TYPE_GET_TRUE_EXTENT_X(v0,v1,v2,ierror)
USE MPI_CONSTANTS,ONLY:MPI_COUNT_KIND
INTEGER v0
INTEGER(KIND=MPI_COUNT_KIND) v1, v2
INTEGER ierror
END SUBROUTINE MPI_TYPE_GET_TRUE_EXTENT_X

SUBROUTINE MPI_BARRIER(v0,ierror)
INTEGER v0
INTEGER ierror
END SUBROUTINE MPI_BARRIER

SUBROUTINE MPI_IS_THREAD_MAIN(v0,ierror)
LOGICAL v0
INTEGER ierror
END SUBROUTINE MPI_IS_THREAD_MAIN

SUBROUTINE MPI_WIN_FREE_KEYVAL(v0,ierror)
```

```
      INTEGER v0
      INTEGER ierror
      END SUBROUTINE MPI_WIN_FREE_KEYVAL

      SUBROUTINE MPI_TYPE_COMMIT(v0,ierror)
      INTEGER v0
      INTEGER ierror
      END SUBROUTINE MPI_TYPE_COMMIT

      SUBROUTINE MPI_GROUP_RANGE_EXCL(v0,v1,v2,v3,ierror)
      INTEGER v0, v1, v2(3,*), v3
      INTEGER ierror
      END SUBROUTINE MPI_GROUP_RANGE_EXCL

      SUBROUTINE MPI_REQUEST_GET_STATUS(v0,v1,v2,ierror)
      USE MPI_CONSTANTS,ONLY:MPI_STATUS_SIZE
      INTEGER v0
      LOGICAL v1
      INTEGER v2(MPI_STATUS_SIZE)
      INTEGER ierror
      END SUBROUTINE MPI_REQUEST_GET_STATUS

      SUBROUTINE MPI_QUERY_THREAD(v0,ierror)
      INTEGER v0
      INTEGER ierror
      END SUBROUTINE MPI_QUERY_THREAD

      SUBROUTINE MPI_ERRHANDLER_CREATE(v0,v1,ierror)
      INTERFACE
      SUBROUTINE v0(vv0,vv1)
      INTEGER vv0,vv1
      END SUBROUTINE
      END INTERFACE
      INTEGER v1
      INTEGER ierror
      END SUBROUTINE MPI_ERRHANDLER_CREATE

      SUBROUTINE
MPI_COMM_SPAWN_MULTIPLE(v0,v1,v2,v3,v4,v5,v6,v7,v8,ierror)
      INTEGER v0
      CHARACTER (LEN=*) v1(*), v2(v0,*)
      INTEGER v3(*), v4(*), v5, v6, v7, v8(*)
      INTEGER ierror
      END SUBROUTINE MPI_COMM_SPAWN_MULTIPLE

      SUBROUTINE MPI_COMM_REMOTE_GROUP(v0,v1,ierror)
      INTEGER v0, v1
      INTEGER ierror
      END SUBROUTINE MPI_COMM_REMOTE_GROUP

      SUBROUTINE MPI_TYPE_GET_EXTENT(v0,v1,v2,ierror)
      USE MPI_CONSTANTS,ONLY:MPI_ADDRESS_KIND
      INTEGER v0
      INTEGER(KIND=MPI_ADDRESS_KIND) v1, v2
```

```
INTEGER ierror
END SUBROUTINE MPI_TYPE_GET_EXTENT

SUBROUTINE MPI_TYPE_GET_EXTENT_X(v0,v1,v2,ierror)
USE MPI_CONSTANTS,ONLY:MPI_COUNT_KIND
INTEGER v0
INTEGER(KIND=MPI_COUNT_KIND) v1, v2
INTEGER ierror
END SUBROUTINE MPI_TYPE_GET_EXTENT_X

SUBROUTINE MPI_COMM_COMPARE(v0,v1,v2,ierror)
INTEGER v0, v1, v2
INTEGER ierror
END SUBROUTINE MPI_COMM_COMPARE

SUBROUTINE MPI_INFO_GET_VALUELEN(v0,v1,v2,v3,ierror)
INTEGER v0
CHARACTER (LEN=*) v1
INTEGER v2
LOGICAL v3
INTEGER ierror
END SUBROUTINE MPI_INFO_GET_VALUELEN

SUBROUTINE MPI_INFO_GET(v0,v1,v2,v3,v4,ierror)
INTEGER v0
CHARACTER (LEN=*) v1
INTEGER v2
CHARACTER (LEN=*) v3
LOGICAL v4
INTEGER ierror
END SUBROUTINE MPI_INFO_GET

SUBROUTINE MPI_OP_COMMUTATIVE(v0,v1,ierror)
INTEGER v0
LOGICAL v1
INTEGER ierror
END SUBROUTINE MPI_OP_COMMUTATIVE

SUBROUTINE MPI_OP_CREATE(v0,v1,v2,ierror)
EXTERNAL v0
LOGICAL v1
INTEGER v2
INTEGER ierror
END SUBROUTINE MPI_OP_CREATE

SUBROUTINE MPI_TYPE_CREATE_STRUCT(v0,v1,v2,v3,v4,ierror)
USE MPI_CONSTANTS,ONLY:MPI_ADDRESS_KIND
INTEGER v0, v1(*)
INTEGER(KIND=MPI_ADDRESS_KIND) v2(*)
INTEGER v3(*), v4
INTEGER ierror
END SUBROUTINE MPI_TYPE_CREATE_STRUCT

SUBROUTINE MPI_TYPE_VECTOR(v0,v1,v2,v3,v4,ierror)
```

```
INTEGER v0, v1, v2, v3, v4
INTEGER ierror
END SUBROUTINE MPI_TYPE_VECTOR

SUBROUTINE MPI_WIN_GET_GROUP(v0,v1,ierror)
INTEGER v0, v1
INTEGER ierror
END SUBROUTINE MPI_WIN_GET_GROUP

SUBROUTINE MPI_GROUP_COMPARE(v0,v1,v2,ierror)
INTEGER v0, v1, v2
INTEGER ierror
END SUBROUTINE MPI_GROUP_COMPARE

SUBROUTINE MPI_CART_SHIFT(v0,v1,v2,v3,v4,ierror)
INTEGER v0, v1, v2, v3, v4
INTEGER ierror
END SUBROUTINE MPI_CART_SHIFT

SUBROUTINE MPI_WIN_SET_ERRHANDLER(v0,v1,ierror)
INTEGER v0, v1
INTEGER ierror
END SUBROUTINE MPI_WIN_SET_ERRHANDLER

SUBROUTINE MPI_COMM_SPAWN(v0,v1,v2,v3,v4,v5,v6,v7,ierror)
CHARACTER (LEN=*) v0, v1(*)
INTEGER v2, v3, v4, v5, v6, v7(*)
INTEGER ierror
END SUBROUTINE MPI_COMM_SPAWN

SUBROUTINE MPI_COMM_GROUP(v0,v1,ierror)
INTEGER v0, v1
INTEGER ierror
END SUBROUTINE MPI_COMM_GROUP

SUBROUTINE MPI_WIN_CALL_ERRHANDLER(v0,v1,ierror)
INTEGER v0, v1
INTEGER ierror
END SUBROUTINE MPI_WIN_CALL_ERRHANDLER

SUBROUTINE MPI_LOOKUP_NAME(v0,v1,v2,ierror)
CHARACTER (LEN=*) v0
INTEGER v1
CHARACTER (LEN=*) v2
INTEGER ierror
END SUBROUTINE MPI_LOOKUP_NAME

SUBROUTINE MPI_INFO_FREE(v0,ierror)
INTEGER v0
INTEGER ierror
END SUBROUTINE MPI_INFO_FREE

SUBROUTINE MPI_COMM_SET_ERRHANDLER(v0,v1,ierror)
INTEGER v0, v1
```

```
INTEGER ierror
END SUBROUTINE MPI_COMM_SET_ERRHANDLER

SUBROUTINE MPI_GRAPH_GET(v0,v1,v2,v3,v4,ierror)
INTEGER v0, v1, v2, v3(*), v4(*)
INTEGER ierror
END SUBROUTINE MPI_GRAPH_GET

SUBROUTINE MPI_GROUP_FREE(v0,ierror)
INTEGER v0
INTEGER ierror
END SUBROUTINE MPI_GROUP_FREE

SUBROUTINE MPI_STATUS_SET_ELEMENTS(v0,v1,v2,ierror)
USE MPI_CONSTANTS,ONLY:MPI_STATUS_SIZE
INTEGER v0(MPI_STATUS_SIZE), v1, v2
INTEGER ierror
END SUBROUTINE MPI_STATUS_SET_ELEMENTS

SUBROUTINE MPI_STATUS_SET_ELEMENTS_X(v0,v1,v2,ierror)
USE MPI_CONSTANTS,ONLY:MPI_STATUS_SIZE,MPI_COUNT_KIND
INTEGER v0(MPI_STATUS_SIZE), v1
INTEGER(KIND=MPI_COUNT_KIND) v2
INTEGER ierror
END SUBROUTINE MPI_STATUS_SET_ELEMENTS_X

SUBROUTINE MPI_WIN_TEST(v0,v1,ierror)
INTEGER v0
LOGICAL v1
INTEGER ierror
END SUBROUTINE MPI_WIN_TEST

SUBROUTINE MPI_WIN_FREE(v0,ierror)
INTEGER v0
INTEGER ierror
END SUBROUTINE MPI_WIN_FREE

SUBROUTINE MPI_GRAPH_MAP(v0,v1,v2,v3,v4,ierror)
INTEGER v0, v1, v2(*), v3(*), v4
INTEGER ierror
END SUBROUTINE MPI_GRAPH_MAP

SUBROUTINE MPI_DIST_GRAPH_NEIGHBORS_COUNT(v0,v1,v2,v3,ierror)
INTEGER v0, v1, v2
LOGICAL v3
INTEGER ierror
END SUBROUTINE MPI_DIST_GRAPH_NEIGHBORS_COUNT

SUBROUTINE MPI_PACK_EXTERNAL_SIZE(v0,v1,v2,v3,ierror)
USE MPI_CONSTANTS,ONLY:MPI_ADDRESS_KIND
CHARACTER (LEN=*) v0
INTEGER v1, v2
INTEGER(KIND=MPI_ADDRESS_KIND) v3
INTEGER ierror
```

```
END SUBROUTINE MPI_PACK_EXTERNAL_SIZE

SUBROUTINE MPI_PUBLISH_NAME(v0,v1,v2,ierror)
CHARACTER (LEN=*) v0
INTEGER v1
CHARACTER (LEN=*) v2
INTEGER ierror
END SUBROUTINE MPI_PUBLISH_NAME

SUBROUTINE MPI_TYPE_CREATE_F90_REAL(v0,v1,v2,ierror)
INTEGER v0, v1, v2
INTEGER ierror
END SUBROUTINE MPI_TYPE_CREATE_F90_REAL

SUBROUTINE MPI_OPEN_PORT(v0,v1,ierror)
INTEGER v0
CHARACTER (LEN=*) v1
INTEGER ierror
END SUBROUTINE MPI_OPEN_PORT

SUBROUTINE MPI_GROUP_UNION(v0,v1,v2,ierror)
INTEGER v0, v1, v2
INTEGER ierror
END SUBROUTINE MPI_GROUP_UNION

SUBROUTINE MPI_COMM_ACCEPT(v0,v1,v2,v3,v4,ierror)
CHARACTER (LEN=*) v0
INTEGER v1, v2, v3, v4
INTEGER ierror
END SUBROUTINE MPI_COMM_ACCEPT

SUBROUTINE MPI_FILE_CREATE_ERRHANDLER(v0,v1,ierror)
INTERFACE
SUBROUTINE v0(vv0,vv1)
INTEGER vv0,vv1
END SUBROUTINE
END INTERFACE
INTEGER v1
INTEGER ierror
END SUBROUTINE MPI_FILE_CREATE_ERRHANDLER

SUBROUTINE MPI_WIN_GET_NAME(v0,v1,v2,ierror)
INTEGER v0
CHARACTER (LEN=*) v1
INTEGER v2
INTEGER ierror
END SUBROUTINE MPI_WIN_GET_NAME

SUBROUTINE MPI_INFO_CREATE(v0,ierror)
INTEGER v0
INTEGER ierror
END SUBROUTINE MPI_INFO_CREATE

SUBROUTINE MPI_TYPE_CREATE_F90_INTEGER(v0,v1,ierror)
```

```
INTEGER v0, v1
INTEGER ierror
END SUBROUTINE MPI_TYPE_CREATE_F90_INTEGER

SUBROUTINE MPI_TYPE_SET_NAME(v0,v1,ierror)
INTEGER v0
CHARACTER (LEN=*) v1
INTEGER ierror
END SUBROUTINE MPI_TYPE_SET_NAME

SUBROUTINE MPI_ATTR_DELETE(v0,v1,ierror)
INTEGER v0, v1
INTEGER ierror
END SUBROUTINE MPI_ATTR_DELETE

SUBROUTINE MPI_GROUP_INCL(v0,v1,v2,v3,ierror)
INTEGER v0, v1, v2(*), v3
INTEGER ierror
END SUBROUTINE MPI_GROUP_INCL

SUBROUTINE MPI_COMM_CREATE_ERRHANDLER(v0,v1,ierror)
INTERFACE
SUBROUTINE v0(vv0,vv1)
INTEGER vv0,vv1
END SUBROUTINE
END INTERFACE
INTEGER v1
INTEGER ierror
END SUBROUTINE MPI_COMM_CREATE_ERRHANDLER

SUBROUTINE MPI_COMM_CONNECT(v0,v1,v2,v3,v4,ierror)
CHARACTER (LEN=*) v0
INTEGER v1, v2, v3, v4
INTEGER ierror
END SUBROUTINE MPI_COMM_CONNECT

SUBROUTINE MPI_ERROR_STRING(v0,v1,v2,ierror)
INTEGER v0
CHARACTER (LEN=*) v1
INTEGER v2
INTEGER ierror
END SUBROUTINE MPI_ERROR_STRING

SUBROUTINE MPI_TYPE_GET_CONTENTS(v0,v1,v2,v3,v4,v5,v6,ierror)
USE MPI_CONSTANTS,ONLY:MPI_ADDRESS_KIND
INTEGER v0, v1, v2, v3, v4(*)
INTEGER(KIND=MPI_ADDRESS_KIND) v5(*)
INTEGER v6(*)
INTEGER ierror
END SUBROUTINE MPI_TYPE_GET_CONTENTS

SUBROUTINE MPI_TYPE_STRUCT(v0,v1,v2,v3,v4,ierror)
INTEGER v0, v1(*), v2(*), v3(*), v4
INTEGER ierror
```

```
END SUBROUTINE MPI_TYPE_STRUCT

SUBROUTINE MPI_TYPE_CREATE_INDEXED_BLOCK(v0,v1,v2,v3,v4,ierror)
INTEGER v0, v1, v2(*), v3, v4
INTEGER ierror
END SUBROUTINE MPI_TYPE_CREATE_INDEXED_BLOCK

SUBROUTINE MPI_TYPE_CREATE_HVECTOR(v0,v1,v2,v3,v4,ierror)
USE MPI_CONSTANTS,ONLY:MPI_ADDRESS_KIND
INTEGER v0, v1
INTEGER(KIND=MPI_ADDRESS_KIND) v2
INTEGER v3, v4
INTEGER ierror
END SUBROUTINE MPI_TYPE_CREATE_HVECTOR

SUBROUTINE MPI_TYPE_FREE_KEYVAL(v0,ierror)
INTEGER v0
INTEGER ierror
END SUBROUTINE MPI_TYPE_FREE_KEYVAL

SUBROUTINE MPI_START(v0,ierror)
INTEGER v0
INTEGER ierror
END SUBROUTINE MPI_START

SUBROUTINE MPI_ABORT(v0,v1,ierror)
INTEGER v0, v1
INTEGER ierror
END SUBROUTINE MPI_ABORT

SUBROUTINE MPI_INTERCOMM_CREATE(v0,v1,v2,v3,v4,v5,ierror)
INTEGER v0, v1, v2, v3, v4, v5
INTEGER ierror
END SUBROUTINE MPI_INTERCOMM_CREATE

SUBROUTINE MPI_COMM_RANK(v0,v1,ierror)
INTEGER v0, v1
INTEGER ierror
END SUBROUTINE MPI_COMM_RANK

SUBROUTINE MPI_COMM_GET_PARENT(v0,ierror)
INTEGER v0
INTEGER ierror
END SUBROUTINE MPI_COMM_GET_PARENT

SUBROUTINE MPI_FINALIZED(v0,ierror)
LOGICAL v0
INTEGER ierror
END SUBROUTINE MPI_FINALIZED

SUBROUTINE MPI_INTERCOMM_MERGE(v0,v1,v2,ierror)
INTEGER v0
LOGICAL v1
INTEGER v2
```

```
INTEGER ierror
END SUBROUTINE MPI_INTERCOMM_MERGE

SUBROUTINE MPI_INFO_GET_NTHKEY(v0,v1,v2,ierror)
INTEGER v0, v1
CHARACTER (LEN=*) v2
INTEGER ierror
END SUBROUTINE MPI_INFO_GET_NTHKEY

SUBROUTINE MPI_TYPE_MATCH_SIZE(v0,v1,v2,ierror)
INTEGER v0, v1, v2
INTEGER ierror
END SUBROUTINE MPI_TYPE_MATCH_SIZE

SUBROUTINE MPI_STATUS_SET_CANCELLED(v0,v1,ierror)
USE MPI_CONSTANTS,ONLY:MPI_STATUS_SIZE
INTEGER v0(MPI_STATUS_SIZE), v1
INTEGER ierror
END SUBROUTINE MPI_STATUS_SET_CANCELLED

SUBROUTINE MPI_FILE_SET_ERRHANDLER(v0,v1,ierror)
INTEGER v0, v1
INTEGER ierror
END SUBROUTINE MPI_FILE_SET_ERRHANDLER

SUBROUTINE MPI_INFO_DELETE(v0,v1,ierror)
INTEGER v0
CHARACTER (LEN=*) v1
INTEGER ierror
END SUBROUTINE MPI_INFO_DELETE

SUBROUTINE MPI_UNPUBLISH_NAME(v0,v1,v2,ierror)
CHARACTER (LEN=*) v0
INTEGER v1
CHARACTER (LEN=*) v2
INTEGER ierror
END SUBROUTINE MPI_UNPUBLISH_NAME

SUBROUTINE MPI_TYPE_CONTIGUOUS(v0,v1,v2,ierror)
INTEGER v0, v1, v2
INTEGER ierror
END SUBROUTINE MPI_TYPE_CONTIGUOUS

SUBROUTINE MPI_INITIALIZED(v0,ierror)
LOGICAL v0
INTEGER ierror
END SUBROUTINE MPI_INITIALIZED

SUBROUTINE MPI_TYPE_CREATE_RESIZED(v0,v1,v2,v3,ierror)
USE MPI_CONSTANTS,ONLY:MPI_ADDRESS_KIND
INTEGER v0
INTEGER(KIND=MPI_ADDRESS_KIND) v1, v2
INTEGER v3
INTEGER ierror
```

```
END SUBROUTINE MPI_TYPE_CREATE_RESIZED

SUBROUTINE MPI_TYPE_UB(v0,v1,ierror)
USE MPI_CONSTANTS,ONLY:MPI_ADDRESS_KIND
INTEGER v0
INTEGER(KIND=MPI_ADDRESS_KIND) v1
INTEGER ierror
END SUBROUTINE MPI_TYPE_UB

SUBROUTINE MPI_INFO_DUP(v0,v1,ierror)
INTEGER v0, v1
INTEGER ierror
END SUBROUTINE MPI_INFO_DUP

SUBROUTINE MPI_TYPE_DUP(v0,v1,ierror)
INTEGER v0, v1
INTEGER ierror
END SUBROUTINE MPI_TYPE_DUP

SUBROUTINE MPI_ERRHANDLER_SET(v0,v1,ierror)
INTEGER v0, v1
INTEGER ierror
END SUBROUTINE MPI_ERRHANDLER_SET

SUBROUTINE MPI_WIN_DELETE_ATTR(v0,v1,ierror)
INTEGER v0, v1
INTEGER ierror
END SUBROUTINE MPI_WIN_DELETE_ATTR

SUBROUTINE MPI_INFO_GET_NKEYS(v0,v1,ierror)
INTEGER v0, v1
INTEGER ierror
END SUBROUTINE MPI_INFO_GET_NKEYS

SUBROUTINE MPI_GROUP_EXCL(v0,v1,v2,v3,ierror)
INTEGER v0, v1, v2(*), v3
INTEGER ierror
END SUBROUTINE MPI_GROUP_EXCL

SUBROUTINE MPI_INFO_SET(v0,v1,v2,ierror)
INTEGER v0
CHARACTER (LEN=*) v1, v2
INTEGER ierror
END SUBROUTINE MPI_INFO_SET

SUBROUTINE MPI_WAIT(v0,v1,ierror)
USE MPI_CONSTANTS,ONLY:MPI_STATUS_SIZE
INTEGER v0, v1(MPI_STATUS_SIZE)
INTEGER ierror
END SUBROUTINE MPI_WAIT

SUBROUTINE MPI_COMM_DELETE_ATTR(v0,v1,ierror)
INTEGER v0, v1
INTEGER ierror
```

```
END SUBROUTINE MPI_COMM_DELETE_ATTR

SUBROUTINE MPI_COMM_GET_NAME(v0,v1,v2,ierror)
INTEGER v0
CHARACTER (LEN=*) v1
INTEGER v2
INTEGER ierror
END SUBROUTINE MPI_COMM_GET_NAME

SUBROUTINE MPI_TEST(v0,v1,v2,ierror)
USE MPI_CONSTANTS,ONLY:MPI_STATUS_SIZE
INTEGER v0
LOGICAL v1
INTEGER v2(MPI_STATUS_SIZE)
INTEGER ierror
END SUBROUTINE MPI_TEST

SUBROUTINE MPI_GET_COUNT(v0,v1,v2,ierror)
USE MPI_CONSTANTS,ONLY:MPI_STATUS_SIZE
INTEGER v0(MPI_STATUS_SIZE), v1, v2
INTEGER ierror
END SUBROUTINE MPI_GET_COUNT

SUBROUTINE MPI_ADD_ERROR_CLASS(v0,ierror)
INTEGER v0
INTEGER ierror
END SUBROUTINE MPI_ADD_ERROR_CLASS

SUBROUTINE MPI_COMM_FREE(v0,ierror)
INTEGER v0
INTEGER ierror
END SUBROUTINE MPI_COMM_FREE

SUBROUTINE MPI_COMM_SET_NAME(v0,v1,ierror)
INTEGER v0
CHARACTER (LEN=*) v1
INTEGER ierror
END SUBROUTINE MPI_COMM_SET_NAME

SUBROUTINE MPI_COMM_DISCONNECT(v0,ierror)
INTEGER v0
INTEGER ierror
END SUBROUTINE MPI_COMM_DISCONNECT

SUBROUTINE MPI_IPROBE(v0,v1,v2,v3,v4,ierror)
USE MPI_CONSTANTS,ONLY:MPI_STATUS_SIZE
INTEGER v0, v1, v2
LOGICAL v3
INTEGER v4(MPI_STATUS_SIZE)
INTEGER ierror
END SUBROUTINE MPI_IPROBE

SUBROUTINE MPI_IMPROBE(v0,v1,v2,v3,v4,v5,ierror)
USE MPI_CONSTANTS,ONLY:MPI_STATUS_SIZE
```

```
INTEGER v0, v1, v2, v3, v4
INTEGER v5(MPI_STATUS_SIZE)
INTEGER ierror
END SUBROUTINE MPI_IMPROBE

SUBROUTINE MPI_MPROBE(v0,v1,v2,v3,v4,ierror)
USE MPI_CONSTANTS,ONLY:MPI_STATUS_SIZE
INTEGER v0, v1, v2, v3
INTEGER v4(MPI_STATUS_SIZE)
INTEGER ierror
END SUBROUTINE MPI_MPROBE

SUBROUTINE MPI_ADD_ERROR_CODE(v0,v1,ierror)
INTEGER v0, v1
INTEGER ierror
END SUBROUTINE MPI_ADD_ERROR_CODE

SUBROUTINE MPI_COMM_GET_ERRHANDLER(v0,v1,ierror)
INTEGER v0, v1
INTEGER ierror
END SUBROUTINE MPI_COMM_GET_ERRHANDLER

SUBROUTINE MPI_COMM_CREATE(v0,v1,v2,ierror)
INTEGER v0, v1, v2
INTEGER ierror
END SUBROUTINE MPI_COMM_CREATE

SUBROUTINE MPI_OP_FREE(v0,ierror)
INTEGER v0
INTEGER ierror
END SUBROUTINE MPI_OP_FREE

SUBROUTINE MPI_TOPO_TEST(v0,v1,ierror)
INTEGER v0, v1
INTEGER ierror
END SUBROUTINE MPI_TOPO_TEST

SUBROUTINE MPI_GET_PROCESSOR_NAME(v0,v1,ierror)
CHARACTER (LEN=*) v0
INTEGER v1
INTEGER ierror
END SUBROUTINE MPI_GET_PROCESSOR_NAME

SUBROUTINE MPI_COMM_SIZE(v0,v1,ierror)
INTEGER v0, v1
INTEGER ierror
END SUBROUTINE MPI_COMM_SIZE

SUBROUTINE MPI_WIN_UNLOCK(v0,v1,ierror)
INTEGER v0, v1
INTEGER ierror
END SUBROUTINE MPI_WIN_UNLOCK

SUBROUTINE MPI_WIN_FLUSH(v0,v1,ierror)
```

```
INTEGER v0, v1
INTEGER ierror
END SUBROUTINE MPI_WIN_FLUSH

SUBROUTINE MPI_WIN_FLUSH_LOCAL(v0,v1,ierror)
INTEGER v0, v1
INTEGER ierror
END SUBROUTINE MPI_WIN_FLUSH_LOCAL

SUBROUTINE MPI_ERRHANDLER_FREE(v0,ierror)
INTEGER v0
INTEGER ierror
END SUBROUTINE MPI_ERRHANDLER_FREE

SUBROUTINE MPI_COMM_REMOTE_SIZE(v0,v1,ierror)
INTEGER v0, v1
INTEGER ierror
END SUBROUTINE MPI_COMM_REMOTE_SIZE

SUBROUTINE MPI_PROBE(v0,v1,v2,v3,ierror)
USE MPI_CONSTANTS,ONLY:MPI_STATUS_SIZE
INTEGER v0, v1, v2, v3(MPI_STATUS_SIZE)
INTEGER ierror
END SUBROUTINE MPI_PROBE

SUBROUTINE MPI_TYPE_HINDEXED(v0,v1,v2,v3,v4,ierror)
INTEGER v0, v1(*), v2(*), v3, v4
INTEGER ierror
END SUBROUTINE MPI_TYPE_HINDEXED

SUBROUTINE MPI_WIN_WAIT(v0,ierror)
INTEGER v0
INTEGER ierror
END SUBROUTINE MPI_WIN_WAIT

SUBROUTINE MPI_WIN_SET_NAME(v0,v1,ierror)
INTEGER v0
CHARACTER (LEN=*) v1
INTEGER ierror
END SUBROUTINE MPI_WIN_SET_NAME

SUBROUTINE MPI_TYPE_SIZE(v0,v1,ierror)
INTEGER v0, v1
INTEGER ierror
END SUBROUTINE MPI_TYPE_SIZE

SUBROUTINE MPI_TYPE_SIZE_X(v0,v1,ierror)
USE MPI_CONSTANTS,ONLY:MPI_COUNT_KIND
INTEGER v0
INTEGER(KIND=MPI_COUNT_KIND) v1
INTEGER ierror
END SUBROUTINE MPI_TYPE_SIZE_X

SUBROUTINE MPI_TYPE_CREATE_SUBARRAY(v0,v1,v2,v3,v4,v5,v6,ierror)
```

```
INTEGER v0, v1(*), v2(*), v3(*), v4, v5, v6
INTEGER ierror
END SUBROUTINE MPI_TYPE_CREATE_SUBARRAY

SUBROUTINE MPI_WIN_CREATE_ERRHANDLER(v0,v1,ierror)
INTERFACE
SUBROUTINE v0(vv0,vv1)
INTEGER vv0,vv1
END SUBROUTINE
END INTERFACE
INTEGER v1
INTEGER ierror
END SUBROUTINE MPI_WIN_CREATE_ERRHANDLER

SUBROUTINE MPI_WIN_START(v0,v1,v2,ierror)
INTEGER v0, v1, v2
INTEGER ierror
END SUBROUTINE MPI_WIN_START

SUBROUTINE MPI_TYPE_FREE(v0,ierror)
INTEGER v0
INTEGER ierror
END SUBROUTINE MPI_TYPE_FREE

SUBROUTINE MPI_WIN_FENCE(v0,v1,ierror)
INTEGER v0, v1
INTEGER ierror
END SUBROUTINE MPI_WIN_FENCE

SUBROUTINE MPI_GRAPHDIMS_GET(v0,v1,v2,ierror)
INTEGER v0, v1, v2
INTEGER ierror
END SUBROUTINE MPI_GRAPHDIMS_GET

SUBROUTINE MPI_FILE_CALL_ERRHANDLER(v0,v1,ierror)
INTEGER v0, v1
INTEGER ierror
END SUBROUTINE MPI_FILE_CALL_ERRHANDLER

SUBROUTINE MPI_TYPE_GET_ENVELOPE(v0,v1,v2,v3,v4,ierror)
INTEGER v0, v1, v2, v3, v4
INTEGER ierror
END SUBROUTINE MPI_TYPE_GET_ENVELOPE

SUBROUTINE MPI_TYPE_DELETE_ATTR(v0,v1,ierror)
INTEGER v0, v1
INTEGER ierror
END SUBROUTINE MPI_TYPE_DELETE_ATTR

SUBROUTINE MPI_TYPE_CREATE_HINDEXED(v0,v1,v2,v3,v4,ierror)
USE MPI_CONSTANTS,ONLY:MPI_ADDRESS_KIND
INTEGER v0, v1(*)
INTEGER(KIND=MPI_ADDRESS_KIND) v2(*)
INTEGER v3, v4
```

```
INTEGER ierror
END SUBROUTINE MPI_TYPE_CREATE_HINDEXED

SUBROUTINE MPI_TYPE_CREATE_HINDEXED_BLOCK(v0,v1,v2,v3,v4,ierror)
USE MPI_CONSTANTS,ONLY:MPI_ADDRESS_KIND
INTEGER v0, v1
INTEGER(KIND=MPI_ADDRESS_KIND) v2(*)
INTEGER v3, v4
INTEGER ierror
END SUBROUTINE MPI_TYPE_CREATE_HINDEXED_BLOCK

SUBROUTINE MPI_TYPE_INDEXED(v0,v1,v2,v3,v4,ierror)
INTEGER v0, v1(*), v2(*), v3, v4
INTEGER ierror
END SUBROUTINE MPI_TYPE_INDEXED

SUBROUTINE MPI_GREQUEST_COMPLETE(v0,ierror)
INTEGER v0
INTEGER ierror
END SUBROUTINE MPI_GREQUEST_COMPLETE

SUBROUTINE MPI_GRAPH_NEIGHBORS_COUNT(v0,v1,v2,ierror)
INTEGER v0, v1, v2
INTEGER ierror
END SUBROUTINE MPI_GRAPH_NEIGHBORS_COUNT

SUBROUTINE MPI_GET_VERSION(v0,v1,ierror)
INTEGER v0, v1
INTEGER ierror
END SUBROUTINE MPI_GET_VERSION

SUBROUTINE MPI_GET_LIBRARY_VERSION(v0,v1,ierror)
CHARACTER (LEN=*) v0
INTEGER v1
INTEGER ierror
END SUBROUTINE MPI_GET_LIBRARY_VERSION

SUBROUTINE MSMPI_GET_BSEND_OVERHEAD(size)
INTEGER size
END SUBROUTINE MSMPI_GET_BSEND_OVERHEAD

SUBROUTINE MSMPI_GET_VERSION(version)
INTEGER version
END SUBROUTINE MSMPI_GET_VERSION

SUBROUTINE MPI_TYPE_HVECTOR(v0,v1,v2,v3,v4,ierror)
USE MPI_CONSTANTS,ONLY:MPI_ADDRESS_KIND
INTEGER v0, v1
INTEGER(KIND=MPI_ADDRESS_KIND) v2
INTEGER v3, v4
INTEGER ierror
END SUBROUTINE MPI_TYPE_HVECTOR

SUBROUTINE MPI_KEYVAL_FREE(v0,ierror)
```

```
INTEGER v0
INTEGER ierror
END SUBROUTINE MPI_KEYVAL_FREE

SUBROUTINE MPI_COMM_CALL_ERRHANDLER(v0,v1,ierror)
INTEGER v0, v1
INTEGER ierror
END SUBROUTINE MPI_COMM_CALL_ERRHANDLER

SUBROUTINE MPI_COMM_JOIN(v0,v1,ierror)
INTEGER v0, v1
INTEGER ierror
END SUBROUTINE MPI_COMM_JOIN

SUBROUTINE MPI_COMM_TEST_INTER(v0,v1,ierror)
INTEGER v0
LOGICAL v1
INTEGER ierror
END SUBROUTINE MPI_COMM_TEST_INTER

SUBROUTINE MPI_CLOSE_PORT(v0,ierror)
CHARACTER (LEN=*) v0
INTEGER ierror
END SUBROUTINE MPI_CLOSE_PORT

SUBROUTINE MPI_TYPE_CREATE_F90_COMPLEX(v0,v1,v2,ierror)
INTEGER v0, v1, v2
INTEGER ierror
END SUBROUTINE MPI_TYPE_CREATE_F90_COMPLEX

SUBROUTINE MPI_GROUP_DIFFERENCE(v0,v1,v2,ierror)
INTEGER v0, v1, v2
INTEGER ierror
END SUBROUTINE MPI_GROUP_DIFFERENCE

SUBROUTINE MPI_COMM_DUP(v0,v1,ierror)
INTEGER v0, v1
INTEGER ierror
END SUBROUTINE MPI_COMM_DUP

SUBROUTINE MPI_ERROR_CLASS(v0,v1,ierror)
INTEGER v0, v1
INTEGER ierror
END SUBROUTINE MPI_ERROR_CLASS

SUBROUTINE MPI_GROUP_RANK(v0,v1,ierror)
INTEGER v0, v1
INTEGER ierror
END SUBROUTINE MPI_GROUP_RANK


 SUBROUTINE MPI_INIT(ierror)
 INTEGER ierror
 END SUBROUTINE MPI_INIT
```

```
SUBROUTINE MPI_INIT_THREAD(v0,v1,ierror)
INTEGER v0, v1, ierror
END SUBROUTINE MPI_INIT_THREAD

FUNCTION MPI_WTIME()
    DOUBLE PRECISION MPI_WTIME
END FUNCTION MPI_WTIME

FUNCTION MPI_WTICK()
    DOUBLE PRECISION MPI_WTICK
END FUNCTION MPI_WTICK

FUNCTION PMPI_WTIME()
    DOUBLE PRECISION PMPI_WTIME
END FUNCTION PMPI_WTIME

FUNCTION PMPI_WTICK()
    DOUBLE PRECISION PMPI_WTICK
END FUNCTION PMPI_WTICK

SUBROUTINE MPI_NULL_DELETE_FN(a,b,c,d,e)
  INTEGER a,b,c,d,e
END SUBROUTINE MPI_NULL_DELETE_FN

SUBROUTINE MPI_DUP_FN(a,b,c,d,e,f,g)
  INTEGER a,b,c,d,e,g
  LOGICAL f
END SUBROUTINE MPI_DUP_FN

SUBROUTINE MPI_NULL_COPY_FN(a,b,c,d,e,f,g)
  INTEGER a,b,c,d,e,g
  LOGICAL f
END SUBROUTINE MPI_NULL_COPY_FN

SUBROUTINE MPI_COMM_NULL_DELETE_FN(a,b,c,d,e)
  USE MPI_CONSTANTS,ONLY: MPI_ADDRESS_KIND
  INTEGER a,b,e
  INTEGER (KIND=MPI_ADDRESS_KIND) c, d
END SUBROUTINE MPI_COMM_NULL_DELETE_FN

SUBROUTINE MPI_COMM_DUP_FN(a,b,c,d,e,f,g)
  USE MPI_CONSTANTS,ONLY: MPI_ADDRESS_KIND
  INTEGER a,b,g
  INTEGER (KIND=MPI_ADDRESS_KIND) c,d,e
  LOGICAL f
END SUBROUTINE MPI_COMM_DUP_FN

SUBROUTINE MPI_COMM_NULL_COPY_FN(a,b,c,d,e,f,g)
  USE MPI_CONSTANTS,ONLY: MPI_ADDRESS_KIND
  INTEGER a,b,g
  INTEGER (KIND=MPI_ADDRESS_KIND) c,d,e
  LOGICAL f
END SUBROUTINE MPI_COMM_NULL_COPY_FN
```

```
 SUBROUTINE MPI_TYPE_NULL_DELETE_FN(a,b,c,d,e)
   USE MPI_CONSTANTS,ONLY: MPI_ADDRESS_KIND
   INTEGER a,b,e
   INTEGER (KIND=MPI_ADDRESS_KIND) c, d
 END SUBROUTINE MPI_TYPE_NULL_DELETE_FN

 SUBROUTINE MPI_TYPE_DUP_FN(a,b,c,d,e,f,g)
   USE MPI_CONSTANTS,ONLY: MPI_ADDRESS_KIND
   INTEGER a,b,g
   INTEGER (KIND=MPI_ADDRESS_KIND) c,d,e
   LOGICAL f
 END SUBROUTINE MPI_TYPE_DUP_FN

 SUBROUTINE MPI_TYPE_NULL_COPY_FN(a,b,c,d,e,f,g)
   USE MPI_CONSTANTS,ONLY: MPI_ADDRESS_KIND
   INTEGER a,b,g
   INTEGER (KIND=MPI_ADDRESS_KIND) c,d,e
   LOGICAL f
 END SUBROUTINE MPI_TYPE_NULL_COPY_FN

 SUBROUTINE MPI_WIN_NULL_DELETE_FN(a,b,c,d,e)
   USE MPI_CONSTANTS,ONLY: MPI_ADDRESS_KIND
   INTEGER a,b,e
   INTEGER (KIND=MPI_ADDRESS_KIND) c, d
 END SUBROUTINE MPI_WIN_NULL_DELETE_FN

 SUBROUTINE MPI_WIN_DUP_FN(a,b,c,d,e,f,g)
   USE MPI_CONSTANTS,ONLY: MPI_ADDRESS_KIND
   INTEGER a,b,g
   INTEGER (KIND=MPI_ADDRESS_KIND) c,d,e
   LOGICAL f
 END SUBROUTINE MPI_WIN_DUP_FN

 SUBROUTINE MPI_WIN_NULL_COPY_FN(a,b,c,d,e,f,g)
   USE MPI_CONSTANTS,ONLY: MPI_ADDRESS_KIND
   INTEGER a,b,g
   INTEGER (KIND=MPI_ADDRESS_KIND) c,d,e
   LOGICAL f
 END SUBROUTINE MPI_WIN_NULL_COPY_FN

END INTERFACE
END MODULE MPI_BASE

MODULE MPI_SIZEOFS
!     This module contains the definitions for MPI_SIZEOF for the
!     predefined, named types in Fortran 90.  This is provided
!     as a separate module to allow MPI_SIZEOF to supply the
!     basic size information even when we do not provide the
!     arbitrary choice types
IMPLICIT NONE

PUBLIC :: MPI_SIZEOF
INTERFACE MPI_SIZEOF
```

```
          MODULE PROCEDURE MPI_SIZEOF_I, MPI_SIZEOF_R,                 &
     &                    MPI_SIZEOF_L, MPI_SIZEOF_CH, MPI_SIZEOF_CX,&
     &           MPI_SIZEOF_IV, MPI_SIZEOF_RV,                        &
     &           MPI_SIZEOF_LV, MPI_SIZEOF_CHV, MPI_SIZEOF_CXV
           MODULE PROCEDURE MPI_SIZEOF_D, MPI_SIZEOF_DV
      END INTERFACE ! MPI_SIZEOF

      CONTAINS

      SUBROUTINE MPI_SIZEOF_I( X, SIZE, IERROR )
      INTEGER X
      INTEGER SIZE, IERROR
      SIZE = 4
      IERROR = 0
      END SUBROUTINE MPI_SIZEOF_I

      SUBROUTINE MPI_SIZEOF_R( X, SIZE, IERROR )
      REAL X
      INTEGER SIZE, IERROR
      SIZE = 4
      IERROR = 0
      END SUBROUTINE MPI_SIZEOF_R

! If reals and doubles have been forced to the same size (e.g., with
! -i8 -r8 to compilers like g95), then the compiler may refuse to
! allow interfaces that use real and double precision (failing to
! determine which one is intended)
       SUBROUTINE MPI_SIZEOF_D( X, SIZE, IERROR )
       DOUBLE PRECISION X
       INTEGER SIZE, IERROR
       SIZE = 8
       IERROR = 0
       END SUBROUTINE MPI_SIZEOF_D

      SUBROUTINE MPI_SIZEOF_L( X, SIZE, IERROR )
      LOGICAL X
      INTEGER SIZE, IERROR
      SIZE = 4
      IERROR = 0
      END SUBROUTINE MPI_SIZEOF_L

      SUBROUTINE MPI_SIZEOF_CH( X, SIZE, IERROR )
      CHARACTER X
      INTEGER SIZE, IERROR
      SIZE = 1
      IERROR = 0
      END SUBROUTINE MPI_SIZEOF_CH

      SUBROUTINE MPI_SIZEOF_CX( X, SIZE, IERROR )
      COMPLEX X
      INTEGER SIZE, IERROR
      SIZE = 2*4
      IERROR = 0
      END SUBROUTINE MPI_SIZEOF_CX
```

```
       SUBROUTINE MPI_SIZEOF_IV( X, SIZE, IERROR )
       INTEGER X(*)
       INTEGER SIZE, IERROR
       SIZE = 4
       IERROR = 0
       END SUBROUTINE MPI_SIZEOF_IV

       SUBROUTINE MPI_SIZEOF_RV( X, SIZE, IERROR )
       REAL X(*)
       INTEGER SIZE, IERROR
       SIZE = 4
       IERROR = 0
       END SUBROUTINE MPI_SIZEOF_RV

! If reals and doubles have been forced to the same size (e.g., with
! -i8 -r8 to compilers like g95), then the compiler may refuse to
! allow interfaces that use real and double precision (failing to
! determine which one is intended)
        SUBROUTINE MPI_SIZEOF_DV( X, SIZE, IERROR )
        DOUBLE PRECISION X(*)
        INTEGER SIZE, IERROR
        SIZE = 8
        IERROR = 0
        END SUBROUTINE MPI_SIZEOF_DV

       SUBROUTINE MPI_SIZEOF_LV( X, SIZE, IERROR )
       LOGICAL X(*)
       INTEGER SIZE, IERROR
       SIZE = 4
       IERROR = 0
       END SUBROUTINE MPI_SIZEOF_LV

       SUBROUTINE MPI_SIZEOF_CHV( X, SIZE, IERROR )
       CHARACTER X(*)
       INTEGER SIZE, IERROR
       SIZE = 1
       IERROR = 0
       END SUBROUTINE MPI_SIZEOF_CHV

       SUBROUTINE MPI_SIZEOF_CXV( X, SIZE, IERROR )
       COMPLEX X(*)
       INTEGER SIZE, IERROR
       SIZE = 2*4
       IERROR = 0
       END SUBROUTINE MPI_SIZEOF_CXV

! We don't include double complex.  If we did, we'd need to include the
! same hack as for real and double above if the compiler has been forced
! to make them the same size.
       END MODULE MPI_SIZEOFS

       MODULE MPI
       USE MPI_CONSTANTS
```

```
USE MPI_SIZEOFS
USE MPI_BASE
END MODULE MPI
```

REFERENCES

1. R. F. Harrington, *Field Computation by Method of Moments* (Macmillan, New York, 1968).

2. E. K. Miller, L. Medgyesi-Mitschang, and E. H. Newman, *Computational Electromagnetics - Frequency Domain Methods* (IEEE Press, New York, 1992).

3. W. C. Gibson, *The Method of Moments in Electromagnetics* (Chapman & Hall, Boca Raton, FL, 2008).

4. T. K. Sarkar, "A note on the choice of weighting functions in the method of moments," *IEEE Transactions on Antennas and Propagation* **33**, 436–441 (1985).

5. T. K. Sarkar, A. R. Djordjevic, and E. Arvas, "A note on the choice of weighting functions in the method of moments," *IEEE Transactions on Antennas and Propagation* **33**, 988–996 (August 1985).

6. J. Song, C. C. Lu, and W. C. Chew, "Multilevel fast multipole algorithm for electromagnetic scattering by large complex objects," *IEEE Transactions on Antennas and Propagation* **45**, 1488–1493 (October 1997).

7. J. Shaeffer, "Direct solve of electrically large integral equations for problem sizes to 1m unknowns," *IEEE Transactions on Antennas and Propagation* **56**, 2306–2313 (1997).

8. S. M. Rao and M. S. Kluskens, "A new power series solution approach to solving electrically large complex electromagnetic scattering problems," *ACES Journal* **31**(9), 1009–1019 (2016).

9. S. M. Rao, D. R. Wilton, and A. W. Glisson, "Electromagnetic scattering by surfaces of arbitrary shape," *IEEE Transactions on Antennas and Propagation* **30**, 409–418 (1982).

10. J. R. Bunch and J. Hopcroft, "Triangular factorization and inversion by fast matrix multiplication," *Mathematics of Computation* **28**, 231–236 (1974).

11. K. Nayanthara, S. M. Rao, and T. K. Sarkar, "Analysis of two dimensional conducting and dielectric bodies utilizing the conjugate gradient method," *IEEE Transactions on Antennas and Propagation* **35**, 451–453 (1987).

12. G. H. Golub and C. F. V. Loan, *Matrix Computations* (Johns Hopkins, Baltimore, 1996).

13. R. F. Harrington, *Time-Harmonic Electromagnetic Fields* (McGraw-Hill, New York, 1961).