University of Kentucky
UKnowledge

Theses and Dissertations--Electrical and Computer Engineering

Electrical and Computer Engineering

2017

HYBRID PARALLELIZATION OF THE NASA GEMINI ELECTROMAGNETIC MODELING TOOL

Buxton L. Johnson Sr.
University of Kentucky, buxton.johnson.sr@uky.edu
Author ORCID Identifier:
http://orcid.org/0000-0002-6417-2333

Digital Object Identifier: https://doi.org/10.13023/ETD.2017.080

Right click to open a feedback form in a new tab to let us know how this document benefits you.

Recommended Citation

Johnson, Buxton L. Sr., "HYBRID PARALLELIZATION OF THE NASA GEMINI ELECTROMAGNETIC MODELING TOOL" (2017). *Theses and Dissertations--Electrical and Computer Engineering*. 99.
https://uknowledge.uky.edu/ece_etds/99

This Master's Thesis is brought to you for free and open access by the Electrical and Computer Engineering at UKnowledge. It has been accepted for inclusion in Theses and Dissertations--Electrical and Computer Engineering by an authorized administrator of UKnowledge. For more information, please contact UKnowledge@lsv.uky.edu.

STUDENT AGREEMENT:

I represent that my thesis or dissertation and abstract are my original work. Proper attribution has been given to all outside sources. I understand that I am solely responsible for obtaining any needed copyright permissions. I have obtained needed written permission statement(s) from the owner(s) of each third-party copyrighted matter to be included in my work, allowing electronic distribution (if such use is not permitted by the fair use doctrine) which will be submitted to UKnowledge as Additional File.

I hereby grant to The University of Kentucky and its agents the irrevocable, non-exclusive, and royalty-free license to archive and make accessible my work in whole or in part in all forms of media, now or hereafter known. I agree that the document mentioned above may be made available immediately for worldwide access unless an embargo applies.

I retain all other ownership rights to the copyright of my work. I also retain the right to use in future works (such as articles or books) all or part of my work. I understand that I am free to register the copyright to my work.

REVIEW, APPROVAL AND ACCEPTANCE

The document mentioned above has been reviewed and accepted by the student’s advisor, on behalf of the advisory committee, and by the Director of Graduate Studies (DGS), on behalf of the program; we verify that this is the final, approved version of the student’s thesis including all changes required by the advisory committee. The undersigned agree to abide by the statements above.

Buxton L. Johnson Sr., Student

Dr. Robert J. Adams, Major Professor

Dr. Cai-Cheng Lu, Director of Graduate Studies

HYBRID PARALLELIZATION OF THE NASA GEMINI ELECTROMAGNETIC MODELING TOOL

THESIS

A thesis submitted in partial fulfillment of the requirements for the
degree of Master of Science in Electrical Engineering in the
College of Engineering
at the University of Kentucky

By

Buxton L. Johnson, Sr.

Lexington, Kentucky

Director: Dr. Robert J. Adams, Professor of Electrical and Computer Engineering

Lexington, Kentucky

2017

Copyright © Buxton L. Johnson, Sr. 2017

ABSTRACT OF THESIS

HYBRID PARALLELIZATION OF THE NASA GEMINI ELECTROMAGNETIC MODELING TOOL

Understanding, predicting, and controlling electromagnetic field interactions on and between complex RF platforms requires high fidelity computational electromagnetic (CEM) simulation. The primary CEM tool within NASA is GEMINI, an integral equation based method-of-moments (MoM) code for frequency domain electromagnetic modeling. However, GEMINI is currently limited in the size and complexity of problems that can be effectively handled. To extend GEMINI'S CEM capabilities beyond those currently available, primary research is devoted to integrating the MFDlib library developed at the University of Kentucky with GEMINI for efficient filling, factorization, and solution of large electromagnetic problems formulated using integral equation methods. A secondary research project involves the hybrid parallelization of GEMINI for the efficient speedup of the impedance matrix filling process. This thesis discusses the research, development, and testing of the secondary research project on the High Performance Computing DLX Linux supercomputer cluster. Initial testing of GEMINI's existing MPI parallelization establishes the benchmark for speedup and reveals performance issues subsequently solved by the NASA CEM Lab. Implementation of hybrid parallelization incorporates GEMINI's existing course level MPI parallelization with Open MP fine level parallel threading. Simple and nested Open MP threading are compared. Final testing documents the improvements realized by hybrid parallelization.

KEYWORDS: computational electromagnetics, method of moments, electric field integral equation, hybrid parallelization, high performance computing.

Buxton L. Johnson, Sr.

April 10, 2017

HYBRID PARALLELIZATION OF THE NASA GEMINI ELECTROMAGNETIC MODELING TOOL

By

Buxton L. Johnson, Sr.

Robert J. Adams, Ph.D.
Director of Thesis

Cai-Cheng Lu, Ph.D.
Director of Graduate Studies

April 10, 2017

Acknowledgements

I wish to express my deep gratitude to my thesis advisor, Dr. Robert J. Adams, for his direction, support, and understanding from the first day in EE 622 and continuing throughout the completion of this work.

Also, I am thankful to Dr. John C. Young for his oft meetings with me to explain and guide me in understanding Open MP programming in a High Performance Computing environment.

In addition, I would like to express my appreciation to Dr. Bill T. Smith for steering me into the graduate program in Electrical Engineering at the University of Kentucky when circumstances forced me to change careers.

Furthermore, I would like to thank my wife, Tina, for her continual encouragement to complete this work while personally experiencing a most difficult season in our life.

Also, I wish to thank my four children, Buxton Jr., Laura, Wesley, and Haddon, for their patience and sacrifice, especially during my first two years back in graduate school.

Finally, I would like to acknowledge my personal savior, Jesus Christ; for without His help, strength, and upholding during great difficulty, I would never have finished this work. Isaiah 41:10

Table of Contents

Acknowledgements iii

List of Tables vi

List of Figures viii

Chapter 1. Introduction 1

- 1.1. Background 1
- 1.2. Motivation 2

Chapter 2. Model 4

- 2.1. Integral Equation Based Formulation using Method of Moments 4
- 2.2. EFIE Formulation for Perfect Electrical Conductors 5
- 2.3. Discretization and Basis Functions 7
- 2.4. RWG Model 8

Chapter 3. NASA GEMINI Solver 15

- 3.1. GEMINI Solver Structure and Existing MPI Parallelization 15
- 3.2. Computing Platforms 17
- 3.3. Preliminary MPI Testing 19
  - 3.3.1. Triangular Mesh Generation Using CUBIT 19
  - 3.3.2. Create GEMINI Solver Input Test Files Using EIGER ANTS 20
  - 3.3.3. MPI Multi-Process Test Runs on Windows-7 22
  - 3.3.4. Initial MPI Runtime Performance Measurements on DLX 25
  - 3.3.5. Improved MPI Runtime Performance Measurements on DLX 29
- 3.4. GEMINI Post RCS Measurements 37

Chapter 4. Hybrid Parallelization: Combining Open MP with MPI 39

- 4.1. Why Open MP Multi-Threading? 39
- 4.2. OMP Directives and Parallelization 45
  - 4.2.1. Using OMP on a Simple Printing Operation 46
  - 4.2.2. Testing OMP on a Matrix Multiply Operation 47
- 4.3. Integrating OMP into GEMINI Solver’s Matrix Fill Routine 56
  - 4.3.1. Incorporating Simple OMP Parallelism into GEMINI Solver 58
  - 4.3.2. Incorporating Nested OMP Parallelism into GEMINI Solver 66

Chapter 5. Testing MPI-OMP Hybrid Parallelization 71

- 5.1. Final Implementation 71
- 5.2. GEMINI Solver Results 78

5.3. GEMINI Post Results 99
Chapter 6. Conclusion & Future Direction 100
6.1. Conclusion 100
6.2. Future Direction 100
Appendix A GEMINI Post RCS Results Fit to Mie Series 101
Appendix B Final Implementation: Changes To GEMINI Solver v3.0 115
References 126
Vita 128

List of Tables

Table 1: GEMINI Solver program tree 16
Table 2: Cubit Script to create a meshed sphere with mesh size lambda/10 19
Table 3: Ten triangular surface meshes for 1-m radius sphere with λ^2 / ½ℓ^2 ≈ 200 19
Table 4: Input Files Generated to Test GEMINI Solver 22
Table 5: GEMINI Solver v1.0 test cases run under MPI for Windows-7 22
Table 6: Building GEMINI Solver on University of Kentucky HPC DLX supercluster 25
Table 7: GEMINI Solver v1.0 test cases run under MPI for Linux 25
Table 8: GEMINI Solver v2.0 test cases run under MPI for Linux 29
Table 9: GEMINI Post RCS results fit to Mie Series 38
Table 10: Loop iterations per thread for nt = 2, 4, 8, and 16 OMP threads 42
Table 11: A few common OMP directives 45
Table 12: FORTRAN 2000 program with OMP directive to execute code in parallel 46
Table 13: Sample output of FORTRAN 2000 program with OMP parallel execution 46
Table 14: Serial FORTRAN 2000 program used to multiply two matrices 48
Table 15: Parallel implementation of matrix multiplication using OMP 49
Table 16: Threading - NRA Combinations 51
Table 17: Matrix Multiply Serial and Parallel Execution Times for NRA = 16M 52
Table 18: Matrix Multiply Serial and Parallel Execution Times for NRA = 32M 53
Table 19: Matrix Multiply Serial and Parallel Execution Times for NRA = 64M 54
Table 20: Matrix Multiply Serial and Parallel Execution Times for NRA = 128M 55
Table 21: Maximum Speedup for each NRA Value 56
Table 22: GEMINI Solver program tree with optimal OMP threading location 57
Table 23: Simple OMP parallelization of FORTRAN 2000 matrix fill routine, part 1 59
Table 24: Simple OMP parallelization of FORTRAN 2000 matrix fill routine, part 2 60
Table 25: Test cases run using simple OMP parallelization of GEMINI Solver v2.0 61
Table 26: GEMINI Solver v2.0 Serial and Parallel Execution Times for N = 1083 61
Table 27: GEMINI Solver v2.0 Serial and Parallel Execution Times for N = 4455 62
Table 28: GEMINI Solver v2.0 Serial and Parallel Execution Times for N = 41,415 64
Table 29: GSv2.0 Parallel Execution S, E, and SEP for N = 4455 65
Table 30: Nested OMP parallelization of FORTRAN 2000 matrix fill routine, part 1 67
Table 31: Nested OMP parallelization of FORTRAN 2000 matrix fill routine, part 2 68
Table 32: Test cases run to compare simple vs. nested OMP parallelism 69
Table 33: Performance ratios comparing nested to simple parallelism speedup 69
Table 34: N=208K case using 224 MPI processes for 8 and 16 MPIs/node 71
Table 35: N=208K case using 224 MPI processes with 8 MPIs/node and 2 OMP threads 71
Table 36: Test cases for final implementation of hybrid MPI-OMP GEMINI Solver v3.0 75
Table 37: DLX batch script with sanity checks 76
Table 38: DLX run script to execute GEMINI Solver v3.0 test sets 77
Table 39: GEMINI Solver v3.0 Matrix Fill & Factor Times for N=41.4K, f=0.8994GHz 78
Table 40: GEMINI Solver v3.0 Matrix Fill & Factor Times for N=74.4K, f=1.1992GHz 82
Table 41: GEMINI Solver v3.0 Matrix Fill & Factor Times for N=168K, f=1.7998GHz 86
Table 42: GEMINI Solver v3.0 Matrix Fill & Factor Times for N=208K, f=2.0000GHz 90
Table 43: GEMINI Solver v3.0 Matrix Fill & Factor Times for N=299K, f=2.3984GHz 94

Table 44: GEMINI Solver v3.0 Matrix Fill & Factor Times for N=326K, f=2.5000GHz 97
Table 45: GEMINI Solver v3.0 Matrix Fill & Factor Times for N=342K, f=2.5624GHz 98
Table 46: GEMINI Post RCS results fit to Mie Series 99
Table 47: Results for best MPI-OMP combinations of matrix fill routine 100

List of Figures

Figure 1: Analyzing antenna systems on Orbital Sciences' Orion Abort Test Booster 2
Figure 2: Astronaut Extra Vehicular Activity (EVA) lunar surface studies 2
Figure 3: UHF-band shuttle antenna placement analysis 3
Figure 4: PEC with surface current $\vec{J}(x', y', z')$ 5
Figure 5: Discretization of spherical surface by triangular patches 7
Figure 6: Triangle pair and geometrical parameters associated with interior edge 8
Figure 7: Plane wave travelling down z-axis and polarized along x-axis 12
Figure 8: Distribution of J over triangular mesh as viewed from north pole 13
Figure 9: Distribution of J over triangular mesh as viewed from south pole 14
Figure 10: RCS plot for 60 MHz plane wave scattering on 1-m (0.2λ) radius PEC sphere 14
Figure 11: Parallelization with MPI processes 15
Figure 12: The CEM Laboratory at NASA's Johnson Space Center 17
Figure 13: The University of Kentucky HPC DLX Linux Cluster 18
Figure 14: EIGER ANTS Project: ℓ = 1.50 cm / f = 2.000 GHz Incident Plane Wave 20
Figure 15: Steps in making/exporting an EFIE solution test input file for a PEC sphere 21
Figure 16: Matrix factor & fill performance comparisons GSv1.0 (f=0.2998GHz/N=4K) 23
Figure 17: Matrix factor & fill performance comparisons GSv1.0 (f=0.5996GHz/N=18K) 24
Figure 18: Matrix factor & fill performance comparisons GSv1.0 (f=0.8994GHz/N=41K) 26
Figure 19: Matrix factor & fill performance comparisons GSv1.0 (f=0.8994GHz/N=83K) 27
Figure 20: Matrix factor & fill performance comparisons GSv1.0 (f=1.1992GHz/N =74K) 28
Figure 21: Matrix factor & fill performance comparisons GSv2.0 (f=0.8994GHz/N=41K) 30
Figure 22: Matrix factor & fill performance comparisons GSv2.0 (f=0.8994GHz/N=83K) 31
Figure 23: Matrix factor & fill performance comparisons GSv2.0 (f=1.1992GHz/N=74K) 32
Figure 24: Matrix factor & fill performance comparisons GSv2.0 (f=1.7988GHz/N=168K) 33
Figure 25: Matrix factor & fill performance comparisons GSv2.0 (f=2.0000GHz/N=208K) 34
Figure 26: Matrix factor & fill performance by MPIs/node GSv2.0 (f=2.0000GHz/N=208K) 35
Figure 27: Multi-threading of MPI processes 36
Figure 28: GPv2.0 EFIE & Dielectric RCS results w/fit to Mie Series for f = 0.1499 GHz 38
Figure 29: The Open MP fork-join programming model 39
Figure 30: Loop structure of the GEMINI Solver impedance matrix fills routine 40
Figure 31: Hybrid Parallelization Programming: Combining MPI/Open MP 41
Figure 32: OMP parallelization of impedance matrix fill routine 43
Figure 33: Speedup and efficiency vs number of threads for NRA = 16M 52
Figure 34: Speedup and efficiency vs number of threads for NRA = 32M 53
Figure 35: Speedup and efficiency vs number of threads for NRA = 64M 54
Figure 36: Speedup and efficiency vs number of threads for NRA = 128M 55
Figure 37: Simple parallelization of GEMINI Solver matrix fill routine 58
Figure 38: Speedup vs number of threads for N = 1083 61
Figure 39: Efficiency vs number of threads for N = 1083 62
Figure 40: Speedup vs number of threads for N = 4455 63
Figure 41: Efficiency vs number of threads for N = 4455 63
Figure 42: Speedup vs number of threads for N = 41,415 64
Figure 43: Efficiency vs number of threads for N = 41,415 65

Figure 44: Nested parallelization of GEMINI Solver 66
Figure 45: Simple vs. Nested OMP Performance ratios with 99¾% intervals 70
Figure 46: Illustration for 4-MPI / 4-Simple OMP hybrid parallelization 72
Figure 47: Matrix fill times vs. MPI-OMP combination for N=41.4K, f=0.8994GHz 78
Figure 48: Required memory/node vs. MPIs/node for N=41.4K, f=0.8994GHz 79
Figure 49: OMP threading vs. No OMP threading for N=41.4K, f=0.8994GHz 80
Figure 50: Matrix Fill Times: OMP vs. No OMP for N=41.4K, f=0.8994GHz 81
Figure 51: Matrix fill times vs. MPI-OMP combination N=74.2K, f=1.1992GHz 82
Figure 52: Required memory/node vs. MPIs/node for N=74.2K, f=1.1992GHz 83
Figure 53: OMP threading vs. No OMP threading for N=74.2K, f=1.1992GHz 84
Figure 54: Matrix Fill Times: OMP vs. No OMP for N=74.2K, f=1.1992GHz 85
Figure 55: Matrix fill times vs. MPI-OMP combination for N=168K, f=1.7998GHz 86
Figure 56: Required memory/node vs. MPIs/node for N=168K, f=1.7998GHz 87
Figure 57: OMP threading vs. No OMP threading for N=168K, f=1.7998GHz 88
Figure 58: Matrix Fill Times: OMP vs. No OMP for N=168K, f=1.7998GHz 89
Figure 59: Matrix fill times vs. MPI-OMP combination for N=208K, f=2.0000GHz 90
Figure 60: Required memory/node vs. MPIs/node for N=208K, f=2.0000GHz 91
Figure 61: OMP threading vs. No OMP threading for N=208K, f=2.0000GHz 92
Figure 62: Matrix Fill Times: OMP vs. No OMP for N=208K, f=2.0000GHz 93
Figure 63: Matrix fill times vs. MPI-OMP combination for N=299K, f=2.3984GHz 94
Figure 64: Required memory/node vs. MPIs/node for N=299K, f=2.3984GHz 95
Figure 65: OMP threading vs. No OMP threading for N=299K, f=2.3984GHz 96
Figure 66: Matrix Fill Times: OMP vs. No OMP for N=299K, f=2.3984GHz 96
Figure 67: Matrix fill times & memory usage for N=326K, f=2.5000GHz 97
Figure 68: Matrix fill times & memory usage for N=342K, f=2.5624GHz 98
Figure 69: N_{max} vs. Number of Unknowns 99
Figure 70: (Upper-Left) Gemini EFIE RCS results vs. Mie Series for f=0.1499 GHz 101
Figure 71: (Upper-Left) Gemini Dielectric RCS results vs. Mie Series for f=0.1499 GHz 102
Figure 72: (Upper-Left) Gemini EFIE RCS results vs. Mie Series for f=0.2998 GHz 103
Figure 73: (Upper-Left) Gemini Dielectric RCS results vs. Mie Series for f=0.2998 GHz 104
Figure 74: (Upper-Left) Gemini EFIE RCS results vs. Mie Series for f=0.5996 GHz 105
Figure 75: (Upper-Left) Gemini Dielectric RCS results vs. Mie Series for f=0.5996 GHz 106
Figure 76: (Upper-Left) Gemini EFIE RCS results vs. Mie Series for f=0.8994 GHz 107
Figure 77: (Upper-Left) Gemini Dielectric RCS results vs. Mie Series for f=0.8994 GHz 108
Figure 78: (Upper-Left) Gemini EFIE RCS results vs. Mie Series for f=1.1992 GHz 109
Figure 79: (Upper-Left) Gemini EFIE RCS results vs. Mie Series for f=1.7988 GHz 110
Figure 80: (Upper-Left) Gemini EFIE RCS results vs. Mie Series for f=2.0000 GHz 111
Figure 81: (Upper-Left) Gemini EFIE RCS results vs. Mie Series for f=2.3984 GHz 112
Figure 82: (Upper-Left) Gemini EFIE RCS results vs. Mie Series for f=2.5000 GHz 113
Figure 83: (Upper-Left) Gemini EFIE RCS results vs. Mie Series for f=2.5624 GHz 114

Chapter 1. Introduction

1.1. Background

Understanding, predicting, and controlling electromagnetic field interactions on and between complex RF platforms is essential to the design and analysis of various NASA communications and sensing systems. Electromagnetic interactions underlie various critical properties of such systems including antenna radiation patterns and impedances, near field distributions, instrument-platform interactions, link budgets, etc. Modern platforms are increasingly sophisticated and complex, and accurate electromagnetic modeling of these platforms in their deployed environment requires high fidelity computational electromagnetic (CEM) simulation tools. While NASA currently has significant capability in this area, existing and future design and analysis requirements exceed the capabilities of its existing CEM tools.

The primary CEM tool within NASA is GEMINI (formerly EIGER [1]), which is maintained by the members of the CEM Laboratory [2] at NASA's Johnson Space Center (JSC) in Houston. GEMINI is an integral equation based method-of-moments (MoM) code for frequency domain electromagnetic modeling [3]. GEMINI is well tested and has been proven accurate and effective in a number of real world applications. However, GEMINI is currently limited in the size and complexity of problems that can be effectively handled. The CEM Lab at JSC relies on a supercomputer with 476 processors and 1.9 terabytes of RAM. The GEMINI tool currently relies on standard distributed LU decomposition techniques. For this reason, the group's simulation capabilities are currently limited to about 290,000 unknowns.

A primary NASA research project at the University of Kentucky involves the integration of the sparse linear solution library, MFDlib, developed at the University of Kentucky, with GEMINI for efficient filling, factorization, and solution of large electromagnetic problems formulated using integral equation methods [4]. A secondary research project involves the hybrid parallelization of GEMINI for the efficient speedup of the filling process. This thesis discusses the research, development, and testing of this secondary research project. The next section identifies several problems that emphasize the need NASA has to extend its CEM capabilities beyond those currently available.

1.2. Motivation

NASA's Computational Electromagnetics (CEM) Laboratory has a strong reputation for providing reliable, accurate solutions for a wide range of practical electromagnetic problems. Some of NASA-JSC CEM Lab's recent activities include analyzing the flight termination system (FTS) antennas on four different vehicles: NASA's ARES I-X Rocket (2008), SpaceX's Falcon-9 rocket (2009), Orbital Science Corporation's Taurus II Rocket (2010-2011), and SpaceX (2011).

Analyzing the antenna systems on the Orion Abort Test Booster (ATB) was required in four frequency bands: UHF, L-Band, S-Band, and C-Band. See Figure 1.

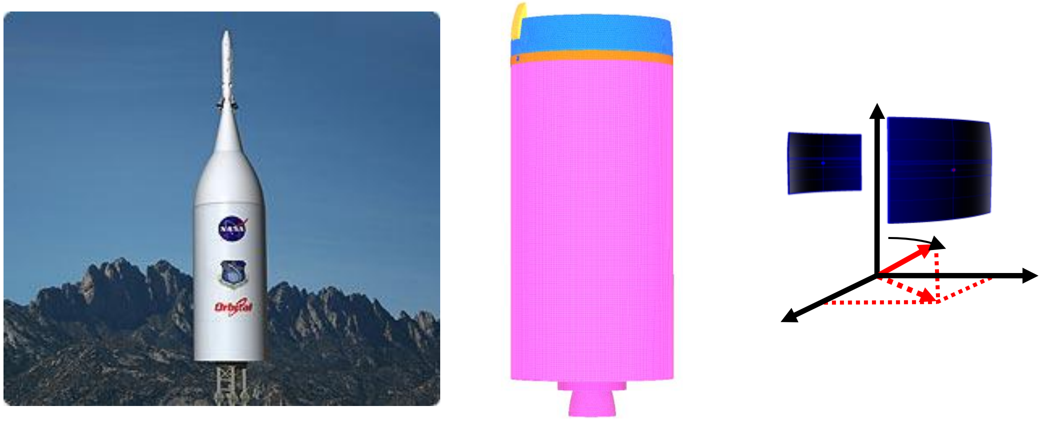


Figure 1: Analyzing antenna systems on Orbital Sciences' Orion Abort Test Booster

NASA is studying C-band antennas mounted at different locations on an astronaut suit. The entire astronaut can be modeled at UHF band using GEMINI. See Figure 2.

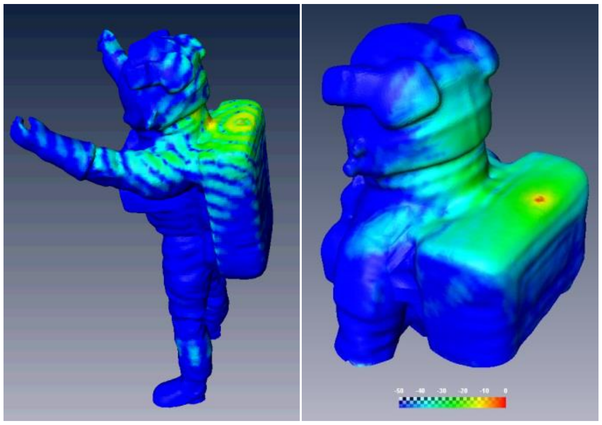


Figure 2: Astronaut Extra Vehicular Activity (EVA) lunar surface studies

The NASA-JSC CEM Lab provided computational analyses to relocate Space-to-Space Orbiter Radio (SSOR) and Wireless Video System (WVS) antennas to accommodate the Orbiter Boom Sensor System. The purpose of the CEM analysis was to ensure that the antennas were located in regions that provided sufficient coverage to astronauts performing EVA maneuvers. The analysis required the evaluation of 25 different antenna locations on the vehicle, with each location requiring a CEM simulation analysis. See antenna photo and CEM antenna simulation analysis in Figure 3.

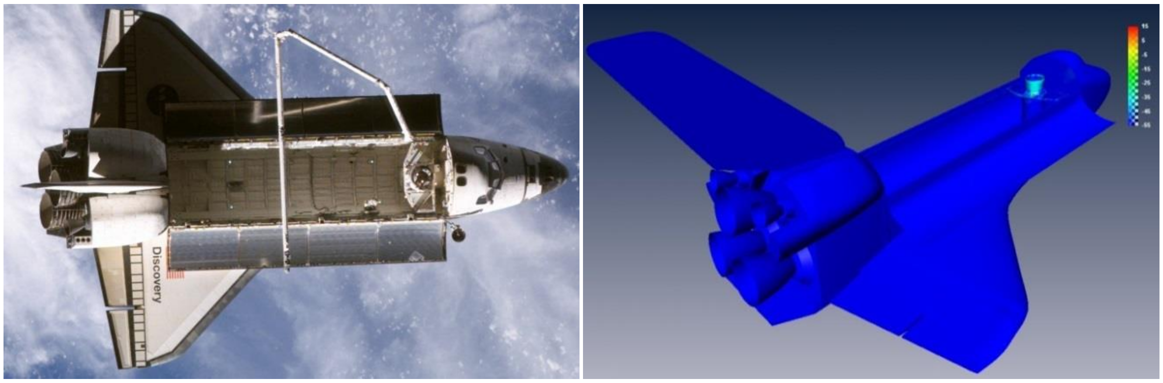


Figure 3: UHF-band shuttle antenna placement analysis

As indicated above, the CEM Laboratory at NASA’s Johnson Space Center has significant experience with and confidence in the GEMINI computational electromagnetics tool for a variety of electromagnetic modeling applications. However, there is also a clear need to extend the capabilities of GEMINI in order to treat more complex problems. While extensions to GEMINI would be desirable in a few areas, the most significant of these is the need to be able to model complex platforms in higher frequency bands (e.g., L-, S- and C-) than is currently possible.

As mentioned previously, the primary research project to integrate the MFDlib library with GEMINI and the secondary research project (discussed in this thesis) to develop hybrid parallelization of GEMINI target the important need NASA has to extend its CEM capabilities beyond those currently available.

Chapter 2. Model

2.1. Integral Equation Based Formulation using Method of Moments

GEMINI employs an integral equation based method-of-moments code for frequency domain electromagnetic modeling. The method-of-moments (MoM) is a numerical method of solving linear partial differential equations which have been formulated as integral equations [5]. It can be applied in many areas of engineering and science including fluid mechanics, acoustics, electromagnetics, fracture mechanics, and plasticity. In this thesis, we consider MoM only for boundary integral equation formulations (BIEs) of time harmonic electromagnetic field scattering from piecewise homogenous dielectric and conducting materials.

MoM for BIEs has become more popular since the 1980s because it enables the solution for fields at all points in space using only boundary values, rather than values of the fields throughout all of the space. It is significantly more efficient in terms of computational resources for problems with a small surface/volume ratio. Conceptually, it works by constructing a "mesh" over the modeled surface. However, for many problems, the boundary element method (BEM) is significantly computationally less efficient than volume-discretization methods (finite element method, finite difference method, finite volume method). Boundary element formulations typically give rise to fully populated matrices. This means the storage requirements will tend to grow according to the square of the problem size, and the computational times will tend to grow as the cube of the problem size. One way to reduce these costs is to use data-sparse solution methods such as those provided by the University of Kentucky MFDlib library [4]. Such strategies are not discussed in this thesis.

BEM is applicable to problems for which Green's functions can be calculated. These usually involve fields in linear homogeneous media. This places considerable restrictions on the range and generality of problems suitable for boundary elements. Nonlinearities can be included in the formulation, although they generally introduce volume integrals which require the volume to be discretized before solution. In some cases, this removes an oft-cited advantage of BEM.

2.2. EFIE Formulation for Perfect Electrical Conductors

The Electric Field Integral Equation (EFIE) formulation has the advantage of being applicable to both open and closed bodies [6]. Let S denote the surface of an open or closed PEC scatterer with unit normal $\hat{n}$. An electric field $\vec{E}^i$ (due to an impressed source in the absence of a scatterer), is incident on and induces surface currents $\vec{J}$ on S.

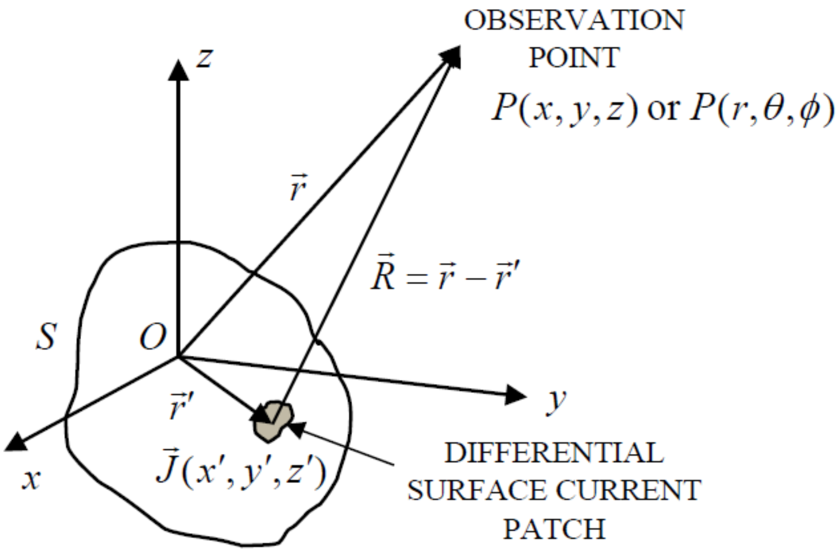


Figure 4: PEC with surface current $\vec{J}(x',y',z')$

If S is open, $\vec{J}$ is the vector sum of the surface current on opposite sides of S and, therefore, the normal component of $\vec{J}$ must vanish on boundaries of S. The scattered electric field $\vec{E}^s$ can be computed from surface current by

$$\vec{E}^s=-j\omega\vec{A}-\vec{\nabla}\Phi \tag{1}$$

where the magnetic vector potential is defined as

$$\vec{A}(\vec{r})=\frac{\mu}{4\pi}\int_S \vec{J}\,\frac{e^{-jkR}}{R}\,dS' \tag{2}$$

and the scalar potential is defined as

$$\Phi(\vec{r}) = \frac{1}{4\pi\varepsilon}\int_S \sigma \frac{e^{-jkR}}{R} dS' \qquad (3)$$

A harmonic time dependence ($e^{j\omega t}$) is assumed and $k = \omega\sqrt{\mu\varepsilon} = 2\pi/\lambda$, where λ is the wavelength. $R = \left|\vec{r} - \vec{r}'\right|$ is the distance between an arbitrarily located observation point $\vec{r}$ and a source point $\vec{r}'$ on S. The surface charge density σ is related to the surface divergence of $\vec{J}$ through the equation of continuity

$$\vec{\nabla}_S \cdot \vec{J} = -j\omega\sigma \qquad (4)$$

Enforcing the boundary condition

$$\hat{n} \times \left(\vec{E}^{\,i} + \vec{E}^{\,s}\right) = 0 \qquad (5)$$

on S, we obtain

$$\left(-j\omega\vec{A} - \vec{\nabla}\Phi\right)_{\tan} = -\vec{E}^{\,i}_{\tan} \qquad (6)$$

which constitutes the electric field integral equation (EFIE). The EFIE method constitutes solving eq.(6) utilizing the magnetic vector potential [eq.(2)], scalar potential [eq.(3)], and continuity equation [eq.(4)].

Although the EFIE method has the advantage of being applicable to both open and closed bodies, it can be difficult to apply due to the kernel of the integral containing a singularity. When computing self-interactions, the source and observation points are the same ($\vec{r} = \vec{r}\,'$) and the integrals contain a singularity at $R = 0$. Transformations such as Duffy's transformation of source coordinates can be used to remove the singularity [5].

2.3. Discretization and Basis Functions

The current $\vec{J}$ on surface S can be approximated in terms of a series of vector basis functions $\vec{f}_n(\vec{r})$. A discrete computational representation of the problem to be solved typically includes a mesh of some simple shape, together with parameter values that specify the physical properties in the material of each mesh element [7]. In EFIE formulations, continuous surface models employ surface "patches" with overlapping basis functions. The patch shape of choice for discretization of a continuous surface is the planar triangular patch [8] displayed in Figure 5.

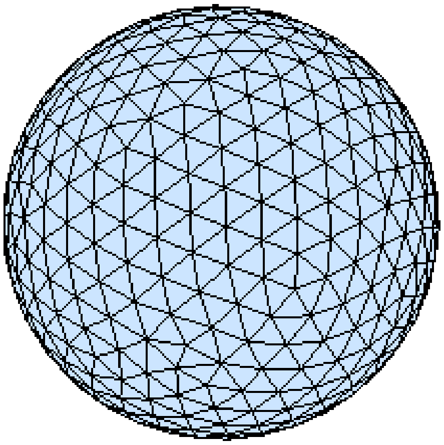


Figure 5: Discretization of spherical surface by triangular patches

Planar triangular patches are capable of accurately conforming to any geometrical surface or boundary with the desired tolerance, are easily specified for computer input, and allow for varying patch density to accommodate small geometry features and sharp variation in anticipated current density.

If the basis functions representing the surface current are not constructed such that their normal components are continuous across the patch edges, then the continuity equation [eq. (4)] requires the presence of point or line charges at the edges. These fictitious charges, if present, can cause erroneous solutions in some cases and are to be avoided for that reason. Thus, the basis function of choice for triangular patches is the RWG basis function (discussed in next section) which avoids difficulties at patch edges.

2.4. RWG Model

The Rao-Wilton-Glisson (RWG) model uses a special set of vector basis functions $\vec{f}_n(\vec{r})$ which are suitable for use with the EFIE and a triangular mesh to approximate S [6]. The surface mesh is divided into triangular pairs with a common interior edge as shown in Figure 6.

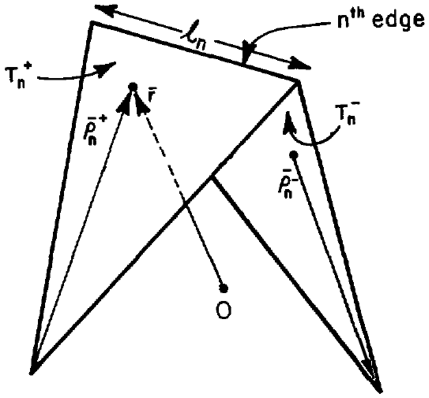


Figure 6: Triangle pair and geometrical parameters associated with interior edge

The vector basis function associated with the nth edge of a triangular pair as shown in Figure 6 is given by [5], [6], [8]

$$\vec{f}_n(\vec{r}) = \begin{cases} \dfrac{l_n}{2A_n^+}\vec{\rho}_n^{\,+} & for \quad \vec{r} \ in \ T_n^+ \\ \dfrac{l_n}{2A_n^-}\vec{\rho}_n^{\,-} & for \quad \vec{r} \ in \ T_n^- \\ 0 & \qquad otherwise \end{cases} \tag{7}$$

where l_n is the length of the edge and $A_n^{\pm}$ is the area of the triangle $T_n^{\pm}$. The current on S can be approximated in terms of the vector basis functions $\vec{f}_n(\vec{r})$ as

$$\vec{J} \cong \sum_{n=1}^{N} I_n \vec{f}_n(\vec{r}) \tag{8}$$

where N is the number of interior (non-boundary) edges and I_n is the coefficient (interpreted as the normal component of current density flowing past the n*th* edge). If the vector basis functions $\vec{f}_m(\vec{r})$ are also used as testing functions at observation triangles for the EFIE [eq. (6)], then we obtain

$$j\omega\int_S \vec{A}(\vec{r})\cdot \vec{f}_m(\vec{r})dS + \int_S \vec{\nabla}\Phi(\vec{r})\cdot \vec{f}_m(\vec{r})dS = \int_S \vec{E}^i(\vec{r})\cdot \vec{f}_m(\vec{r})dS \tag{9}$$

The testing integral over each triangle can be eliminated by using the surface vector calculus identity and approximating Φ, $\vec{E}^{\,i}$, and $\vec{A}$ by their values at the observation triangle centroid yielding the EFIE equation

$$\begin{aligned} l_m\left\{ j\omega\left[\vec{A}\left(\vec{r}_m^{\,c+}\right)\cdot \frac{\vec{\rho}_m^{\,c+}}{2} + \vec{A}\left(\vec{r}_m^{\,c-}\right)\cdot \frac{\vec{\rho}_m^{\,c-}}{2} \right] + \left[\Phi\left(\vec{r}_m^{\,c-}\right) - \Phi\left(\vec{r}_m^{\,c+}\right)\right] \right\} &= \\ l_m\left[\vec{E}^i\left(\vec{r}_m^{\,c+}\right)\cdot \frac{\vec{\rho}_m^{\,c+}}{2} + \vec{E}^i\left(\vec{r}_m^{\,c-}\right)\cdot \frac{\vec{\rho}_m^{\,c-}}{2} \right]& \end{aligned} \tag{10}$$

Substituting the current $\vec{J}$ [eq.(8)] into the EFIE [eq. (10)] yields an N x N system of linear equations

$$\begin{aligned} l_m\left[\begin{array}{l} j\dfrac{\omega\mu}{4\pi}\left\{\left[\displaystyle\int_S \vec{f}_n(\vec{r}\,')\frac{e^{-jk\left|\vec{r}_m^{\,c+}-\vec{r}\,'\right|}}{\left|\vec{r}_m^{\,c+}-\vec{r}\,'\right|}dS'\right]\cdot\dfrac{\vec{\rho}_m^{\,c+}}{2} + \left[\displaystyle\int_S \vec{f}_n(\vec{r}\,')\frac{e^{-jk\left|\vec{r}_m^{\,c-}-\vec{r}\,'\right|}}{\left|\vec{r}_m^{\,c-}-\vec{r}\,'\right|}dS'\right]\cdot\dfrac{\vec{\rho}_m^{\,c-}}{2}\right\} \\ +\, j\dfrac{1}{4\pi\omega\varepsilon}\left\{\left[\displaystyle\int_S \vec{\nabla}'_S\cdot\vec{f}_n(\vec{r}\,')\frac{e^{-jk\left|\vec{r}_m^{\,c+}-\vec{r}\,'\right|}}{\left|\vec{r}_m^{\,c+}-\vec{r}\,'\right|}dS'\right] - \left[\displaystyle\int_S \vec{\nabla}'_S\cdot\vec{f}_n(\vec{r}\,')\frac{e^{-jk\left|\vec{r}_m^{\,c-}-\vec{r}\,'\right|}}{\left|\vec{r}_m^{\,c-}-\vec{r}\,'\right|}dS'\right]\right\} \end{array}\right] I_n &= \\ l_m\left[\vec{E}^i\left(\vec{r}_m^{\,c+}\right)\cdot \frac{\vec{\rho}_m^{\,c+}}{2} + \vec{E}^i\left(\vec{r}_m^{\,c-}\right)\cdot \frac{\vec{\rho}_m^{\,c-}}{2} \right]& \end{aligned} \tag{11}$$

The N x N system of linear equations can be written in matrix form as follows

$$\tilde{Z}\,\vec{I} = \vec{V} \tag{12}$$

where $\tilde{Z} = [Z_{mn}]$ is an N x N matrix (known as the impedance matrix), $\vec{I} = [I_n]$ and $\vec{V} = [V_m]$ are column vectors of length N, *m* is the index over N observation triangles, and *n* is the index over N source triangles. Elements of $\tilde{Z}$ and $\vec{V}$ are given by [6]

$$Z_{mn} = l_m \left[\begin{aligned} & j\frac{\omega\mu}{4\pi}\left\{ \left[\int_S \vec{f}_n(\vec{r}\,')\frac{e^{-jk\left|\vec{r}_m^{\,c+}-\vec{r}\,'\right|}}{\left|\vec{r}_m^{\,c+}-\vec{r}\,'\right|}dS' \right]\cdot\frac{\vec{\rho}_m^{\,c+}}{2} + \left[\int_S \vec{f}_n(\vec{r}\,')\frac{e^{-jk\left|\vec{r}_m^{\,c-}-\vec{r}\,'\right|}}{\left|\vec{r}_m^{\,c-}-\vec{r}\,'\right|}dS' \right]\cdot\frac{\vec{\rho}_m^{\,c-}}{2} \right\} \\ & + j\frac{1}{4\pi\omega\varepsilon}\left\{ \left[\int_S \vec{\nabla}'_S\cdot\vec{f}_n(\vec{r}\,')\frac{e^{-jk\left|\vec{r}_m^{\,c+}-\vec{r}\,'\right|}}{\left|\vec{r}_m^{\,c+}-\vec{r}\,'\right|}dS' \right] - \left[\int_S \vec{\nabla}'_S\cdot\vec{f}_n(\vec{r}\,')\frac{e^{-jk\left|\vec{r}_m^{\,c-}-\vec{r}\,'\right|}}{\left|\vec{r}_m^{\,c-}-\vec{r}\,'\right|}dS' \right] \right\} \end{aligned} \right] \tag{13}$$

and

$$V_m = l_m\left[\vec{E}^{\,i}\left(\vec{r}_m^{\,c+}\right)\cdot\frac{\vec{\rho}_m^{\,c+}}{2} + \vec{E}^{\,i}\left(\vec{r}_m^{\,c-}\right)\cdot\frac{\vec{\rho}_m^{\,c-}}{2} \right] \tag{14}$$

Once the elements Z_{mn} of the impedance matrix and the elements V_m of the column vector are determined, the N x N system of linear equations can be solved for the unknown current coefficients I_n. Once solved, the current across the nth edge is found by

$$\vec{J}_n = I_n\,\vec{f}_n(\vec{r}) \tag{15}$$

For plane wave incidence, the components of $\vec{E}^{\,i}(\vec{r})$ are

$$\begin{aligned} E_x^i &= \left(E_\theta\cos\theta_0\cos\phi_0 - E_\phi\sin\phi_0\right)e^{j\vec{k}\cdot\vec{r}} \\ E_y^i &= \left(E_\theta\cos\theta_0\sin\phi_0 + E_\phi\cos\phi_0\right)e^{j\vec{k}\cdot\vec{r}} \\ E_z^i &= -\left(E_\theta\sin\theta_0\right)e^{j\vec{k}\cdot\vec{r}} \end{aligned} \tag{16}$$

where the components of the propagation vector $\vec{k}$ are

$$\begin{aligned} k_x &= k \sin\theta_0 \cos\phi_0 \\ k_y &= k \sin\theta_0 \sin\phi_0 \\ k_z &= k \cos\theta_0 \end{aligned} \tag{17}$$

and (θ_0, ϕ_0) defines the angle of arrival of the plane wave.

The ability of surface S to scatter electromagnetic waves can be summarized into a single term, σ, known as the radar cross-section (RCS). The RCS of surface S can be viewed as a ratio of the strength of the scattered wave from surface S to the scattered wave from a perfectly smooth sphere of cross sectional area of 1 m^2. After solving for the unknown currents, the RCS can be computed by [5]

$$\sigma_{3-D} = \frac{(k\eta)^2}{4\pi} \left| \int_S \vec{E}^{s*}(\vec{r}) \cdot \vec{J}(\vec{r}) dS \right|^2 \tag{18}$$

where the plane wave $\vec{E}^{s*}(\vec{r})$ in the scattering direction is polarized in the $\hat{\theta}$ or $\hat{\phi}$ direction. The integral [eq. (18)] can be evaluated by substituting the expansion for the surface current density in terms of basis functions [eq. (8)]. This leads to

$$\sigma_{3-D} = \frac{(k\eta)^2}{4\pi} \left| \sum_{n=1}^{N} I_n \int_S \vec{E}^{s*}(\vec{r}) \cdot \vec{f}_n(\vec{r}) dS \right|^2 \tag{19}$$

For a given pair of incident and scattering directions, there are four possible combinations of polarizations for the incident and scattered plane waves, so in general $\sigma_{\theta\theta}$, $\sigma_{\theta\phi}$, $\sigma_{\phi\theta}$, and $\sigma_{\phi\phi}$ must be computed to characterize the scattering properties of the object.

To demonstrate, the EFIE method can be used on the spherical mesh geometry of Figure 5 to solve for the triangle edge currents. Consider the spatial extent along the diameter of a 2-m sphere to be 0.4λ. A plane wave is incident from above the sphere, travelling down the z-axis, and polarized along the x-axis.

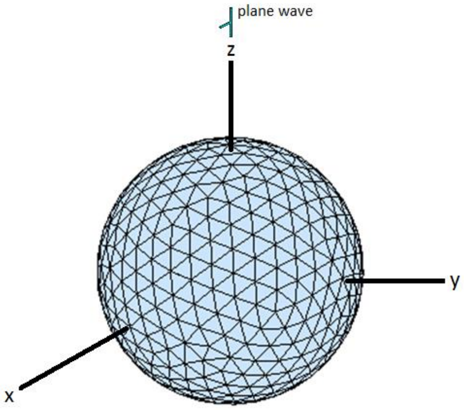


Figure 7: Plane wave travelling down z-axis and polarized along x-axis

For a plane wave incidence at $\phi_0 = 0$, $\theta_0 = 0$, $E_\phi = 0$, $E_\theta = -1$, the components of $\vec{E}^{\,i}(\vec{r})$ are

$$\begin{aligned} E_x^i &= -e^{jkz} \\ E_y^i &= 0 \\ E_z^i &= 0 \end{aligned} \tag{20}$$

and the components of the propagation vector $\vec{k}$ are

$$\begin{aligned} k_x &= 0 \\ k_y &= 0 \\ k_z &= k \end{aligned} \tag{21}$$

The incident plane wave is therefore in the **-z** direction and theta-polarized. The incident plane wave has wavelength λ = 2 m / 0.4 = 5 m and frequency f = 60.0 MHz.

The triangle edge currents are solved as follows:

1. $\vec{V} = [V_m]$ is filled;
2. Impedance matrix $\tilde{Z} = [Z_{mn}]$ is computed (by matrix fill routine);
3. Inverse impedance matrix $\tilde{Z}^{-1}$ is computed (by matrix factorization routine);
4. Matrix equation $\vec{I} = \tilde{Z}^{-1}\vec{V}$ yields $\vec{I} = [I_n]$ giving the current coefficients I_n for the triangle pair edges;
5. Each current coefficient I_n is multiplied by the appropriate basis function $\vec{f}_n(\vec{r})$ [eq.(15)] to determine the distribution of $\vec{J}$ over the triangles in the mesh.

Color density plots showing the distribution of $\vec{J}$ over the triangle mesh as viewed from the north pole and south pole, respectively, are shown in Figure 8 and Figure 9.

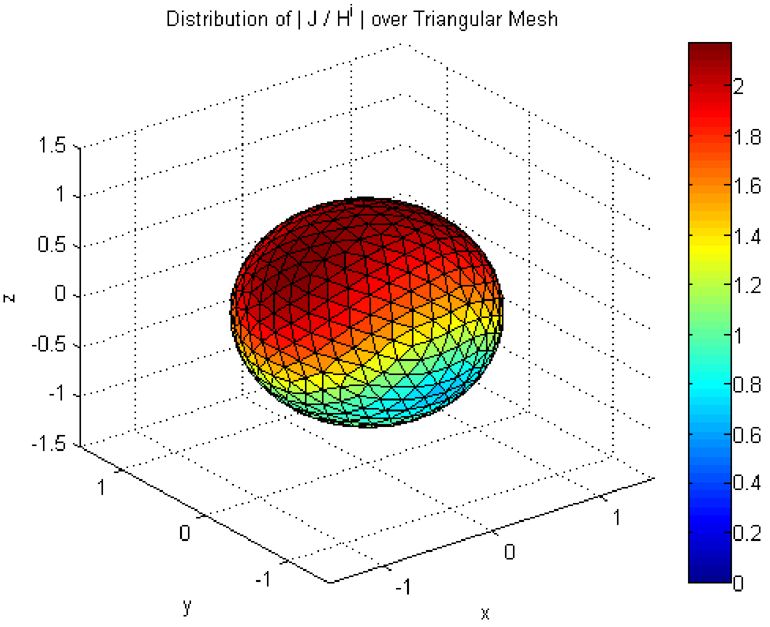


Figure 8: Distribution of J over triangular mesh as viewed from north pole

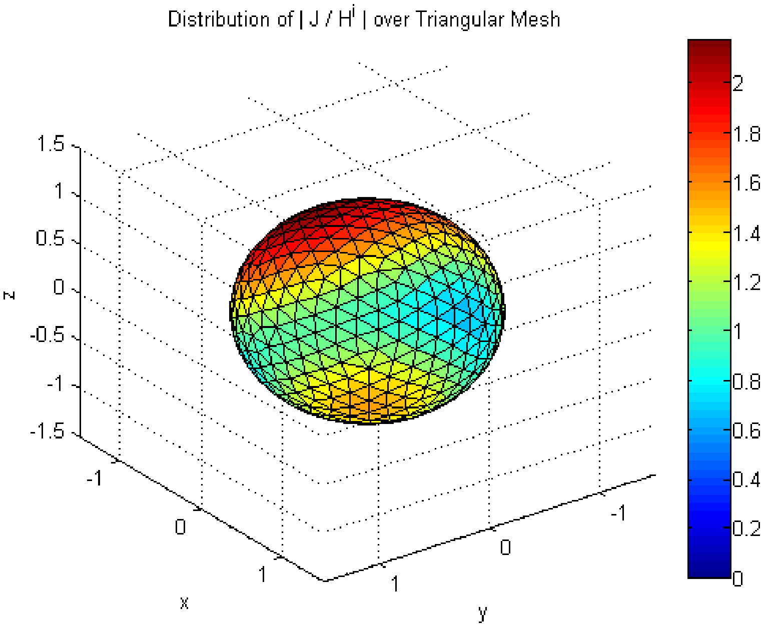


Figure 9: Distribution of J over triangular mesh as viewed from south pole

The RCS scattering results (E plane ϕ = 0) for a 60 MHz incident plane wave impingent on the 1 meter radius PEC sphere is shown in Figure 10.

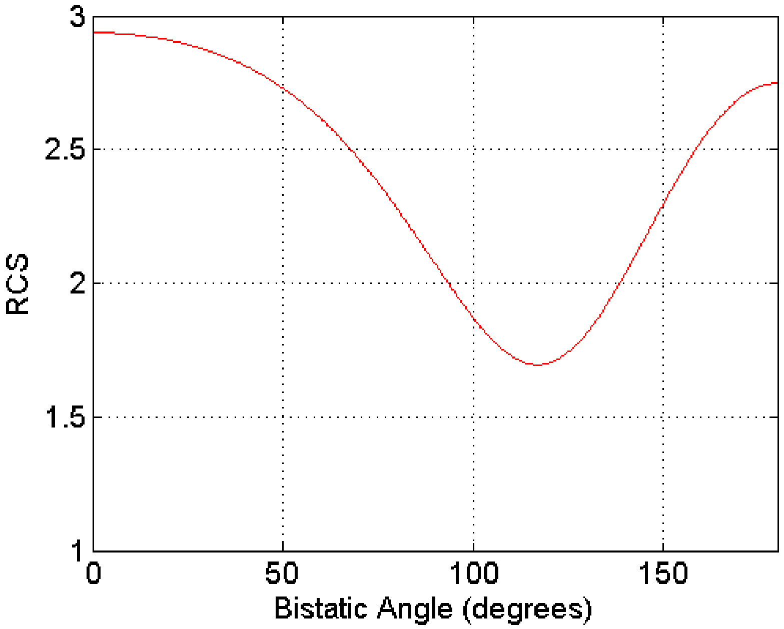


Figure 10: RCS plot for 60 MHz plane wave scattering on 1-m (0.2λ) radius PEC sphere

Chapter 3. NASA GEMINI Solver

3.1. GEMINI Solver Structure and Existing MPI Parallelization

Accurate electromagnetic (EM) analysis plays a critical role in NASA's mission. The primary tool for this purpose within NASA is GEMINI. GEMINI Solver calculates the primary electromagnetic quantities (e.g., currents) and secondary quantities of interest (e.g., far fields, etc.), if desired. The general framework is open to many applications including antenna design, radio frequency design, and passive microwave device design. GEMINI Solver runs on a variety of platforms with a FORTRAN 2000 compiler, including Windows and Linux. Additionally, GEMINI executes in parallel on machines using Message Passing Interface (MPI) libraries. MPICH [9] by Argonne National Laboratory is a high performance, widely portable implementation of the MPI standard.

MPI was developed to facilitate portable programming for distributed-memory architectures, where multiple processes execute independently and communicate data as needed by exchanging messages. MPI provides a comprehensive set of library routines for managing processes and exchanging messages. MPI is widely used in high-end computing, where problems are so large that a cluster of computers is needed to solve them. Figure 11 depicts a simple MPI program running with four processes.

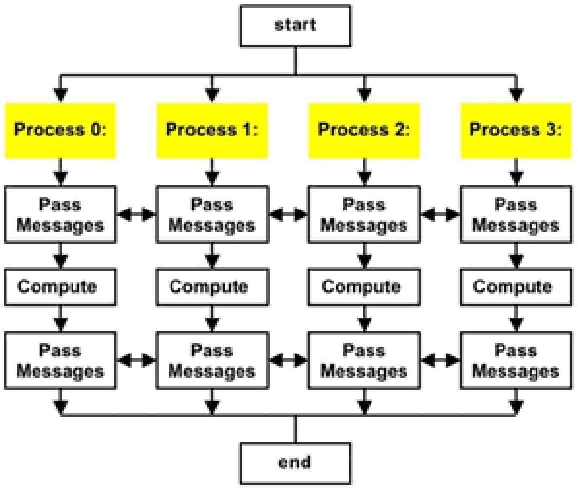


Figure 11: Parallelization with MPI processes

The program tree for GEMINI Solver is shown in Table 1 on the next page.

Table 1: GEMINI Solver program tree

GEMINI Solver Program Tree

```
Begin GEMINI Solver
    CALL & RETURN ProjectModule%openPrimaryFiles
    CALL & RETURN ProjectModule%readFromFile
    CALL & RETURN ProjectModule%writeSimulationData
    CALL SolutionModule%solution
        CALL SolutionModule%nonperiodic/periodic
        ! loop over frequencies
            ! Begin “Fill the impedance matrix”
            CALL GlobalMatrixModule%fillNormal
            ! loop over observation elements --> loop over node sets for this observation element
                ! Begin “Moment method analysis”
                ! loop over source elements --> loop over node sets for this source element
                    CALL LocalMatrixModule%fillMoM
                    ! loop over observation points --> loop over the number of common regions
                        CALL OperatorsModule%FillZMatrix
                        ! loop over quadrature points --> loop over source nodes --> loop over observation nodes
                            ! calculation: obsBasis%lambda(iObs) .dot. srcBasis(iQuad)%lambda(jSrc))*G(iQuad)*srcWghtJacobian(iQuad)*obsWghtJacobian
                            ! calculation: obsBasis%divLambda(iObs)*srcBasis(iQuad)%divLambda(jSrc)*srcWghtJacobian(iQuad)*obsWghtJacobian
                        ! end loop over observation nodes --> end loop over source nodes --> end loop over quadrature points
                        RETURN OperatorsModule%FillZMatrix
                        CALL OperatorsModule%FillBetaMatrix
                        ! loop over quadrature points --> loop over source nodes --> loop over observation nodes
                            ! calculation: obsBasisCrossNormal(iObs) .dot. (gradG(iQuad) .cross. srcBasis(iQuad)%lambda(jSrc)))*srcWghtJacobian(iQuad)*obsWghtJacobian
                        ! end loop over observation nodes --> end loop over source nodes --> end loop over quadrature points
                        RETURN OperatorsModule%FillBetaMatrix
                        CALL OperatorsModule%FillYMatrix
                        ! loop over quadrature points --> loop over source nodes --> loop over observation nodes
                            ! calculation: obsBasisCrossNormal(iObs) .dot. srcBasis(iQuad)%lambda(jSrc))*G(iQuad)*srcWghtJacobian(iQuad)*obsWghtJacobian
                            ! calculation: obsBasisCrossNormal(iObs) .dot. gradG(iQuad))*srcBasis(iQuad)%divLambda(jSrc)*srcWghtJacobian(iQuad)*obsWghtJacobian
                        ! end loop over observation nodes --> end loop over source nodes --> end loop over quadrature points
                        RETURN OperatorsModule%FillYMatrix
                        CALL OperatorsModule%FillBetaTildeMatrix
                        ! loop over quadrature points --> loop over source nodes --> loop over observation nodes
                            ! calculation: obsBasis%lambda(iObs) .dot. (gradG(iQuad) .cross. srcBasis(iQuad)%lambda(jSrc)))*srcWghtJacobian(iQuad)*obsWghtJacobian
                        ! end loop over observation nodes --> end loop over source nodes --> end loop over quadrature points
                        RETURN OperatorsModule%FillBetaTildeMatrix
                    ! end loop over the number of common regions --> end loop over observation points
                    RETURN LocalMatrixModule%fillMoM
                    CALL & RETURN GlobalMatrixModule%localToGlobal
                ! end loop over node sets for this source element --> end loop over source elements
                ! End “Moment method analysis”
            ! end loop over node sets for this observation element --> end loop over observation elements
            RETURN GlobalMatrixModule%fillNormal
            ! End “Fill the impedance matrix”
            ! Begin “Fill the right hand side for each excitation group”
            ! loop over excitation groups
                CALL & RETURN GlobalVector%fill
                ! loop over global wave sources --> loop over elements --> loop over node sets for this element
                    LocalVectorModule%fillLocalSource
                    LocalVectorModule%localToGlobal
                ! end loop over node sets for this element --> end loop over elements --> loop over global wave sources
                ! loop over voltage sources-> loop over node sets
                    LocalVectorModule%fillLocalSource
                    LocalVectorModule%localToGlobal
                ! end loop over node sets --> end loop over voltage sources
                CALL & RETURN ISISComplexSolverModule%solve
                CALL & RETURN ISISComplexVectorModule%gather
                CALL & RETURN SolutionModule%writeSolution
            ! end loop over excitation groups
            ! End “Fill the right hand side for each excitation group”
        ! end loop over frequencies
        RETURN SolutionModule%nonperiodic/periodic
    RETURN SolutionModule%solution
    CALL & RETURN ProjectModule%closeFiles
End GEMINI Solver
```

3.2. Computing Platforms

GEMINI is maintained by the members of the CEM Laboratory at NASA's Johnson Space Center (JSC) in Houston, Texas. The CEM Lab at JSC [2], shown below in Figure 12, relies on a supercomputer with 476 processors and 1.9 terabytes of RAM.

Figure 12: The CEM Laboratory at NASA's Johnson Space Center

The GEMINI tool currently relies on standard distributed LU decomposition techniques. For this reason, the group's simulation capabilities are currently limited to about 290,000 unknowns. Simulations of that size require about 12 hours. Due to space/cooling limitations within the CEM Lab, they can fit only five more supercomputer racks in the lab. If the lab were filled to capacity with oct-core blade clusters with 8 GB RAM/processor, then the group could handle problems with up to 940,000 unknowns using GEMINI in its current configuration.

GEMINI Solver is first built at the University of Kentucky within Visual Studio under MPI (MPICH) and tested in the ECE Electromagnetics Laboratory on a Dell workstation with an 8-core Intel Xeon CPU X5450 @ 3.00 GHz and 64.0 GB RAM running Windows-7. Section 3.3.3 discusses the results of the initial MPI parallel testing.

Once the computing capacity of the Dell workstation is exceeded due to large size MoM problems, GEMINI Solver is built under MPI [10] and tested on the University of Kentucky HPC DLX Linux Cluster [11]. The DLX, shown in Figure 13, is a supercomputer cluster with 256 Nodes (4096 cores), ~95 Teraflops, Dell C6220 Server, 4 nodes per 2U chassis, Dual Intel E5-2670 8 Core (Sandy Bridge) @ 2.6 GHz, 2 sockets/node x 8 cores/socket = 16 cores/node, 64 GB/node of 1600 MHz RAM, 500 GB local (internal) SATA disk, Linux OS (RHEL).

UK UNIVERSITY OF KENTUCKY
Enabling Research through Cyber Infrastructure

Figure 13: The University of Kentucky HPC DLX Linux Cluster

Sections 3.3.4 and 3.3.5 of this thesis present the results of MPI parallel testing on the DLX. MoM problem sizes are increased to find GEMINI Solver’s limit on the DLX utilizing MPI parallelization. Chapter 4 and Chapter 5 discuss the hybrid parallelization development and testing of GEMINI Solver’s existing course level MPI parallelization with Open MP (OMP) [12] fine level parallel threading. MoM problem sizes are increased to find GEMINI Solver’s limit on the DLX for hybrid parallelization.

3.3. Preliminary MPI Testing

3.3.1. Triangular Mesh Generation Using CUBIT

Cubit 14.1 [13] by Sandia National Laboratory is used to generate a triangular surface mesh for a sphere with N triangle edges. Table 2 shows the script for generating a triangular surface mesh for a 1-m radius sphere with 1.50 cm size mesh elements yielding 207,663 edges for testing a $\lambda = 15.0$ cm ($f = 2.000$ GHz) incident plane wave.

Table 2: Cubit Script to create a meshed sphere with mesh size lambda/10

```
# This example creates a meshed sphere with mesh size lambda/10.
#cubit 14.1
reset
create sphere radius 1
surface 1 size .0150
surface 1  scheme TriMesh
mesh surface 1
block 1 surface 1
save as"C:/Users/bljo222/Desktop/EIGER ANTS/Gemini
samples/sphere/spherebiggestmore.cub" overwrite
export ideas "C:\Users\bljo222\Desktop\EIGER ANTS\Gemini
samples\sphere\spherebiggestmore.unv" block all overwrite
```

Ten triangular surface meshes are generated to test GEMINI Solver's existing MPI parallelization. Table 3 shows the triangular surface meshes for a 1-m radius sphere with varying size mesh elements from 1,083 edges to 342,087 edges. Each surface mesh created by Cubit is exported as a universal ".unv" file.

Table 3: Ten triangular surface meshes for 1-m radius sphere with $\lambda^2 / \frac{1}{2}\ell^2 \approx 200$

Frequency, f (GHz)	Wavelength, λ (cm)	Mesh Element Size, ℓ (cm)	Triangle Pair Edges, N (# unknowns)
0.1499	**200**	**20.0**	**1,083**
0.2998	**100**	**10.0**	**4,455**
0.5996	**50.0**	**5.00**	**18,162**
0.8994	**33.3**	**3.33**	**41,415**
1.1992	**25.0**	**2.50**	**74,211**
1.7988	**16.7**	**1.67**	**167,652**
2.0000	**15.0**	**1.50**	**207,663**
2.3984	**12.5**	**1.25**	**298,863**
2.5000	**12.0**	**1.20**	**326,430**
2.5624	**11.7**	**1.17**	**342,087**

3.3.2. Create GEMINI Solver Input Test Files Using EIGER ANTS

EIGER ANTS [14] is used to import each mesh file and make a project suitable for exporting a GEMINI Solver ".eig" test input file. EIGER ANTS projects are developed with the following designated properties:

- Material: $\varepsilon_r = 1$ (air) for PEC or $\varepsilon_r = 4$ (slate) for Dielectric
- Frequency: 0.1499 GHz to 2.5624 GHz
- Plane Wave: $\phi_0 = 0$, $\theta_0 = 0$, $E_\phi = 0$, $E_\theta = -1$
- Far Field Scans: $\phi = 0$, $0 \le \theta \le 180$ and $\phi = 90$, $0 \le \theta \le 180$

Figure 14 shows an EIGER ANTS Project with 1.50 cm triangle mesh elements and a 2.000 GHz incident plane wave. The incident plane wave is in the -z direction and is theta-polarized.

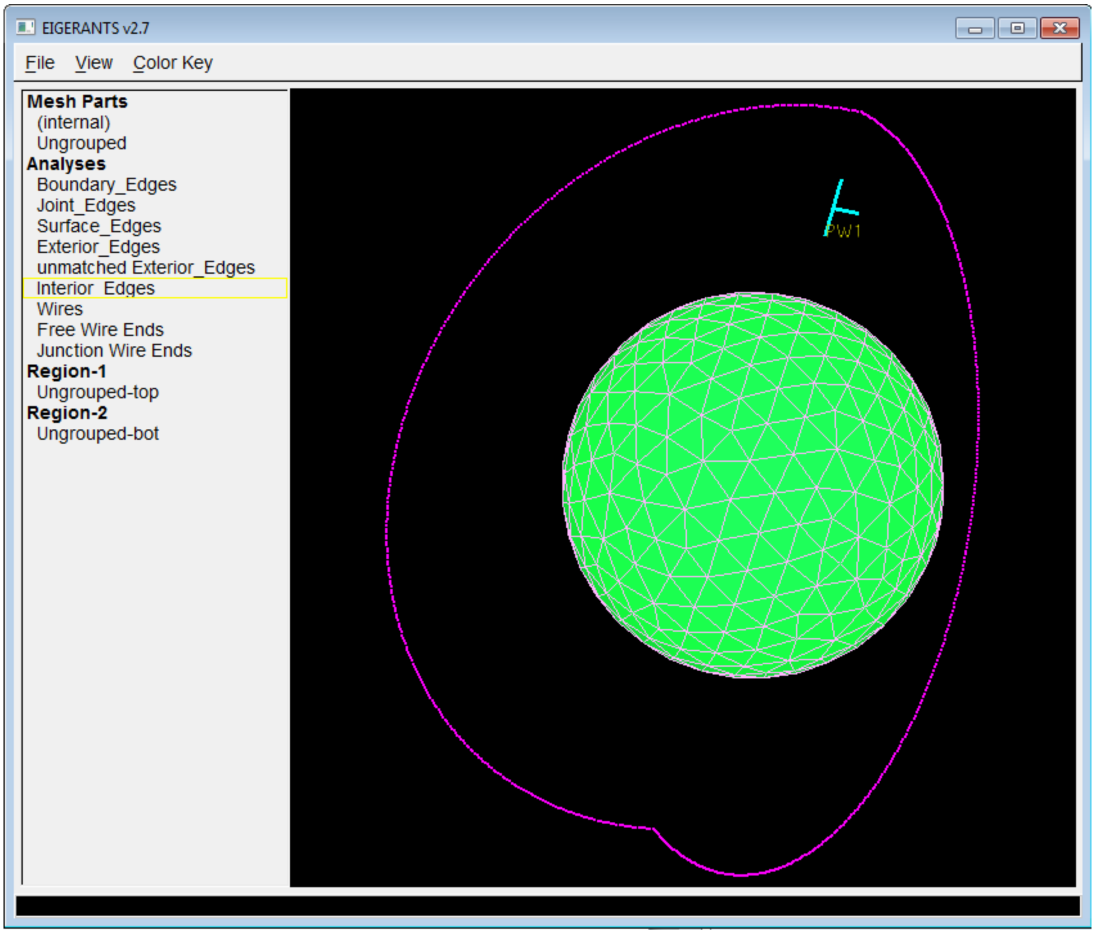


Figure 14: EIGER ANTS Project: ℓ = 1.50 cm / f = 2.000 GHz Incident Plane Wave

To complete a project, associations are made in EIGER ANTS. The finished project is exported to a ".eig" test input file. Most ".eig" input files are generated to test GEMINI Solver's EFIE solutions. A few input files are generated to test Gemini's DIELECTRIC solutions. PEC associations for EFIE solutions require (1) Outside Region association to mesh ID 1, (2) Material association to air, (3) Basis Function association to linear basis, and (4) Integral Equation association to EFIE. Dielectric associations for DIELECTRIC solutions require (1) Outside Region association to mesh ID 1, (2) Material association to air (ε_r = 1), (3) Basis Function association to linear basis, (4) Integral Equation association to DIELECTRIC, (5) Inside Region association to mesh ID 2, (6) Material association dielectric (ε_r = 4), (7) Basis Function association to linear basis, and (8) Integral Equation association to DIELECTRIC. Figure 15 shows the steps in making an EFIE solution test input file for a PEC sphere. Table 4 on next page shows the input files created to test GEMINI Solver.

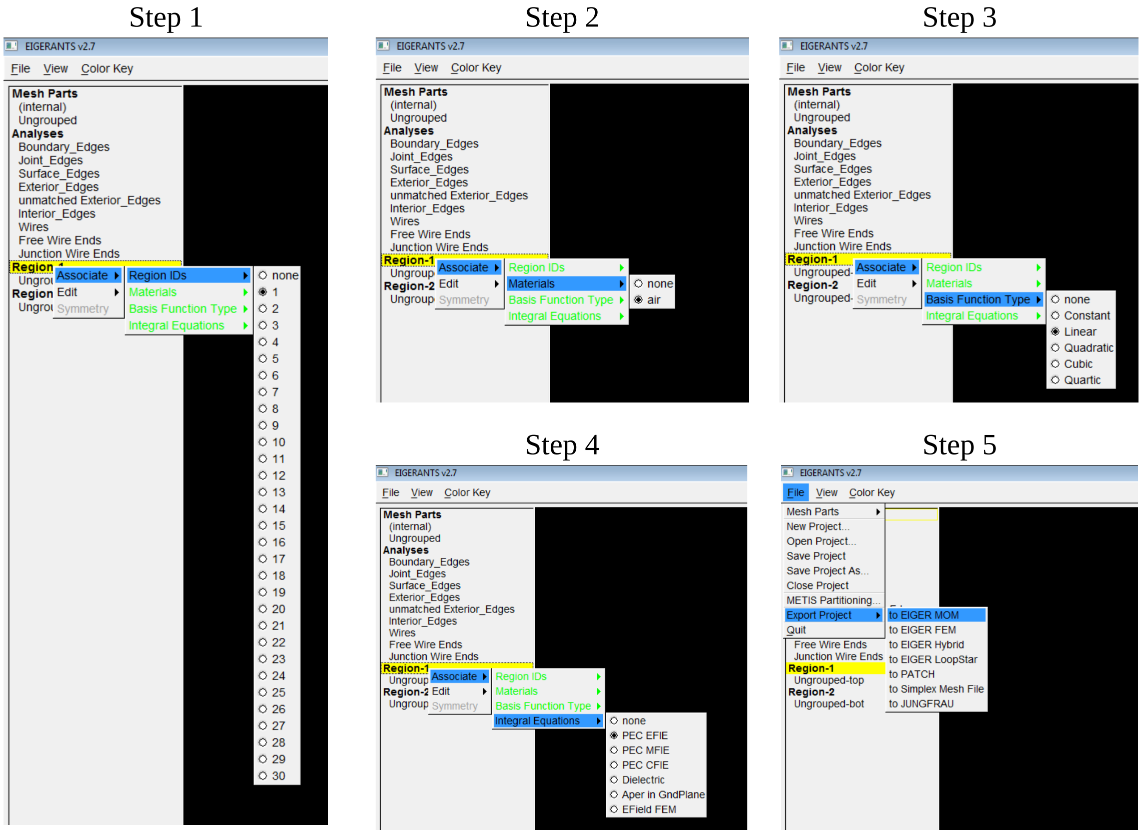

Step 1
Step 2
Step 3
Step 4
Step 5

Figure 15: Steps in making/exporting an EFIE solution test input file for a PEC sphere

Table 4: Input Files Generated to Test GEMINI Solver

Frequency, f (GHz)	Triangle Pair Edges, N (# unknowns)	Solution Test	GEMINI Solver Input Test File
0.1499	1,083	EFIE	spheresmaller0_1499GHz.eig
0.1499	2,166	DIELECTRIC	spheresmallerDIE0_1499GHz.eig
0.2998	4,455	EFIE	sphere0_2998GHz.eig
0.2998	8,910	DIELECTRIC	sphereDIE0_2998GHz.eig
0.5996	18,162	EFIE	spherebig0_5996GHz.eig
0.5996	36,324	DIELECTRIC	spherebigDIE0_5996GHz.eig
0.8994	41,415	EFIE	spherebiggerless0_8994GHz.eig
0.8994	82,830	DIELECTRIC	spherebiggerlessDIE0_8994GHz.eig
1.1992	74,211	EFIE	spherebigger1_1992GHz.eig
1.7988	167,652	EFIE	spherebiggermore1_7988GHz.eig
2.0000	207,663	EFIE	spherebiggestless2_0000GHz.eig
2.3984	298,863	EFIE	spherebiggest2_3984GHz.eig
2.5000	326,430	EFIE	spherebiggestmore2_5000GHz.eig
2.5624	342,087	EFIE	spherebiggestmost2_5624GHz.eig

3.3.3. MPI Multi-Process Test Runs on Windows-7

GEMINI Solver v1.0 (GSv1.0) is built within Visual Studio 2012 to run multiple MPI processes under Windows-7 on a Dell 8-core workstation. Table 5 lists the test cases which are run successfully. GEMINI Solver is run on each test case in Table 5 to solve for the triangular edge output currents. MPI performance comparisons are run for cases N = 4K and 18K. Testing for cases N > 42 K would not complete after seven days of processing on the Dell workstation.

Table 5: GEMINI Solver v1.0 test cases run under MPI for Windows-7

Frequency, f (GHz)	Edges, N (# unknowns)	Solution Test	# MPI Processes	MPI Performance Comparison
0.1499	1,083	EFIE	1	No
0.1499	2,166	DIELECTRIC	1	No
0.2998	4,455	EFIE	1 to 8	Yes
0.2998	8,910	DIELECTRIC	1	No
0.5996	18,162	EFIE	1 to 8	Yes
0.5996	36,324	DIELECTRIC	1	No
0.8994	41,415	EFIE	1	No

Matrix factor and fill performance comparisons are shown in Figure 16 for N=4K. As expected, the matrix factor time decreases as the number of MPI processes increases. Significant speedup is observed as the number of MPI processes increases. However, the matrix fill time is approximately constant and unexpectedly long. No speedup is observed as the number MPI processes increases.

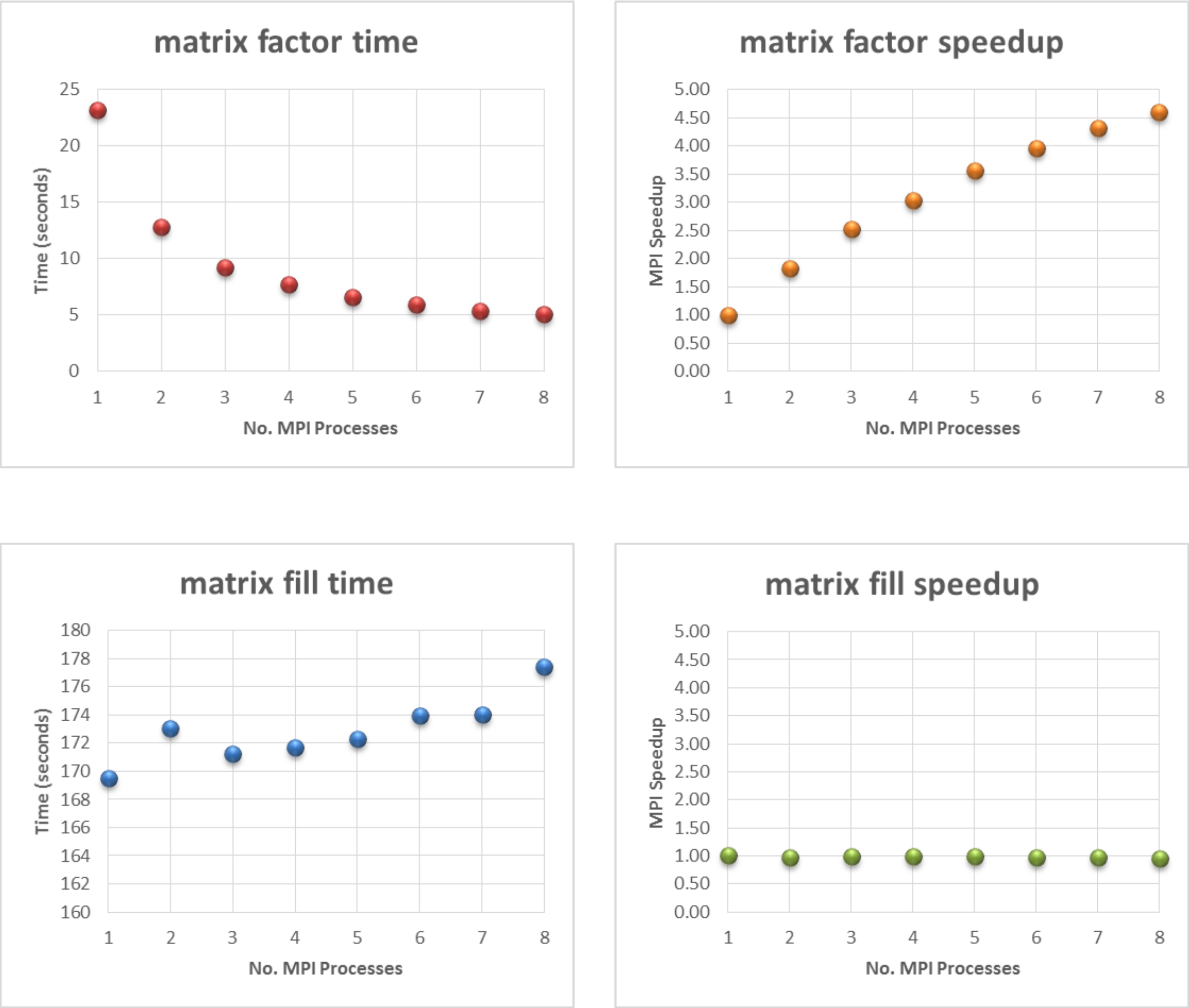


Figure 16: Matrix factor & fill performance comparisons GSv1.0 (f=0.2998GHz/N=4K)

Matrix factor and fill performance comparisons are shown in Figure 17 for N=18K. As expected, the matrix factor time decreases as the number of MPI processes increases. Significant speedup is observed as the number of MPI processes increases. However, the matrix fill time is approximately constant and unexpectedly long. No speedup is observed as the number MPI processes increases.

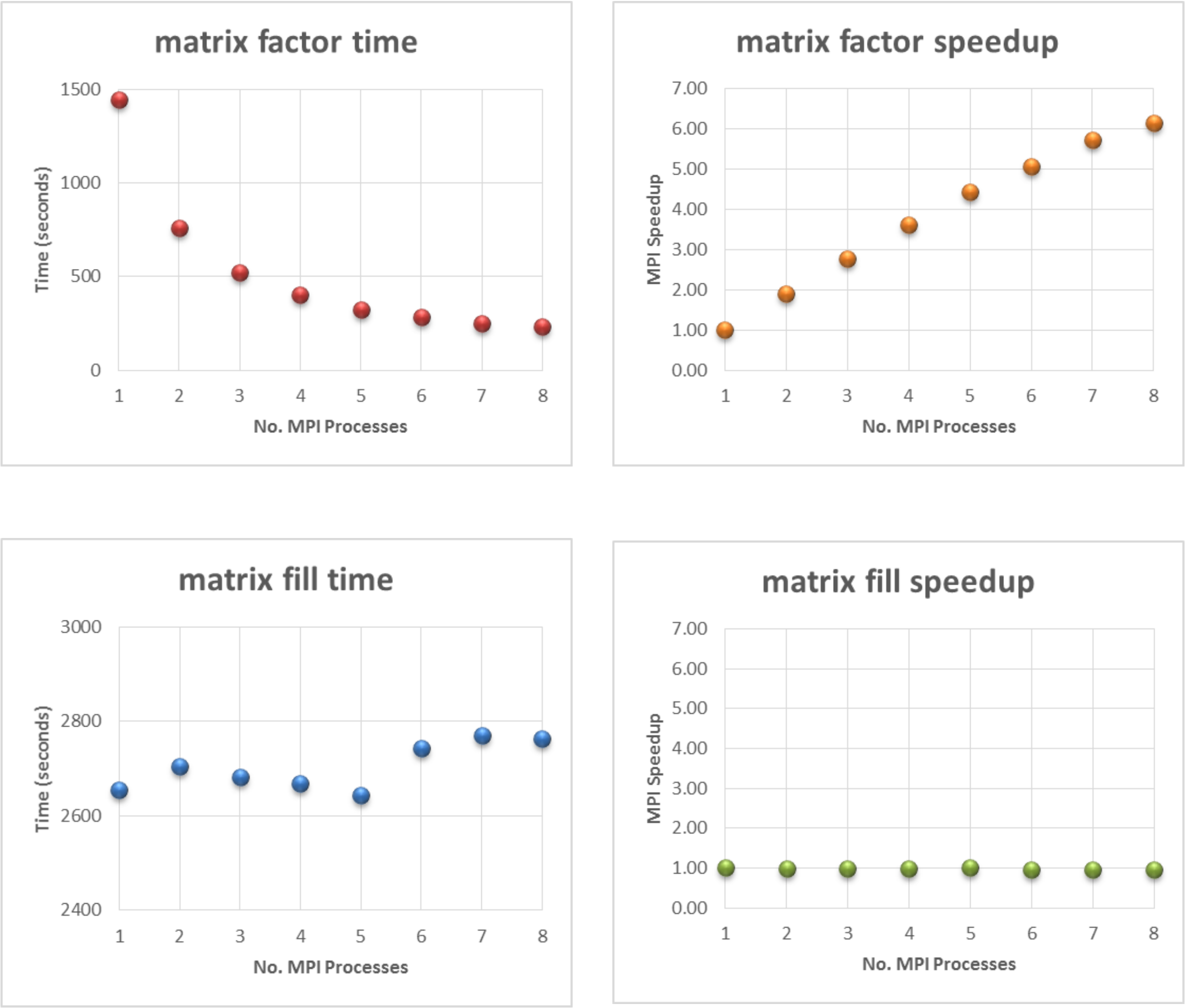


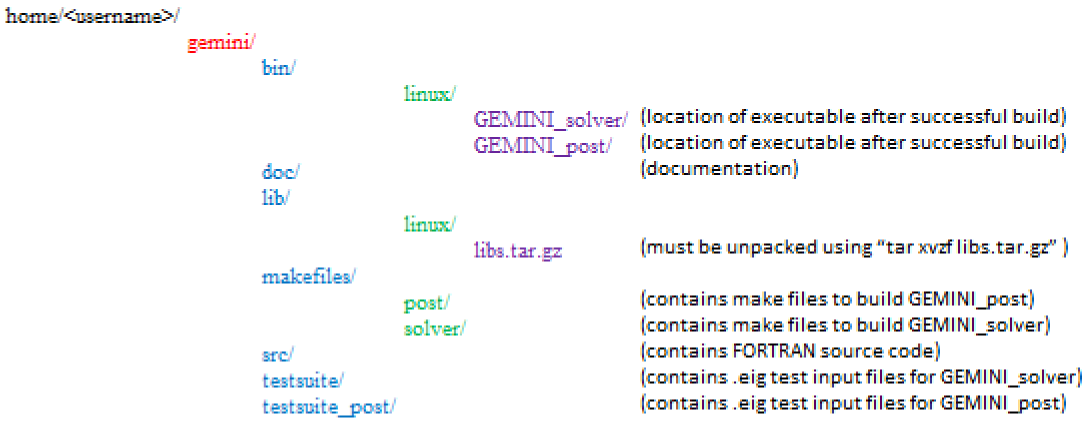
Figure 17: Matrix factor & fill performance comparisons GSv1.0 (f=0.5996GHz/N=18K)

3.3.4. Initial MPI Runtime Performance Measurements on DLX

GEMINI Solver v1.0 is subsequently built on the University of Kentucky HPC DLX Linux Cluster [11] to run larger problems requiring many more MPI processes. The steps to build GSv1.0 on DLX are shown in Table 6.

Table 6: Building GEMINI Solver on University of Kentucky HPC DLX supercluster

1. Make a directory called **gemini** on your **/home/<username>/** directory on DLX.
2. Copy the contents of the gemini folder to this new directory on DLX.
3. This should be the directory structure:



```
home/<username>/
                gemini/
                       bin/
                           linux/
                                 GEMINI_solver/  (location of executable after successful build)
                                 GEMINI_post/    (location of executable after successful build)
                       doc/                      (documentation)
                       lib/
                           linux/
                                 libs.tar.gz     (must be unpacked using "tar xvzf libs.tar.gz" )
                       makefiles/
                           post/                 (contains make files to build GEMINI_post)
                           solver/               (contains make files to build GEMINI_solver)
                       src/                      (contains FORTRAN source code)
                       testsuite/                (contains .eig test input files for GEMINI_solver)
                       testsuite_post/           (contains .eig test input files for GEMINI_post)
```

4. Unpack **libs.tar.gz** in **/home/<username>/gemini/lib/linux/** using "tar xvzf libs.tar.gz"
5. Rename the unpacked file **libISIS_juggernaut.a** to **libISIS_dlx.a**
6. Copy the two files, **makefile** and **make.options.dlx** to **/home/<username>/gemini/**
7. Make **GEMINI_solver** on DLX

Table 7 lists the test cases which run successfully. GEMINI Solver is run on each test case in Table 7 to solve for the triangular edge output currents. MPI performance comparisons are run for cases N = 41K, 83K, and 74K. Testing for N > 168 K would not complete on the DLX supercluster within three days (maximum allowed time).

Table 7: GEMINI Solver v1.0 test cases run under MPI for Linux

Frequency, f (GHz)	Edges, N (# unknowns)	Solution Test	# DLX Nodes	# MPI Processes per DLX Node	# MPI Processes	Performance Comparison
0.8994	41,415	EFIE	1,2,3	16	16,32,48	Yes
0.8994	82,830	DIELECTRIC	2,3,4,5,6	16	32,48,64,80,96	Yes
1.1992	74,211	EFIE	2,3,4,5	16	32,48,64,80	Yes
1.7988	167,652	EFIE	9	16	144	No

Matrix factor and fill performance comparisons are shown in Figure 18 for N=41K. As expected, the matrix factor time decreases as the number of MPI processes increases. Significant speedup is observed as the number of MPI processes increases. As the number of MPI processes is doubled (16→32) and tripled (16→48), the speedup is likewise increased by a factor of approximately two and three, respectively. The matrix fill time is approximately constant and unexpectedly long. No speedup is observed as the number of MPI processes increases.

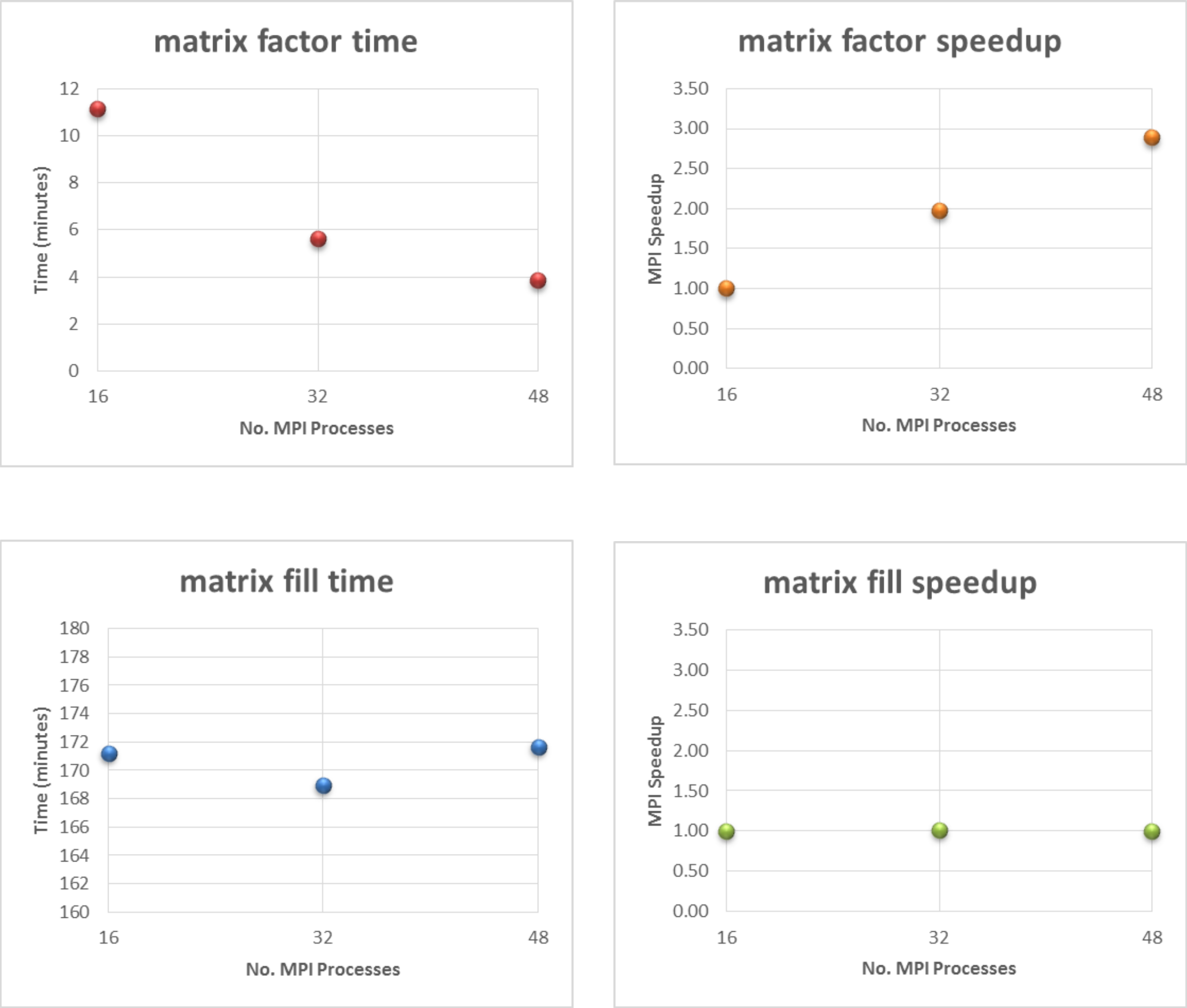


Figure 18: Matrix factor & fill performance comparisons GSv1.0 (f=0.8994GHz/N=41K)

Matrix factor performance comparisons are shown in Figure 19 for N=83K. As expected, the matrix factor time decreases as the number of MPI processes increases. Significant speedup is observed as the number of MPI processes increases. As the number of MPI processes is doubled (32→64) and tripled (32→96), the speedup is likewise increased by a factor of approximately two and three, respectively. The matrix fill time is approximately constant and unexpectedly long. No speedup is observed as the number MPI processes increases.

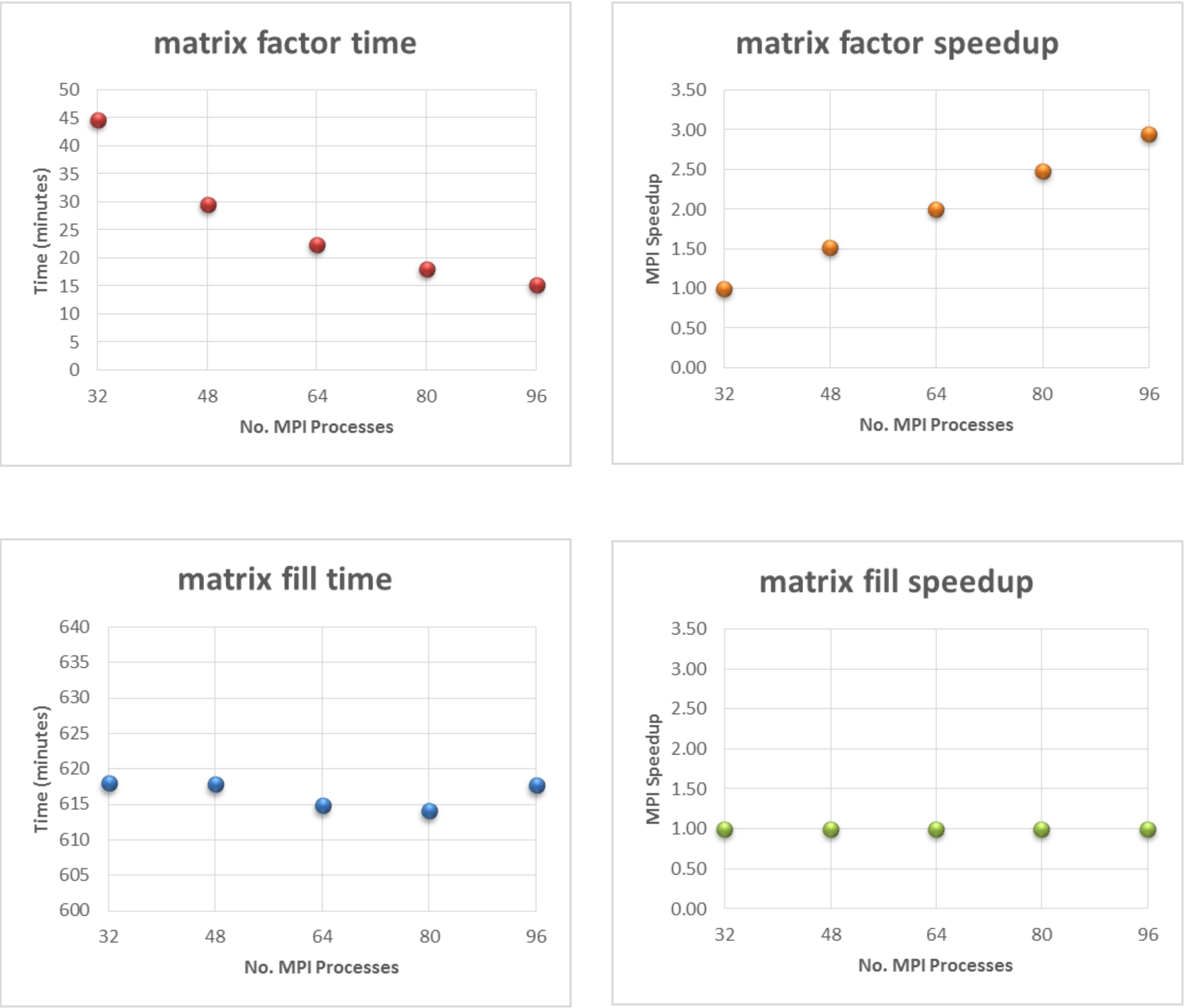


Figure 19: Matrix factor & fill performance comparisons GSv1.0 (f=0.8994GHz/N=83K)

Matrix factor and fill performance comparisons are shown in Figure 20 for N=74K. As expected, the matrix factor time decreases as the number of MPI processes increases. Significant speedup is observed as the number of MPI processes increases. As the number of MPI processes is doubled (32→64), the speedup is likewise increased by a factor of approximately two. The matrix fill time is approximately constant and unexpectedly long. No speedup is observed as the number MPI processes increases.

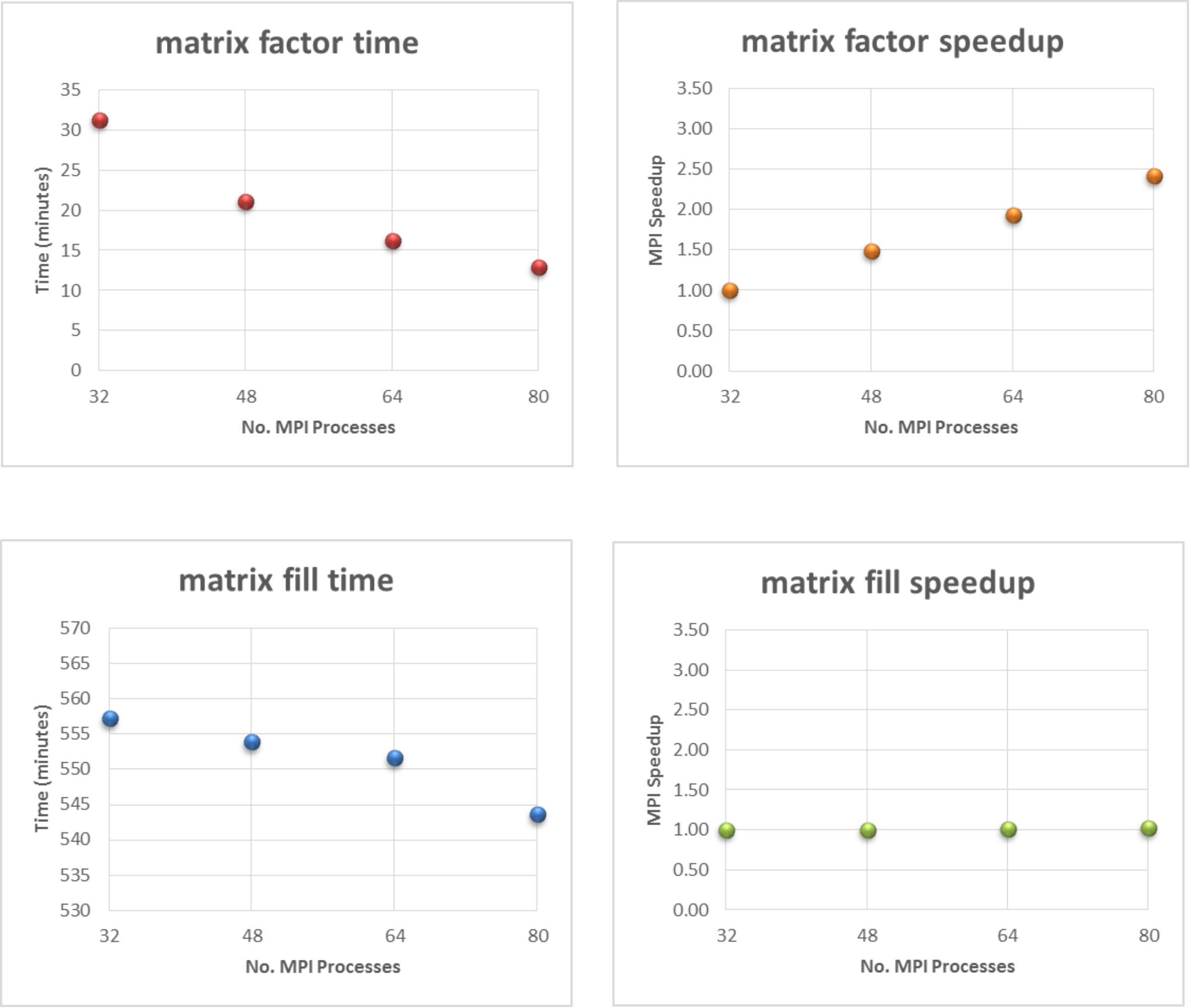


Figure 20: Matrix factor & fill performance comparisons GSv1.0 (f=1.1992GHz/N =74K)

For the single N=168 K run with 144 MPI process, the factor time is reasonable at 1.82 hours, but the fill time is too long. The implementation of MPI parallelization in GEMINI Solver v1.0 successfully increases the matrix factorization performance, but has no observable effect on the matrix fill performance. Matrix factor and fill performance results are presented to Nathan Champagne (author of GEMINI Solver) at the CEM group at NASA in Houston to guide in improving GEMINI Solver's performance. An improved version is subsequently developed for further testing.

3.3.5. Improved MPI Runtime Performance Measurements on DLX

Several issues including long project load time and long, constant matrix fill time are solved with the improved GEMINI Solver v2.0 (GSv2.0). GEMINI Solver is run on each test case in Table 8 to solve for the triangular edge output currents. Matrix fill times are significantly reduced with the improved version. Testing for N > 208 K would not complete on the DLX due to excessive page swapping. The total memory required for runs with N > 208 K exceeds the physical memory limit on each DLX node due to the very large problem sizes (mesh sizes).

Table 8: GEMINI Solver v2.0 test cases run under MPI for Linux

Frequency, f (GHz)	Edges, N (# unknowns)	Solution Test	# DLX Nodes	# MPI Processes per DLX Node	# MPI Processes	Performance Comparison
0.8994	41,415	EFIE	4,5,6	16	64,80,96	Yes
0.8994	41,415	EFIE	7,8,9	16	112,128,144	
0.8994	82,830	DIELECTRIC	6,7	16	96,112	Yes
0.8994	82,830	DIELECTRIC	8,9	16	128,144	
1.1992	74,211	EFIE	4,5,6	16	64,80,96	Yes
1.1992	74,211	EFIE	7,8,9	16	112,128,144	
1.7988	167,652	EFIE	9,10	16	144,160	Yes
1.7988	167,652	EFIE	11,12	16	176,192	
2.0000	207,663	EFIE	14,16	16	224,256	Special
2.0000	207,663	EFIE	28	8	224	

Matrix factor and fill performance comparisons are shown in Figure 21 for N=41K. As expected, the matrix factor time decreases as the number of MPI processes increases. Significant speedup is observed as the number of MPI processes increases. The matrix fill time now decreases as the number of MPI processes increases. Speedup is now observed as the number of MPI processes increases.

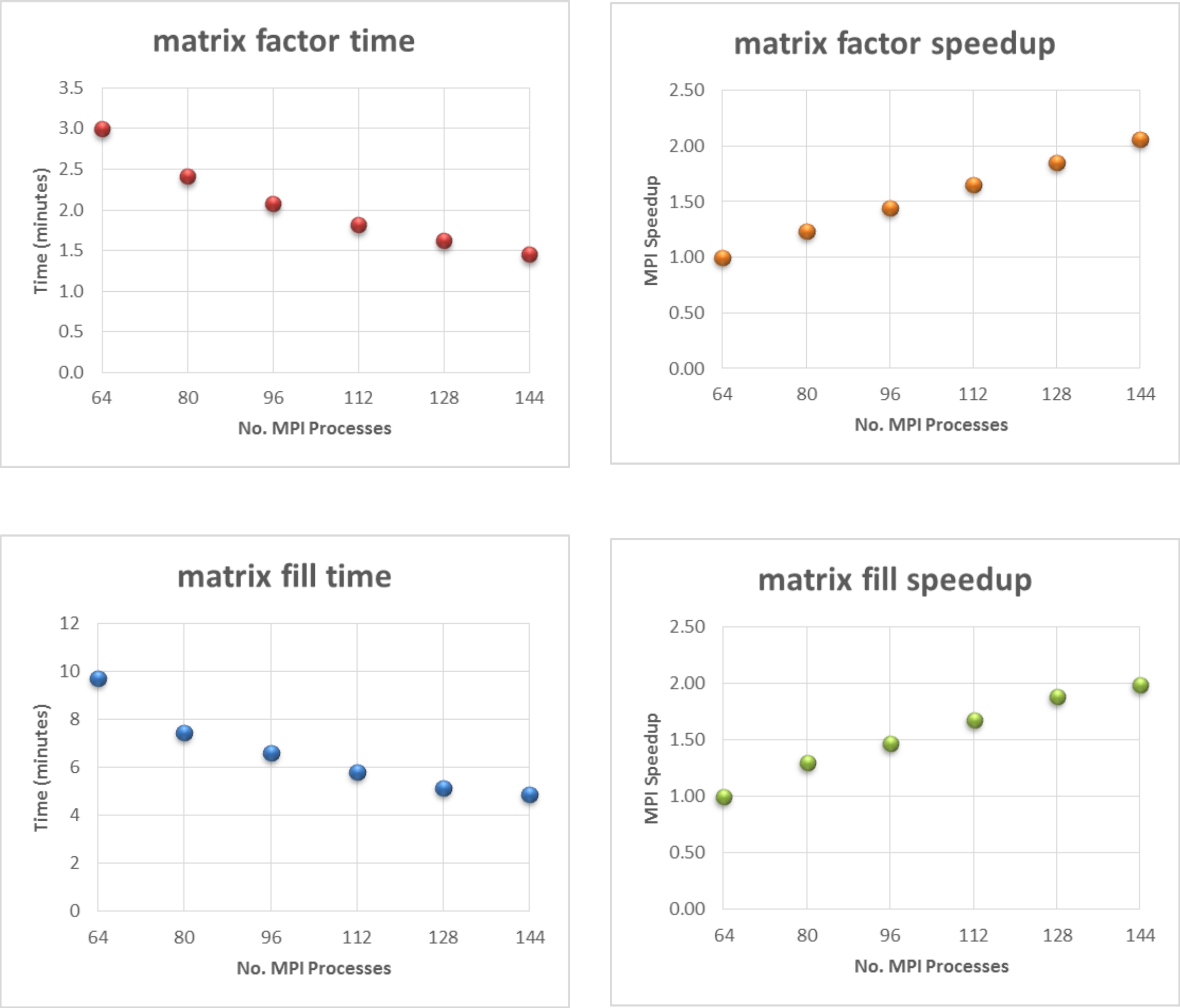


Figure 21: Matrix factor & fill performance comparisons GSv2.0 (f=0.8994GHz/N=41K)

Matrix factor and fill performance comparisons are shown in Figure 22 for N=83K. As expected, the matrix factor time decreases as the number of MPI processes increases. Significant speedup is observed as the number of MPI processes increases. The matrix fill time now decreases as the number of MPI processes increases. Speedup is now observed as the number of MPI processes increases.

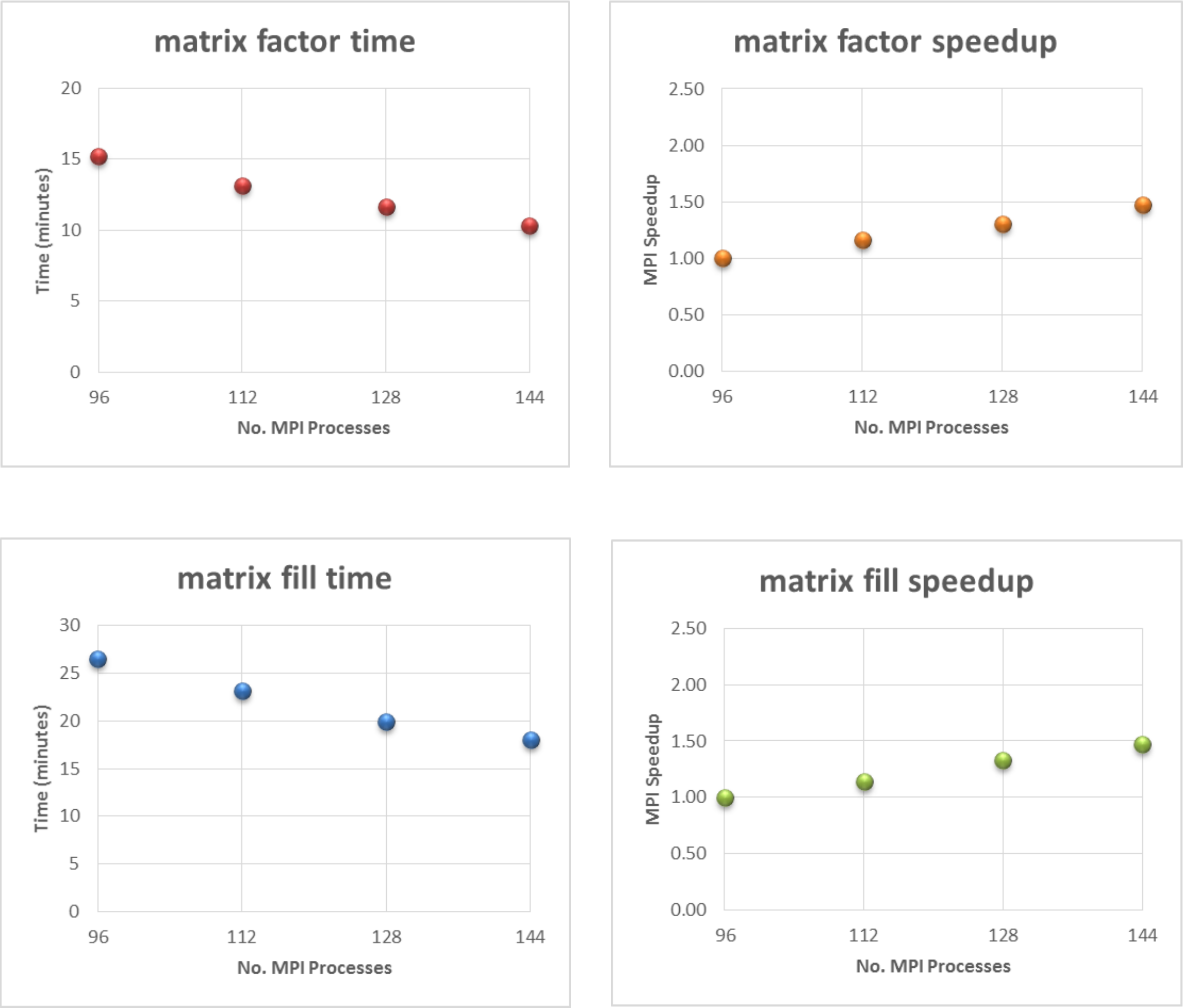


Figure 22: Matrix factor & fill performance comparisons GSv2.0 (f=0.8994GHz/N=83K)

Matrix factor and fill performance comparisons are shown in Figure 23 for N=74K. As expected, the matrix factor time decreases as the number of MPI processes increases. Significant speedup is observed as the number of MPI processes increases. The matrix fill time now decreases as the number of MPI processes increases. Speedup is now observed as the number of MPI processes increases.

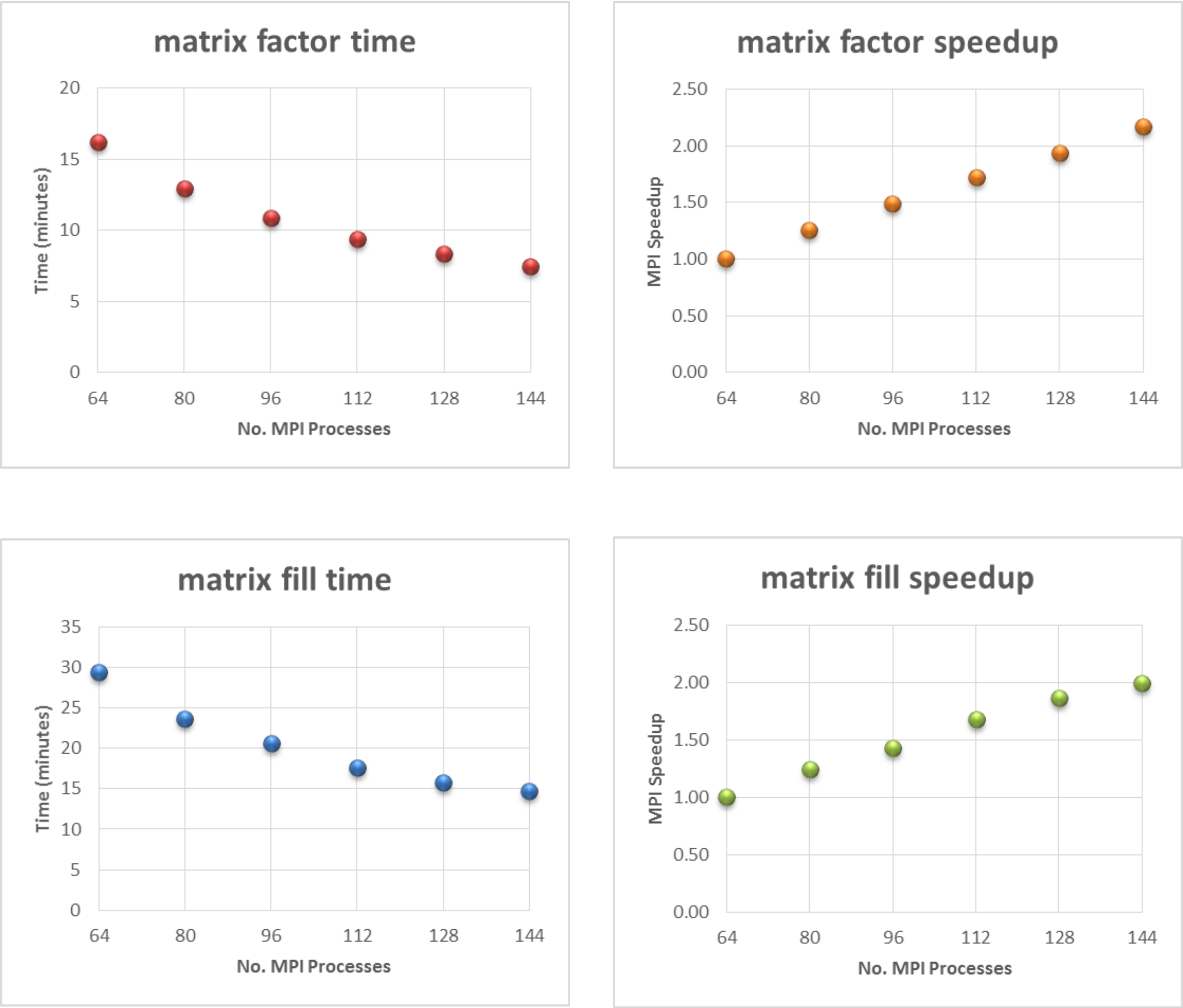


Figure 23: Matrix factor & fill performance comparisons GSv2.0 (f=1.1992GHz/N=74K)

Matrix factor and fill performance comparisons are shown in Figure 24 for N=168K. As expected, the matrix factor time decreases as the number of MPI processes increases. Significant speedup is observed as the number of MPI processes increases. In addition, the matrix fill time decreases as the number of MPI processes increases. Speedup is observed as the number of MPI processes increases.

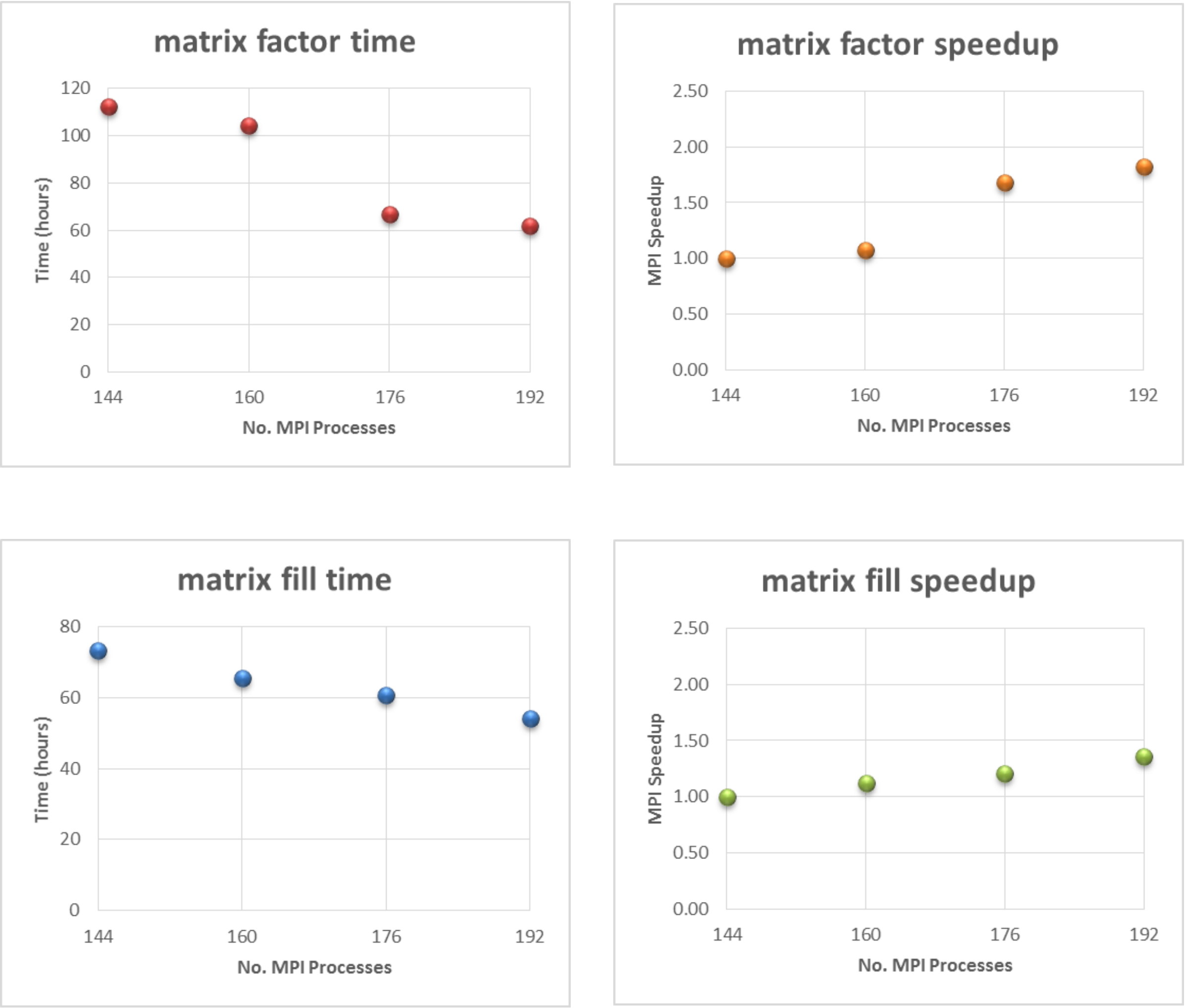


Figure 24: Matrix factor & fill performance comparisons GSv2.0 (f=1.7988GHz/N=168K)

Matrix factor and fill performance comparisons are shown in Figure 25 for N=208K. The matrix factor time varies little as the number of MPI processes increases. The matrix fill time decreases as the number of MPI processes increases. Speedup is observed as the number of MPI processes increases, but not as significantly as for previous test cases. The large mesh size for N=208K could be a limiting factor in performance for 224, 256, etc. MPI processes. Since the mesh is duplicated for each MPI process running on a node, reducing the number of processes per node while keeping the total number of MPI processes constant should decrease the matrix factor and fill times.

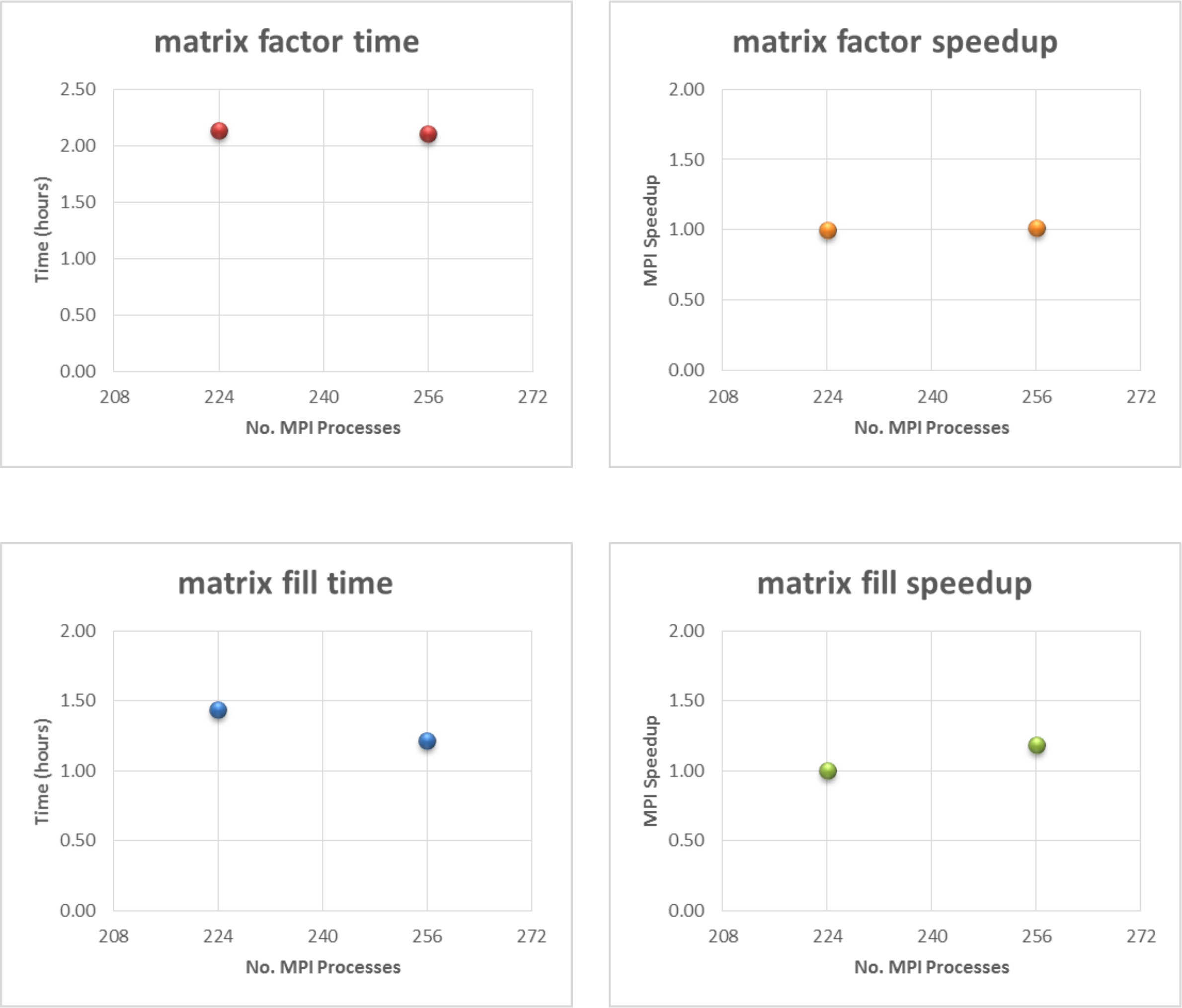


Figure 25: Matrix factor & fill performance comparisons GSv2.0 (f=2.0000GHz/N=208K)

Matrix factor and fill performance comparisons are shown in Figure 26 for N=208K using 224 MPI processes for 8 and 16 MPI processes per node. As suspected, while holding the total number of MPI processes constant, reducing the number of MPI process per node reduces both the matrix fill and factor times. Therefore, higher performance can be achieved for larger problems when the number of MPI processes per node is reduced.

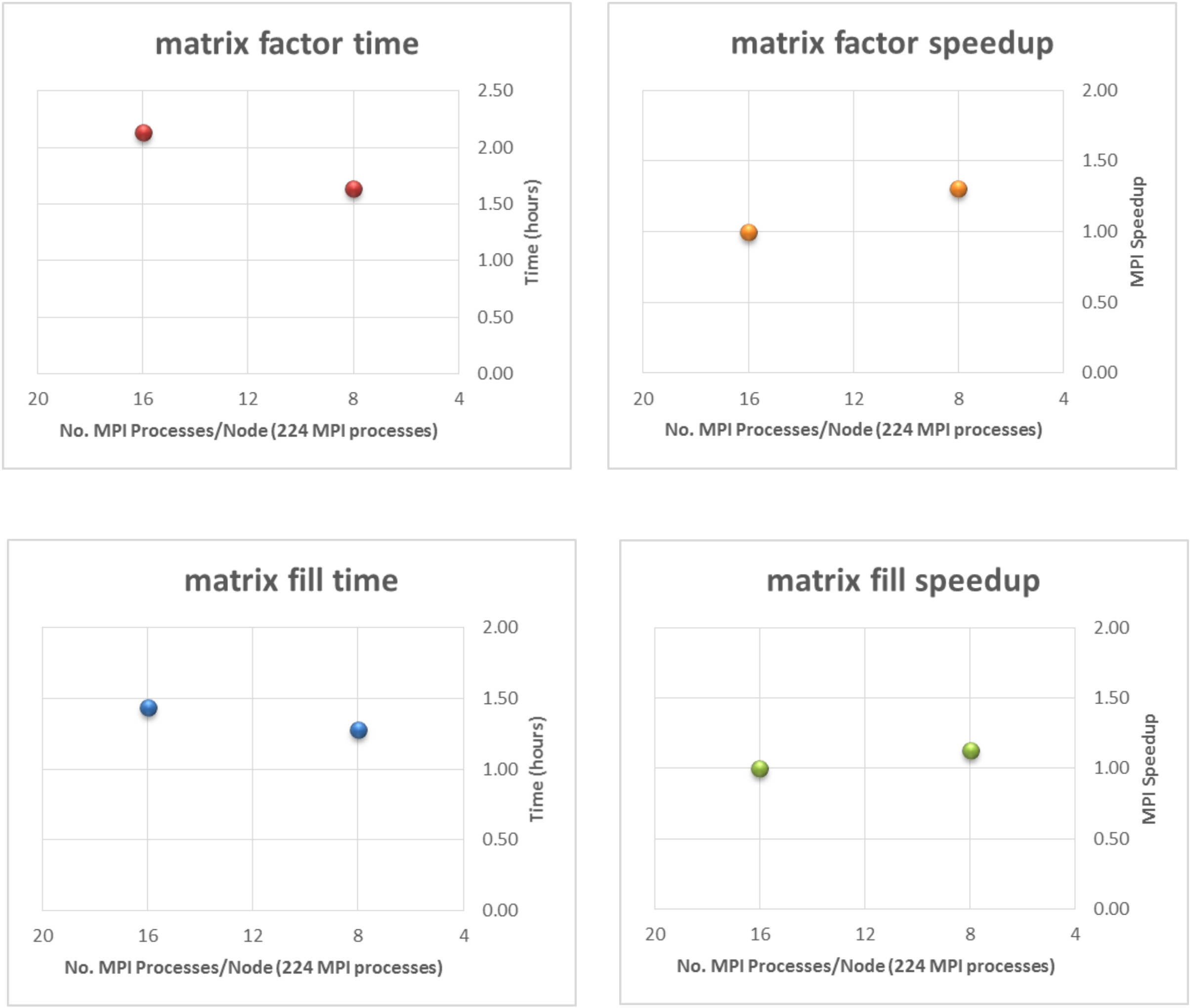


Figure 26: Matrix factor & fill performance by MPIs/node GSv2.0 (f=2.0000GHz/N=208K)

The cases for N > 208K will not run on DLX with 16 MPI processes per node due to excessive page swapping. However, reducing the number of MPI processes from 16 to 8 (or less) will free up memory per node which should allow larger problems to execute. On the down side, efficiency suffers because cores not executing an MPI process are idle. On a supercluster such as DLX, this waste cannot be permitted (just ask Jerry Grooms!). To address the inefficiency problem, multiple threading of each MPI process can make use of otherwise idle cores. For example, using four MPI processes per node with four threads per MPI process would use all sixteen cores per node, allowing larger problems to execute with shorter matrix fill and factor times. See diagram in Figure 27.

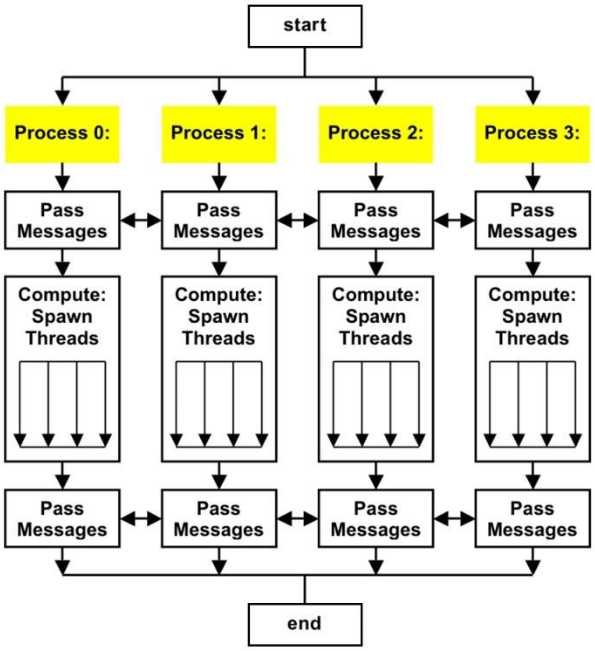


Figure 27: Multi-threading of MPI processes

Open MP (OMP) threading of the impedance matrix fill routine in GEMINI Solver will be explored in the next chapter. Briefly, OMP threads can be created to parallelize loop routines within the matrix fill routine of each MPI process. MPI currently provides course parallelization (process level) while OMP can provide fine parallelization (thread level) of the GEMINI Solver matrix fill routine. Other methods utilizing the MFDlib library of data sparse methods have been developed to improve the GEMINI Solver matrix factorization beyond the scope of this work [4].

3.4. GEMINI Post RCS Measurements

GEMINI Post v2.0 (GPv2.0) is built for Windows-7 on the Dell workstation as well as on the University of Kentucky HPC Linux DLX Cluster. Gemini Post generates the bistatic cross-section (RCS) pattern using eq. (19) for the solved currents of all test cases discussed in sections 3.3.3, 3.3.4, and 3.3.5. Each RCS pattern is compared to the Mie Series solution. The Mie Series is an analytical series used to calculate the RCS scattered solution for a plane wave incident on a sphere. The Mie Series solution is given by

$$\sigma_\theta(\theta',\phi') = \frac{4\pi}{k_0^2}\left|S_1(\theta')\right|^2 \cos^2\phi' \qquad \sigma_\phi(\theta',\phi') = \frac{4\pi}{k_0^2}\left|S_2(\theta')\right|^2 \sin^2\phi' \tag{22}$$

where $S_1(\theta')$ and $S_2(\theta')$ are the complex scattering amplitudes [15]. Often for purposes of illustration, one specifies the cases where $\phi' = 0$ and $\phi' = \pi/2$, giving

$$\sigma_\theta(\theta',0) = \frac{4\pi}{k_0^2}\left|S_1(\theta')\right|^2 = \sigma_{VV} \qquad \sigma_\phi(\theta',0) = 0 \tag{23}$$

$$\sigma_\theta\left(\theta',\frac{\pi}{2}\right) = 0 \qquad \sigma_\phi\left(\theta',\frac{\pi}{2}\right) = \frac{4\pi}{k_0^2}\left|S_2(\theta')\right|^2 = \sigma_{HH} \tag{24}$$

Hence the quantities sought are the magnitudes $\left|S_1(\theta')\right|$ and $\left|S_2(\theta')\right|$. Computation of the bistatic cross-sections using the Mie Series is performed using Walton C. Gibson's *MieScattered* MATLAB program [16]. The magnitudes of the complex scattering amplitudes $|S_1(\theta')|$ and $|S_2(\theta')|$ are calculated by *MieScattered* to the optimal value of terms N_{max}. The optimal value N_{max} is reached when further terms only improve the Mie Series solution by 0.01%. Figure 28 on the next page shows two plots of the GEMINI Post RCS results with fits to the Mie Series for EFIE and Dielectric solutions for a 0.1499 GHz plane wave incident on a 1-m sphere with a triangle surface mesh.

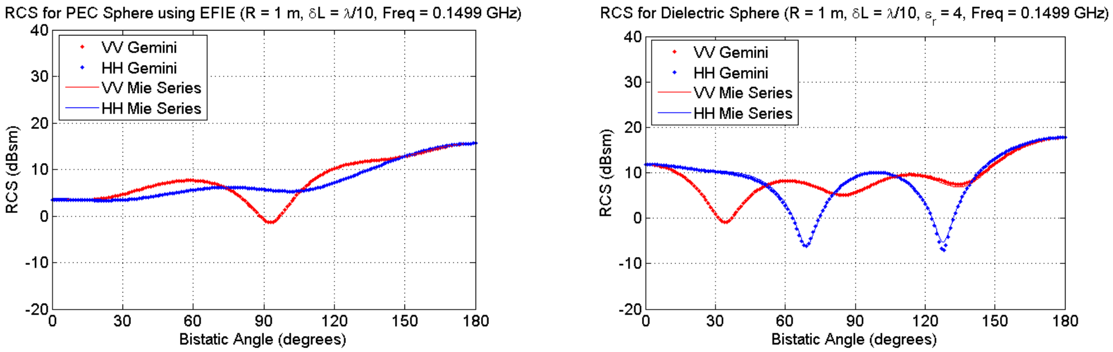


Figure 28: GPv2.0 EFIE & Dielectric RCS results w/fit to Mie Series for f = 0.1499 GHz

Comparisons of the RCS pattern generated by GEMINI Post are made to the Mie Series using the Chi-Square goodness-of-fit (GOF) [15] given by:

$$\chi^2 = \frac{\sum \left(y_i - y_i^*\right)^2}{\sigma^2\left(y_i^*\right)} \qquad (25)$$

where y_i represents GEMINI Post RCS values, and y_i^* and $\sigma^2\left(y_i^*\right)$ represent the Mie Series values and variance, respectively. Table 9 displays the goodness-of-fit (GOF) between GEMINI Post RCS values and the Mie Series using N_{max} terms [16]. The bistatic angle resolution is given by Δangle.

Table 9: GEMINI Post RCS results fit to Mie Series

Frequency, f (GHz)	Edges, N (# unknowns)	Solution Test	Quality of Fit	Nmax Terms	Δangle (°)	$\sigma_{VV}\ \chi^2$ GOF	$\sigma_{HH}\ \chi^2$ GOF
0.1499	1,083	EFIE	Excellent	4	1	0.103	0.080
0.1499	2,166	DIELECTRIC	Good	4	1	0.475	0.351
0.2998	4,455	EFIE	Excellent	7	1	0.008	0.007
0.2998	8,910	DIELECTRIC	Good	7	1	0.204	0.321
0.5996	18,162	EFIE	Excellent	13	1	0.009	0.010
0.5996	36,324	DIELECTRIC	Fair	13	1	2.123	2.308
0.8994	41,415	EFIE	Excellent	19	1	0.013	0.015
0.8994	82,830	DIELECTRIC	Poor	19	1	4.115	4.604
1.1992	74,211	EFIE	Excellent	25	1	0.018	0.019
1.7988	167,652	EFIE	Excellent	38	1	0.026	0.025
2.0000	207,663	EFIE	Excellent	42	0.5	0.055	0.054

Appendix A contains the plots and goodness-of-fit statistics for all cases listed in Table 9.

Chapter 4. Hybrid Parallelization: Combining Open MP with MPI

4.1. Why Open MP Multi-Threading?

In this chapter hybrid parallelization is employed to decrease impedance matrix fill time and increase the problem size potential by reducing the number of MPI processes running on a node while utilizing all cores. Hybrid parallelization incorporates Open MP (OMP) multi-threading within MPI processes. Hybrid parallelization is most efficient when MPI processes work on a course level of parallelism and OMP is used with the shared address space of each MPI process for additional fine-grained parallelization [17]. OMP enables the creation of shared-memory parallel threads within a program or process. A thread is a runtime element that can execute a stream of instructions independently [18]. When the operating system creates a process, such as an MPI process, to execute a program, such as GEMINI Solver, it will allocate resources to that process. If multiple threads work together to execute a program, they will share these resources, including the address space of the associated process. OMP offers a structured approach to multi-threaded programming utilizing the fork-join programming model illustrated in Figure 29. In this approach, the program starts a single thread of execution referred to as the initial thread. If the initial thread encounters an OMP parallel construct (fork), it creates a team of collaborating threads, becomes the master of the team, and

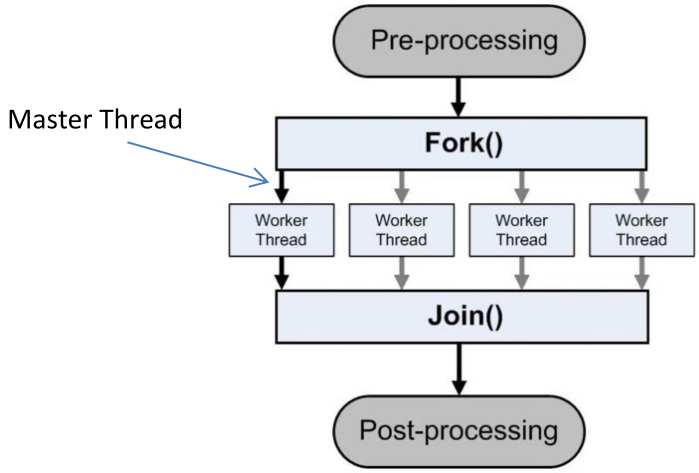


Figure 29: The Open MP fork-join programming model

works with the other team members to execute the code dynamically. At the end of the parallel construct (join), only the initial thread, or master thread, continues; all other threads terminate. Any portion of the program enclosed by a parallel construct (fork-join region) is known as a parallel region.

Consider four MPI processes running on a node threaded with four OMP threads per each MPI process. Figure 31 on the following page illustrates the 4-MPI / 4-OMP hybrid parallelization model for a node. Instead of sixteen copies of the mesh, only four copies reside in node memory, one in the shared memory of each MPI process. Each MPI process has four OMP threads working in parallel and sharing memory to accomplish many times more work than a single MPI process with no threading. Although only four MPI processes execute on the node, all sixteen cores are busy, increasing efficiency.

OMP threading is well suited for parallelizing the loop structure found within the impedance matrix fill routine of GEMINI Solver. Figure 30 shows a simplified diagram of the matrix fill loop structure used in the impedance matrix fill routine.

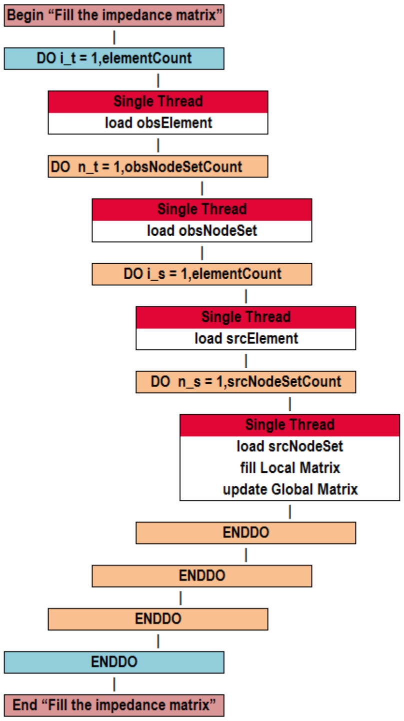


Figure 30: Loop structure of the GEMINI Solver impedance matrix fills routine

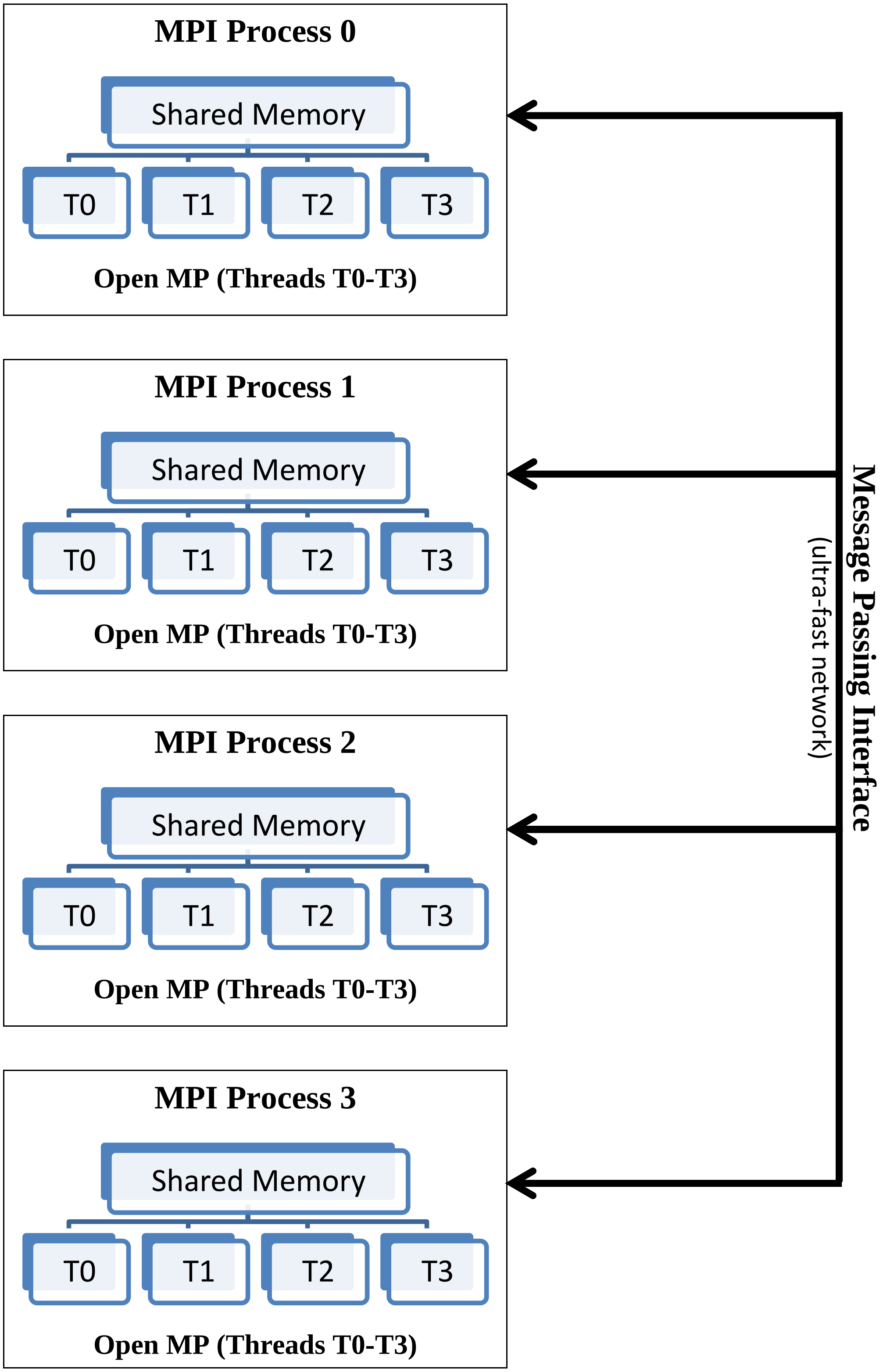


Figure 31: Hybrid Parallelization Programming: Combining MPI/Open MP

Consider the EFIE test case with N = 1083 (f = 0.1449 GHz) from Table 4 which has a mesh size of 722 triangular elements. In executing the outermost loop of the impedance matrix fill routine, 722 loop iterations (elementCount) are performed serially. During an outer loop iteration, the selected observation element is used with each of 722 source elements to compute and fill 722 local matrices. In turn, each local matrix is used to update the global matrix. Because iterations of the outer loop can be made independent of each other, OMP threading can be successfully implemented. Loop independence requires the results of one loop iteration do not depend on the results of any other loop iteration; otherwise a data race condition may occur. A data race condition arises when two or more threads access the same shared variable without any synchronization to order the accesses, and at least one of the accesses is a write [18]. A quick check for independence can be made by executing the outer loop backwards and obtaining the same result [19]. GEMINI Solver's matrix fill outer loop passes the independence test with modifications discussed later. Assuming loop independence, the 722 iterations can be divided into parallel groups with as little as two threads or as many as sixteen threads. OMP threads working in parallel can accomplish many more iterations in a given time than serial processing with no threading. Table 10 shows the number of triangular elements in a mesh and the estimated number of loop iterations performed by each thread per number of OMP threads nt = 2, 4, 8, 16 for the EFIE cases listed in Table 4.

Table 10: Loop iterations per thread for nt = 2, 4, 8, and 16 OMP threads

Frequency, f (GHz)	Edges, N (# unknowns)	# Triangular elements	Loop Iterations Per Thread			
			nt = 2	nt = 4	nt = 8	nt = 16
0.1499	1,083	722	361	181	91	46
0.2998	4,455	2,970	1,485	743	372	186
0.5996	18,162	12,108	6,054	3,027	1,514	757
0.8994	41,415	27,610	13,805	6,903	3,452	1,726
1.1992	74,211	49,474	24,737	12,369	6,185	3,093
1.7988	167,652	111,768	55,884	27,942	13,971	6,986
2.0000	207,663	138,442	69,221	34,611	17,306	8,653
2.3984	298,863	199,242	99,621	49811	24,906	12,453
2.5000	326,430	217,620	108,810	54,405	27,203	13,602
2.5624	342,087	228,058	114,029	57,015	28,508	14,254

Generally as the number of threads increase, parallel sharing increases and the time to execute the outer loop decreases.

Envision implementing OMP threading in the loop structure used to fill the impedance matrix. Figure 32 shows OMP threading of the outer loop of the impedance matrix fill routine with nt = 4. The single threading of Figure 30 has been

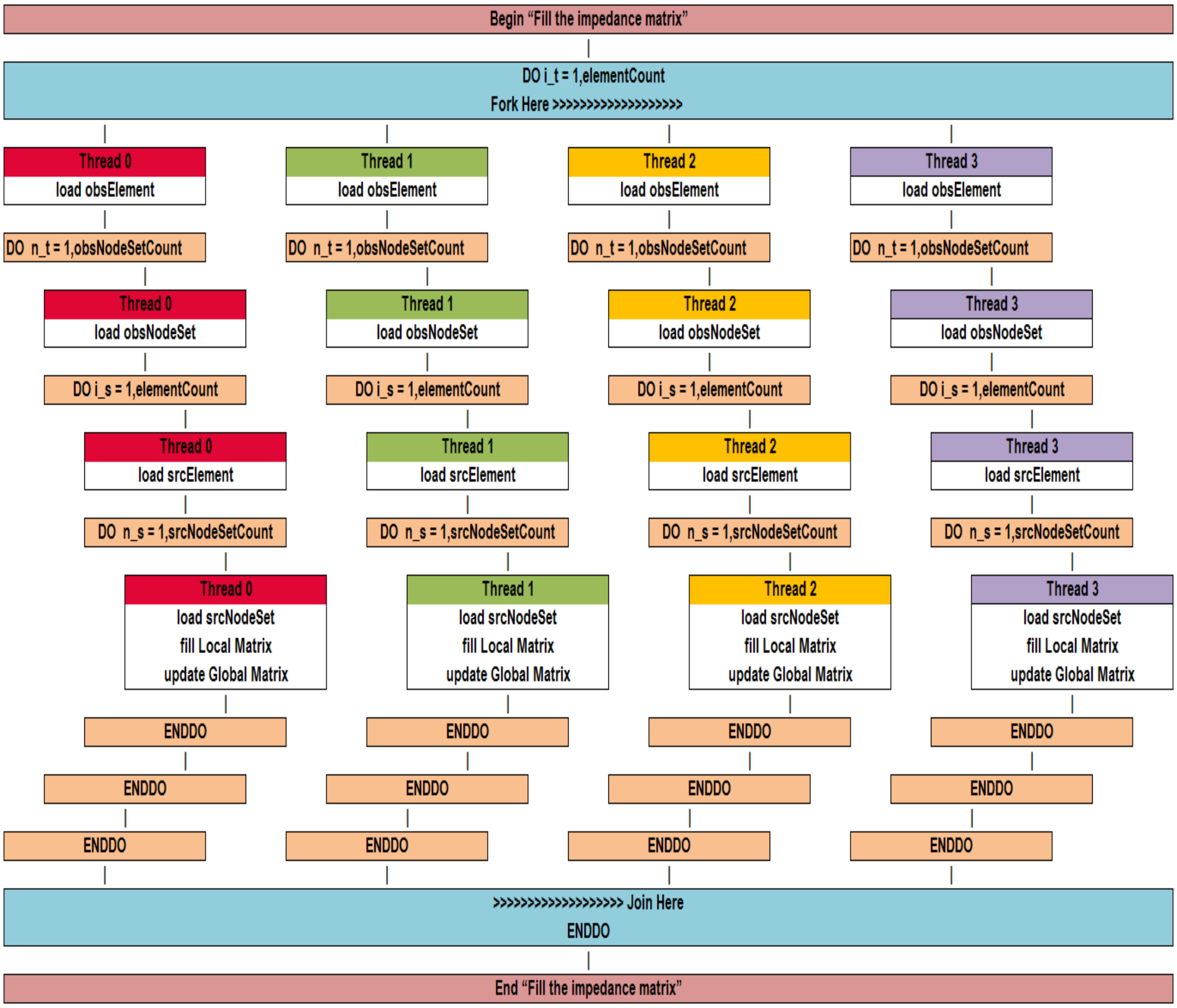


Figure 32: OMP parallelization of impedance matrix fill routine

replaced with multiple threading. For the first case in Table 10, each OMP thread works independently on a subset of about 181 iterations. Since the threads work in parallel, the outer loop is effectively reduced from 722 iterations to 181 iterations, thus reducing the execution time. However, execution time reduction also depends on "thread overhead cost" and "load balance". Generally, the reduction in execution time of the outer loop depends on three factors:

- number of threads: nt =2, 4, 8 or 16
- thread overhead cost: time cost related to creating and maintaining threads
- load balance: balance of workload among threads

Increasing the number of threads typically increases loop performance and thereby reduces execution time. However, increasing the number of threads also increases the overhead cost which reduces loop performance and lengthens execution time. Furthermore, if the workload is not evenly balanced among the team of threads, some threads can be idle while waiting for others to finish working, leading to inefficient execution and longer than needed execution times. Therefore, optimizing execution time requires a balance between the number of threads created, thread overhead cost, and load balance.

Speedup measures the ratio of execution time for single threading to execution time for multi-threading given by [18]:

$$S = \frac{T_1}{T_P} \tag{26}$$

where T_1 and T_p are the execution (wall clock) times required to perform loop iterations with one thread (sequential or serial processing) and p threads (parallel processing), respectively. Parallel efficiency measures the reduction in execution time per thread for multi-threading compared to execution time for single threading given by:

$$E = \frac{S}{p} \tag{27}$$

where S is the speedup and p is the number of threads.

4.2. OMP Directives and Parallelization

OMP consists of a set of compiler directives, runtime library routines, and environment variables to enable the creation of shared-memory parallel threads within a FORTRAN 2000 program. An OMP directive is a specially formatted comment statement beginning with “!$OMP” which generally applies to the executable code immediately following it in a program [20]. OMP directives provide the means for the programmer to: create teams of threads for parallel execution, specify how to share work among the members of a team, declare both shared and private variables, and synchronize threads and enable them to perform certain operations exclusively. Table 11 shows a few common OMP directives [21].

Table 11: A few common OMP directives

Create a team of threads which execute the enclosed code in parallel

```
!$OMP PARALLEL
< code to execute in parallel >
!$OMP END PARALLEL
```

Direct team of threads to execute iterations of the enclosed loop code in parallel

```
!$OMP DO
< loop code to execute in parallel >
!$OMP END DO
```

Create a team of threads and then direct team to execute iterations of the enclosed loop code in parallel

```
!$OMP PARALLEL DO
< loop code to execute in parallel >
!$OMP END PARALLEL DO
```

Restricts execution of the enclosed code to only one thread in the team

```
!$OMP SINGLE
< code to be executed by only one thread in the team >
!$OMP END SINGLE
```

Restricts execution of the enclosed code to one thread at a time

```
!$OMP CRITICAL
< code to be executed by one thread at a time >
!$OMP END CRITAL
```

OMP directives will be illustrated in the examples of the next two sections.

4.2.1. Using OMP on a Simple Printing Operation

In the FORTRAN 2000 program shown in Table 12, the **!$OMP PARALLEL** directive creates a team of threads which execute in parallel. Each thread displays its own thread identification (TID) asynchronously and then one thread displays the number of threads created. When the team of threads encounter the **!$OMP END PARALLEL** directive, they join and disband except for the master thread. By default the number of threads created equals the number of cores on the node. Output is shown in Table 13.

Table 12: FORTRAN 2000 program with OMP directive to execute code in parallel

```
      PROGRAM HELLO
C*******************************************************************************
C   In this simple example, the master thread forks a parallel region.  All threads in the team obtain
C   their unique thread number and print it.  The master thread only prints the total number of threads.
C   Two OpenMP library routines are used to obtain each thread's number and the number of threads.
C*******************************************************************************
      INTEGER NTHREADS, TID, OMP_GET_NUM_THREADS, OMP_GET_THREAD_NUM
C*** OMP Directive:  Fork a team of threads giving them their own copies of variables. ***
!$OMP PARALLEL PRIVATE(NTHREADS, TID)
C*** Obtain thread number and print. ***
      TID = OMP_GET_THREAD_NUM()
      PRINT *, 'Hello World from thread = ', TID
C*** Only one thread does this. ***
!$OMP SINGLE
      NTHREADS = OMP_GET_NUM_THREADS()
      PRINT *, 'Number of threads = ', NTHREADS
!$OMP END SINGLE
C*** All threads join master thread (TID=0) and disband. ***
!$OMP END PARALLEL
      END
```

Table 13: Sample output of FORTRAN 2000 program with OMP parallel execution

```
Hello World from thread = 14
Hello World from thread = 8
Hello World from thread = 10
Hello World from thread = 3
Hello World from thread = 6
Hello World from thread = 2
Hello World from thread = 11
Hello World from thread = 7
Hello World from thread = 0
Hello World from thread = 9
Hello World from thread = 15
Hello World from thread = 12
Hello World from thread = 1
Hello World from thread = 4
Hello World from thread = 13
Hello World from thread = 5
Number of threads = 16
```

4.2.2. Testing OMP on a Matrix Multiply Operation

In the next example, OMP is used to parallelize a fundamental, but important, problem: multiplying an *n* by *m* matrix **A** with an *m* by *p* matrix **B** and storing the result in an *n* by *p* matrix **C**. Implementing the solution to this example will demonstrate some key features of OMP used to parallelize loops which normally execute sequentially (one iteration at a time). The formula for computing **C** = **AB** can be expressed as follows:

$$\begin{pmatrix} C_{11} & C_{12} & \cdots & C_{1p} \\ C_{21} & C_{22} & \cdots & C_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ C_{n1} & C_{n2} & \cdots & C_{np} \end{pmatrix} = \begin{pmatrix} A_{11} & A_{12} & \cdots & A_{1m} \\ A_{21} & A_{22} & \cdots & A_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ A_{n1} & A_{n2} & \cdots & A_{nm} \end{pmatrix} \begin{pmatrix} B_{11} & B_{12} & \cdots & B_{1p} \\ B_{21} & B_{22} & \cdots & B_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ B_{m1} & B_{m2} & \cdots & B_{mp} \end{pmatrix} \qquad (28)$$

Thus the formula for computing each element of matrix **C** using a serial (one thread) implementation can be expressed as follows:

$$C_{ij} = \sum_{k=1}^{m} A_{ik} B_{kj} \qquad (29)$$

The serial (sequential) implementation of the matrix-matrix multiplication using FORTRAN 2000 is shown in Table 14 on the next page. The loops initializing matrices **A** and **B**, zeroing **C**, and computing the elements of matrix **C** are all independent. Independence is tested by running all program loops in reverse [19]. Therefore, loop iterations can be executed simultaneously – each thread can work in parallel on its own loop iteration without affecting the others. OMP will be used to parallelize these loops and reduce the execution times. In addition to using the **!$OMP PARALLEL** directive to create a team of threads, the **!$OMP DO** directive can be used to direct the team of threads to execute the loop iterations in parallel. The FORTRAN 2000 parallel implementation of the matrix-matrix multiplication using OMP is shown in Table 15 on the page following the next.

Table 14: Serial FORTRAN 2000 program used to multiply two matrices

```
  PROGRAM MMSL
!===============================================================================
!    This program sequentially multiplies matrix A by matrix B and places the results in matrix C.
!===============================================================================
  IMPLICIT NONE
!    Declare variables.  Set NRA: # rows in A.  Set NCA: # columns in A.  Set NCB: # of rows in B.
  INTEGER, PARAMETER :: NRA=32000000
  INTEGER, PARAMETER :: NCA=20
  INTEGER, PARAMETER :: NCB=40
  INTEGER :: I, J, K
  REAL, ALLOCATABLE :: A(:,:), B(:,:), C(:,:)
!    Allocate arrays
  ALLOCATE (A(NRA,NCA), B(NCA,NCB), C(NRA,NCB))
!    Initialize matrix A
  DO I=1, NRA
   DO J=1, NCA
    A(I,J) = ((I)+(J))
   END DO
  END DO
!    Initialize matrix B
  DO I=1, NCA
   DO J=1, NCB
    B(I,J) = ((I)*(J))
   END DO
  END DO
!    Zero matrix C
  DO I=1, NRA
   DO J=1, NCB
    C(I,J) = 0
   END DO
  END DO
!    Multiply matrix A by matrix B and store in matrix C
  DO I=1, NRA
   DO J=1, NCB
    DO K=1, NCA
     C(I,J) = C(I,J) + A(I,K) * B(K,J)
    END DO
   END DO
  END DO
!    Print results
  PRINT *, '********************************************************'
  PRINT *, 'Result Matrix:'
  DO I=1, NRA
   DO J=1, NCB
    WRITE (*,'(2x,f10.2,$)') C(I,J)
   END DO
   PRINT *, ' '
  END DO
  PRINT *, '********************************************************'
  PRINT *, 'Done.'
  PRINT *, ' '
  END PROGRAM MMSL
```

Table 15: Parallel implementation of matrix multiplication using OMP

```
  PROGRAM MMPL
!==========================================================================
!    This program uses OMP to multiply, in parallel, matrix A by matrix B and store in matrix C.
!==========================================================================
  IMPLICIT NONE
!    Declare variables.  Set NRA: # rows in A.  Set NCA: # columns in A.  Set NCB: # of rows in B.
  INTEGER, PARAMETER :: NRA=32000000, NCA=20, NCB=40, CHUNK=1
  INTEGER :: TID, I, J, K, OMP_GET_NUM_THREADS,OMP_GET_THREAD_NUM
  REAL (KIND=8) :: OMP_GET_WTIME, WSTIME
  REAL, ALLOCATABLE :: A(:,:), B(:,:), C(:,:)
!    Allocate arrays
  ALLOCATE (A(NRA,NCA), B(NCA,NCB), C(NRA,NCB))
!    Get wall clock start time
  WSTIME = OMP_GET_WTIME()
!    FORK: Spawn a parallel region explicitly scoping all variables
!$OMP PARALLEL SHARED(A,B,C,CHUNK) PRIVATE(TID,I,J,K)
  TID = OMP_GET_THREAD_NUM()
  IF (TID .EQ. 0) PRINT *, 'Starting matrix multiply with ', OMP_GET_NUM_THREADS(),' threads'
!    Initialize matrix A and matrix B using parallel threads on outer loop iterations
!$OMP DO SCHEDULE(STATIC, CHUNK)
  DO I=1, NRA
   DO J=1, NCA
    A(I,J) = ((I)+(J))
   END DO
  END DO
!$OMP END DO
!$OMP DO SCHEDULE(STATIC, CHUNK)
  DO I=1, NCA
   DO J=1, NCB
    B(I,J) = ((I)*(J))
   END DO
  END DO
!$OMP END DO
!    Zero matrix C using parallel threads on outer loop iterations
!$OMP DO SCHEDULE(STATIC, CHUNK)
  DO I=1, NRA
   DO J=1, NCB
    C(I,J) = 0
   END DO
  END DO
!$OMP END DO
!    Multiply matrix A by matrix B and store in matrix C using parallel threads on outer loop iterations
!$OMP DO SCHEDULE(STATIC, CHUNK)
  DO I=1, NRA
   DO J=1, NCB
    DO K=1, NCA
     C(I,J) = C(I,J) + A(I,K) * B(K,J)
    END DO
   END DO
  END DO
!$OMP END DO
!    JOIN: End of parallel region
!$OMP END PARALLEL
!    Display elapsed time
  PRINT *, 'Elapsed Time ', OMP_GET_WTIME() – WSTIME
  END PROGRAM MMPL
```

In this example, shared and private variables are explicitly scoped within the **!$OMP PARALLEL** directive with the following properties:

- SHARED – Only one copy of a shared variable exists. All threads on a team can access and modify a shared variable [18]. Care must be taken to ensure two or more threads do not simultaneously write to a shared variable, otherwise a race condition may result. Variables not specifically scoped within the parallel directive are shared by default. In the current example, the variables representing the matrices **A**, **B**, and **C** are shared by all threads. Each thread can access and update these matrices simultaneously; however, the same matrix element cannot be modified by two or more threads simultaneously, otherwise a race condition may occur.
- PRIVATE – Each thread has its own copy of a private variable [18]. For example, when the team of threads executes a parallel loop using the **!$OMP DO** directive, each thread needs its own copy of the iteration variable. In the current example, each thread has its own private copy of the thread identification number (TID) and loop iteration variables I, J, and K. Each thread works in parallel on its own loop iteration. For example, in computing the elements of matrix **C**, thread 1 may be executing iteration I=2, J=4, K=1 while thread 7 is executing I=4, J=3, K=5 simultaneously, resulting in the concurrent update of elements **C**(2,4) and **C**(4,3) which will not cause a race condition.

To test speedup and efficiency, the OMP parallel program shown in Table 15 is built and executed on the HPC DLX Cluster in serial (single threaded) and in parallel (multi-threaded) while measuring the execution time (a.k.a. wall clock time) for an increasingly larger number of rows in matrix **A**. Increasing the number of rows in matrix **A** increases the load of the matrix-matrix multiplication problem. The example in Table 15 shows thirty-two million rows (NRA = 32,000,000).

Table 16 shows combinations to be tested for the number of threads and the number of rows of matrix **A**. Five trials of serial and parallel execution are implemented.

Table 16: Threading - NRA Combinations

Number of Trials	Threading	Number of Threads	NRA (millions)
5	Serial	1	16, 32, 64, 128
5	Parallel	2	16, 32, 64, 128
5	Parallel	3	16, 32, 64, 128
5	Parallel	4	16, 32, 64, 128
5	Parallel	5	16, 32, 64, 128
5	Parallel	6	16, 32, 64, 128
5	Parallel	7	16, 32, 64, 128
5	Parallel	8	16, 32, 64, 128
5	Parallel	9	16, 32, 64, 128
5	Parallel	10	16, 32, 64, 128
5	Parallel	11	16, 32, 64, 128
5	Parallel	12	16, 32, 64, 128
5	Parallel	13	16, 32, 64, 128
5	Parallel	14	16, 32, 64, 128
5	Parallel	15	16, 32, 64, 128
5	Parallel	16	16, 32, 64, 128

The Matrix multiplication program is performed for each threading-NRA combination. For example, NRA = 32M yields the following matrix **C** terms:

$$\begin{pmatrix} C_{1,1} & C_{1,2} & \cdots & C_{1,32M} \\ C_{2,1} & C_{2,2} & \cdots & C_{2,32M} \\ \vdots & \vdots & \ddots & \vdots \\ C_{40,1} & C_{40,2} & \cdots & C_{40,32M} \end{pmatrix} = \begin{pmatrix} 3.080E03 & 3.290E03 & \cdots & 6.720E09 \\ 6.160E03 & 6.580E03 & \cdots & 1.344E10 \\ \vdots & \vdots & \ddots & \vdots \\ 1.232E05 & 1.316E05 & \cdots & 2.688E11 \end{pmatrix}$$

The average execution time is determined for each threading-NRA combination. Speedup and efficiency are then calculated using equations (26) and (27), respectively. Graphs of speedup and efficiency vs. number of threads for each NRA are fit with a polynomial to locate the maximum speedup (max point on polynomial curve). Average execution times, speedup and efficiency calculations, and graphs for each NRA are shown on the following pages.

Serial and parallel execution times, speedup, and efficiency are shown in Table 17 for the matrix multiplication program with NRA = 16M. Graphs of speedup and efficiency vs. number of threads are shown in Figure 33.

Table 17: Matrix Multiply Serial and Parallel Execution Times for NRA = 16M

Number of Trials	Threading	NRA (millions)	Number of Threads	Average Execution Time (s)	Speedup S	Efficiency E
5	Serial	16	1	21.658	1.00	100%
5	Parallel	16	2	12.104	1.79	89.5%
5	Parallel	16	3	10.452	2.07	69.1%
5	Parallel	16	4	9.414	2.30	57.5%
5	Parallel	16	5	7.880	2.75	55.0%
5	Parallel	16	6	6.836	3.17	52.8%
5	Parallel	16	7	6.140	3.53	50.4%
5	Parallel	16	8	5.780	3.75	46.8%
5	Parallel	16	9	5.044	4.29	47.7%
5	Parallel	16	10	4.858	4.46	44.6%
5	Parallel	16	11	4.846	4.47	40.6%
5	Parallel	16	12	4.850	4.47	37.2%
5	Parallel	16	13	4.920	4.40	33.9%
5	Parallel	16	14	5.964	3.63	25.9%
5	Parallel	16	15	6.588	3.29	21.9%
5	Parallel	16	16	7.670	2.82	17.6%

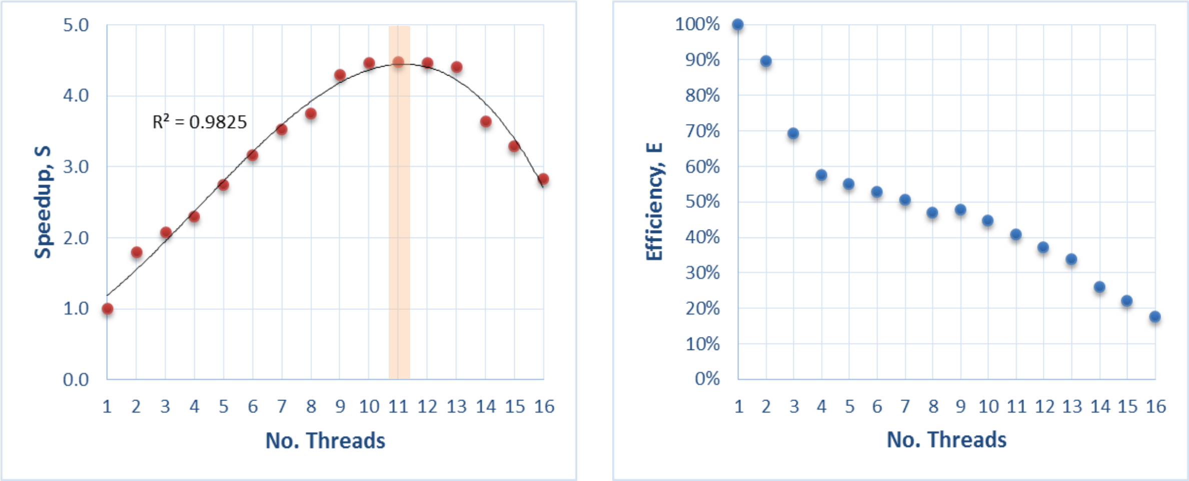


Figure 33: Speedup and efficiency vs number of threads for NRA = 16M

Serial and parallel execution times, speedup, and efficiency are shown in Table 18 for the matrix multiplication program with NRA = 32M. Graphs of speedup and efficiency vs. number of threads are shown in Figure 34.

Table 18: Matrix Multiply Serial and Parallel Execution Times for NRA = 32M

Number of Trials	Threading	NRA (millions)	Number of Threads	Average Execution Time (s)	Speedup S	Efficiency E
5	Serial	32	1	49.068	1.00	100%
5	Parallel	32	2	29.908	1.64	82.0%
5	Parallel	32	3	23.814	2.06	68.7%
5	Parallel	32	4	19.418	2.53	63.2%
5	Parallel	32	5	16.196	3.03	60.6%
5	Parallel	32	6	13.978	3.51	58.5%
5	Parallel	32	7	12.476	3.93	56.2%
5	Parallel	32	8	12.016	4.08	51.0%
5	Parallel	32	9	11.094	4.42	49.1%
5	Parallel	32	10	10.706	4.58	45.8%
5	Parallel	32	11	10.442	4.70	42.7%
5	Parallel	32	12	10.924	4.49	37.4%
5	Parallel	32	13	10.752	4.56	35.1%
5	Parallel	32	14	11.906	4.12	29.4%
5	Parallel	32	15	12.850	3.82	25.5%
5	Parallel	32	16	14.988	3.27	20.5%

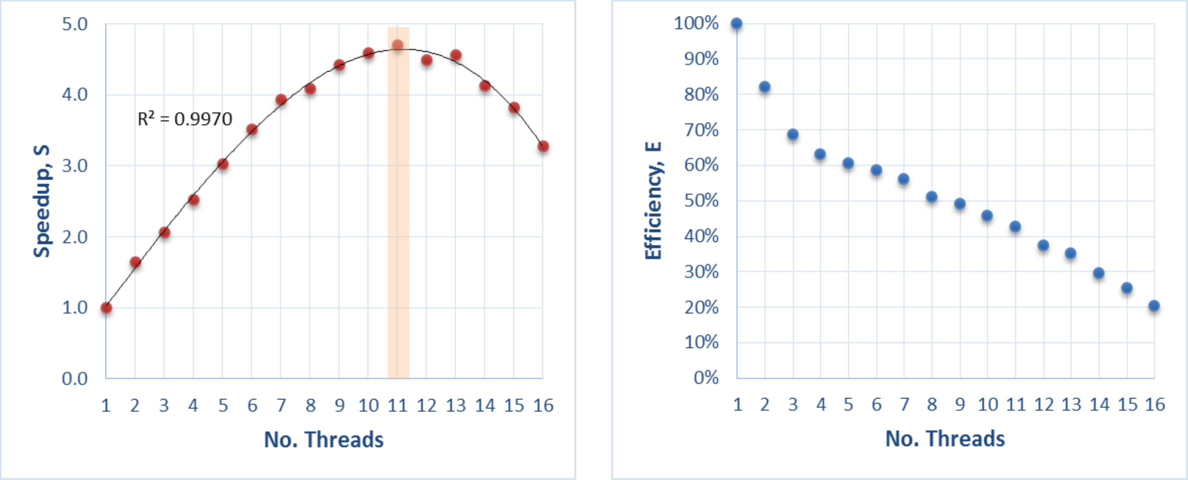


Figure 34: Speedup and efficiency vs number of threads for NRA = 32M

Serial and parallel execution times, speedup, and efficiency are shown in Table 19 for the matrix multiplication program with NRA = 64M. Graphs of speedup and efficiency vs. number of threads are shown in Figure 35.

Table 19: Matrix Multiply Serial and Parallel Execution Times for NRA = 64M

Number of Trials	Threading	NRA (millions)	Number of Threads	Average Execution Time (s)	Speedup S	Efficiency E
5	Serial	64	1	113.598	1.00	100%
5	Parallel	64	2	69.004	1.65	82.3%
5	Parallel	64	3	49.000	2.32	77.3%
5	Parallel	64	4	40.564	2.80	70.0%
5	Parallel	64	5	35.560	3.19	63.9%
5	Parallel	64	6	30.694	3.70	61.7%
5	Parallel	64	7	27.666	4.11	58.7%
5	Parallel	64	8	26.646	4.26	53.3%
5	Parallel	64	9	24.466	4.64	51.6%
5	Parallel	64	10	24.128	4.71	47.1%
5	Parallel	64	11	21.902	5.19	47.2%
5	Parallel	64	12	22.024	5.16	43.0%
5	Parallel	64	13	21.384	5.31	40.9%
5	Parallel	64	14	20.918	5.43	38.8%
5	Parallel	64	15	21.136	5.37	35.8%
5	Parallel	64	16	21.476	5.29	33.1%

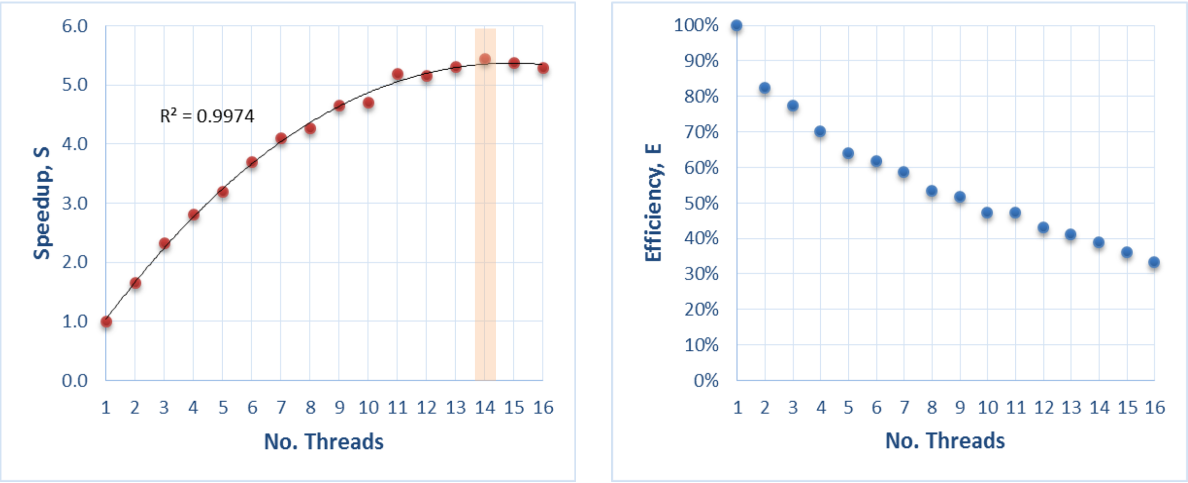


Figure 35: Speedup and efficiency vs number of threads for NRA = 64M

Serial and parallel execution times, speedup, and efficiency are shown in Table 20 for the matrix multiplication program with NRA = 128M. Graphs of speedup and efficiency vs. number of threads are shown in Figure 36.

Table 20: Matrix Multiply Serial and Parallel Execution Times for NRA = 128M

Number of Trials	Threading	NRA (millions)	Number of Threads	Average Execution Time (s)	Speedup S	Efficiency E
5	Serial	128	1	243.822	1.00	100%
5	Parallel	128	2	138.812	1.76	87.8%
5	Parallel	128	3	106.870	2.28	76.0%
5	Parallel	128	4	85.240	2.86	71.5%
5	Parallel	128	5	73.022	3.34	66.8%
5	Parallel	128	6	63.840	3.82	63.7%
5	Parallel	128	7	57.732	4.22	60.3%
5	Parallel	128	8	57.538	4.24	53.0%
5	Parallel	128	9	50.332	4.84	53.8%
5	Parallel	128	10	48.812	5.00	50.0%
5	Parallel	128	11	47.924	5.09	46.3%
5	Parallel	128	12	46.412	5.25	43.8%
5	Parallel	128	13	44.006	5.54	42.6%
5	Parallel	128	14	44.344	5.50	39.3%
5	Parallel	128	15	45.426	5.37	35.8%
5	Parallel	128	16	47.496	5.13	32.1%

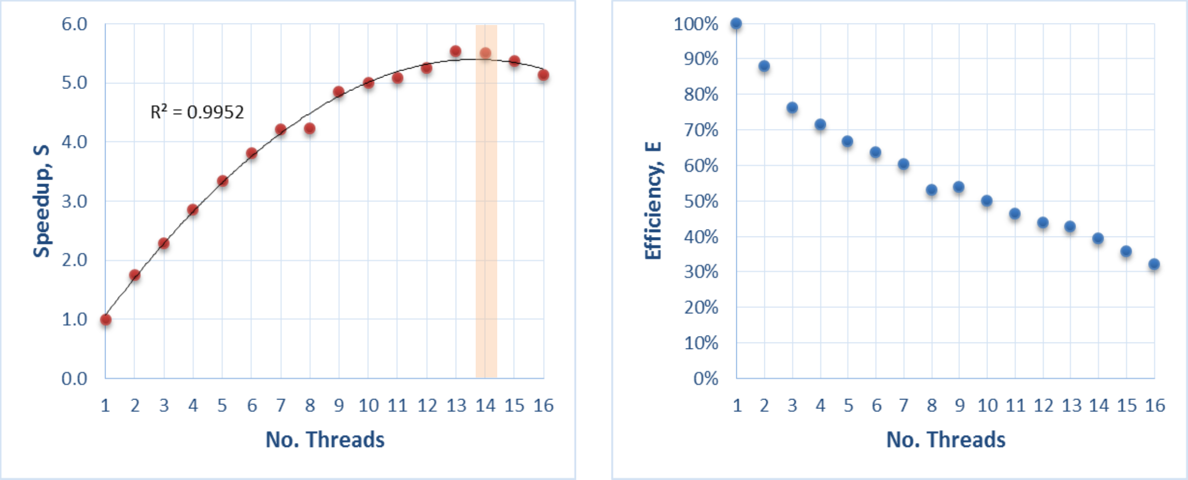


Figure 36: Speedup and efficiency vs number of threads for NRA = 128M

Table 21 shows the maximum speedup observed with the associated number of threads and efficiency for each NRA.

Table 21: Maximum Speedup for each NRA Value

Number of Trials	Threading	NRA (millions)	Number of Threads	Maximum Speedup S_{max}	Efficiency E
5	Parallel	16	11	4.47	40.6%
5	Parallel	32	11	4.70	42.7%
5	Parallel	64	14	5.43	38.8%
5	Parallel	128	14	5.50	39.3%

Increasing the number of threads increases the performance of each loop and thereby reduces the execution time. However, increasing the number of threads also increases the overhead cost which reduces loop performance and lengthens execution time. A maximum is observed in the speedup vs. the number of threads. In addition, increasing the number of threads decreased the efficiency as each new thread requires more resources than can be compensated for by speedup. Speedup and efficiency have an inverse relationship. A balance between speedup and efficiency will be sought.

4.3. Integrating OMP into GEMINI Solver's Matrix Fill Routine

Open MP threading is well suited for parallelizing the loop structure found within the impedance matrix fill routine of GEMINI Solver. Table 22 on the next page shows the GEMINI Solver program tree with the optimal location to incorporate OMP threading. This location is optimal because (1) MPI is used only outside of the OMP parallel region [17] and (2) each OMP thread is dedicated to an observation element's interaction with its source elements when calculating Z_{mn} [eq. (18)]. OMP threading can be implemented on the outer loop (loop over observation elements) of the matrix fill routine in two ways:

- Simple OMP parallelism
- Nested OMP parallelism.

In the first case, a team of threads performs the iterations of the outer loop over observation elements. In the second case, a team of parent threads performs the iterations of the outer loop over observations elements and each parent has its own team of daughter threads perform the iterations of the inner loop over source elements.

Table 22: GEMINI Solver program tree with optimal OMP threading location

<u>**GEMINI SOLVER Program Tree**</u>

```
Begin GEMINI SOLVER
    CALL & RETURN ProjectModule%openPrimaryFiles
    CALL & RETURN ProjectModule%readFromFile
    CALL & RETURN ProjectModule%writeSimulationData
    CALL SolutionModule%solution
        CALL SolutionModule%nonperiodic/periodic
        ! loop over frequencies
            ! Begin “Fill the impedance matrix”
            CALL GlobalMatrixModule%fillNormal
            ! loop over observation elements --> loop over node sets for this observation element
                ! Begin “Moment method analysis”
                ! loop over source elements --> loop over node sets for this source element
                    CALL LocalMatrixModule%fillMoM
                    ! loop over observation points --> loop over the number of common regions
                        CALL OperatorsModule%FillZMatrix
                        ! loop over quadrature points --> loop over source nodes --> loop over observation nodes
                            ! calculation: obsBasis%lambda(iObs) .dot. srcBasis(iQuad)%lambda(jSrc))*G(iQuad)*srcWghtJacobian(iQuad)*obsWghtJacobian
                            ! calculation: obsBasis%divLambda(iObs)*srcBasis(iQuad)%divLambda(jSrc)*srcWghtJacobian(iQuad)*obsWghtJacobian
                        ! end loop over observation nodes --> end loop over source nodes --> end loop over quadrature points
                        RETURN OperatorsModule%FillZMatrix
                        CALL OperatorsModule%FillBetaMatrix
                        ! loop over quadrature points --> loop over source nodes --> loop over observation nodes
                            ! calculation: obsBasisCrossNormal(iObs) .dot. (gradG(iQuad) .cross. srcBasis(iQuad)%lambda(jSrc)))*srcWghtJacobian(iQuad)*obsWghtJacobian
                        ! end loop over observation nodes --> end loop over source nodes --> end loop over quadrature points
                        RETURN OperatorsModule%FillBetaMatrix
                        CALL OperatorsModule%FillYMatrix
                        ! loop over quadrature points --> loop over source nodes --> loop over observation nodes
                            ! calculation: obsBasisCrossNormal(iObs) .dot. srcBasis(iQuad)%lambda(jSrc))*G(iQuad)*srcWghtJacobian(iQuad)*obsWghtJacobian
                            ! calculation: obsBasisCrossNormal(iObs) .dot. gradG(iQuad))*srcBasis(iQuad)%divLambda(jSrc)*srcWghtJacobian(iQuad)*obsWghtJacobian
                        ! end loop over observation nodes --> end loop over source nodes --> end loop over quadrature points
                        RETURN OperatorsModule%FillYMatrix
                        CALL OperatorsModule%FillBetaTildeMatrix
                        ! loop over quadrature points --> loop over source nodes --> loop over observation nodes
                            ! calculation: obsBasis%lambda(iObs) .dot. (gradG(iQuad) .cross. srcBasis(iQuad)%lambda(jSrc)))*srcWghtJacobian(iQuad)*obsWghtJacobian
                        ! end loop over observation nodes --> end loop over source nodes --> end loop over quadrature points
                        RETURN OperatorsModule%FillBetaTildeMatrix
                    ! end loop over the number of common regions --> end loop over observation points
                    RETURN LocalMatrixModule%fillMoM
                    CALL & RETURN GlobalMatrixModule%localToGlobal
                ! end loop over node sets for this source element --> end loop over source elements
                ! End “Moment method analysis”
            ! end loop over node sets for this observation element --> end loop over observation elements
            RETURN GlobalMatrixModule%fillNormal
            ! End “Fill the impedance matrix”
            ! Begin “Fill the right hand side for each excitation group”
            ! loop over excitation groups
                CALL & RETURN GlobalVector%fill
                ! loop over global wave sources --> loop over elements --> loop over node sets for this element
                    LocalVectorModule%fillLocalSource
                    LocalVectorModule%localToGlobal
                ! end loop over node sets for this element --> end loop over elements --> loop over global wave sources
                ! loop over voltage sources-> loop over node sets
                    LocalVectorModule%fillLocalSource
                    LocalVectorModule%localToGlobal
                ! end loop over node sets --> end loop over voltage sources
                CALL & RETURN ISISComplexSolverModule%solve
                CALL & RETURN ISISComplexVectorModule%gather
                CALL & RETURN SolutionModule%writeSolution
            ! end loop over excitation groups
            ! End “Fill the right hand side for each excitation group”
        ! end loop over frequencies
        RETURN SolutionModule%nonperiodic/periodic
    RETURN SolutionModule%solution
    CALL & RETURN ProjectModule%closeFiles
End GEMINI SOLVER
```

OMP

Threading

Here

4.3.1. Incorporating Simple OMP Parallelism into GEMINI Solver

In the case of simple OMP parallelism, a team of threads is created to perform the iterations of the outer loop over observation elements. Each thread uses a selected observation element and loops over each source element to compute a local observation-source element interaction matrix, which in turn is used by the thread to update the global matrix. Figure 31 shows simple outer loop parallelization of the matrix fill routine.

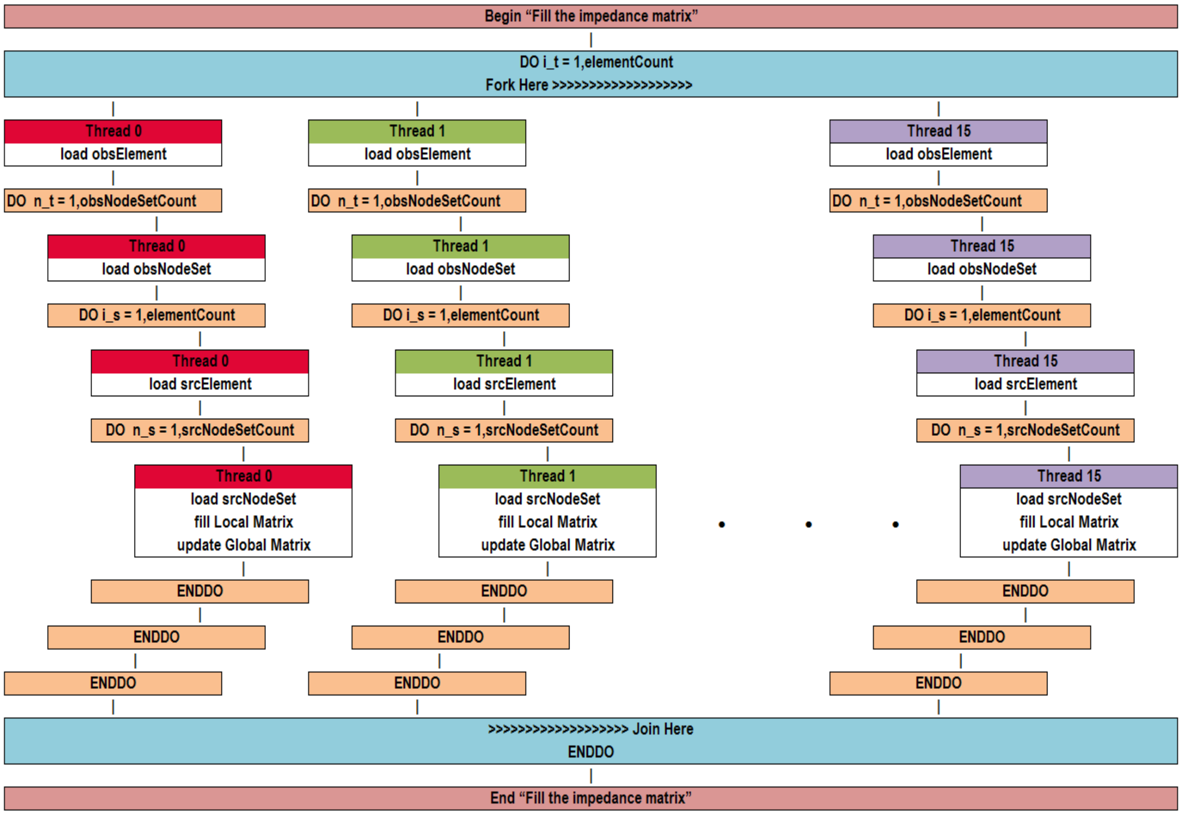


Figure 37: Simple parallelization of GEMINI Solver matrix fill routine

The integration of simple OMP parallelism into the FORTRAN 2000 matrix fill routine is displayed in Table 23 and Table 24 on the following pages.

Several race conditions occur arising from OMP threading which cause moderate errors in the output edge currents. These race conditions will be solved in the final MPI-OMP hybrid parallelization of GEMINI Solver discussed in section 5.1

Table 23: Simple OMP parallelization of FORTRAN 2000 matrix fill routine, part 1

```
  SUBROUTINE fillNormal(project,freqIndex,gMatrix)
!
  < Declarations Omitted >
  elementCount = project%elements%listSize()
  freqPointer => project%frequencies%at(freqIndex)
  modelndex = freqPointer%modeIndex
  omega = 2.0_dk*Constants%pi*freqPointer%frequency
!
! Refresh obsBases for each observation element and scrBases for each source element
  DO ip = 1,elementCount
    oE => project%elements%at(ip)
    obsBasis => project%obsBases%at(oE%type() + 1)
    sE => project%elements%at(ip)
    srcBasis => project%srcBases%at(sE%type() + 1)
  ENDDO
!
! Create a team of threads to execute code in parallel
!$OMP PARALLEL                                                                                                  &
!$OMP SHARED(project,gMatrix,freqIndex, elementCount,freqPointer,modeIndex,omega)                               &
!$OMP PRIVATE(i_t,n_t,obsBasis,obsElement,obsNodeSetCount,obsNodeSet,obsNodeCount,obsArray)                     &
!$OMP PRIVATE(i_s,n_s,srcBasis,srcElement,srcNodeSetCount,srcNodeSet,srcNodeCount,commonRegions,regionCount)    &
!$OMP PRIVATE(obsSourceFlag,obsJsourceJ,obsJsourceM,obsMsourceJ,obsMsourceM,obsIndex,srcIndex)                  &
!$OMP NUM_THREADS(NTHREADS)
!>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>> Threads Fork Here
!
! Direct team of threads to execute iterations of outer observation element loop in parallel using dynamic scheduling
!$OMP DO SCHEDULE(STATIC, CHUNK)
!
! Loop over observation elements
  DO i_t = 1,elementCount
    obsElement => project%elements%at(i_t)
    IF (.NOT. obsElement%contributesToRow .OR. obsElement%isGhost()) CYCLE
    obsBasis => project%obsBases%at(obsElement%type() + 1)
    obsNodeSetCount = SIZE(obsElement%nodeSets(:))
!
! Loop over node sets for this element
    DO n_t = 1,obsNodeSetCount
      obsNodeSet => obsElement%nodeSets(n_t)%object
      obsNodeCount = obsElement%nodeCount(n_t)
      CALL ObservationArrayStatic%create(obsElement,n_t,obsBasis,obsArray)
!
! Loop over source elements
        DO i_s = 1,elementCount
          srcElement => project%elements%at(i_s)
          IF (.NOT. srcElement%contributesToColumn) CYCLE
          srcBasis => project%srcBases%at(srcElement%type() + 1)
          srcNodeSetCount = SIZE(srcElement%nodeSets(:))
!
! Loop over node sets for this element
          DO n_s = 1,srcNodeSetCount
            srcNodeSet => srcElement%nodeSets(n_s)%object
            IF (srcNodeSet%equation == NodeSetEquations%E_FEM) CYCLE
            srcNodeCount = srcElement%nodeCount(n_s)
            CALL CommonRegionStatic%create(obsNodeSet,srcNodeSet,commonRegions)
            regionCount = CommonRegionStatic%regionCount
            IF (regionCount == 0) CYCLE
            IF (obsNodeSet%equation == NodeSetEquations%HYBRID_SOURCES .OR.                                     &
                srcNodeSet%equation == NodeSetEquations%HYBRID_SOURCES) THEN
              CommonRegionStatic%regionCount = 1
              regionCount = 1
            ENDIF
```

Table 24: Simple OMP parallelization of FORTRAN 2000 matrix fill routine, part 2

```
!
! Create element-element interaction matrix
      CALL createLocals(regionCount,obsNodeCount,srcNodeCount,obsJsourceJ,obsJsourceM,obsMsourceJ,obsMsourceM)!
! Find element to element interaction
      CALL LocalMatrix%fill(project,omega,obsElement,n_t,obsArray,srcElement,n_s,commonRegions,                   &
                   obsSourceFlag,obsJsourceJ,obsJsourceM,obsMsourceJ,obsMsourceM)
!
! Thin material contribution
      IF (srcNodeSet%equation == NodeSetEquations%THIN_PEC_EFIE .OR.                                            &
          srcNodeSet%equation == NodeSetEquations%THIN_PEC_MFIE .OR.                                            &
          srcNodeSet%equation == NodeSetEquations%THIN_PEC_CFIE) THEN
!
       CALL LocalMatrix%fill(project,omega,obsElement,n_t, obsArray,srcElement,n_s,                             &
                    commonRegions,obsSourceFlag,obsJsourceJ)
      ENDIF
!
! Place element interactions into global matrix
      IF (obsSourceFlag(1)) THEN
       obsIndex = 1
       srcIndex = 1
       CALL localToGlobal(modelIndex,obsElement,n_t,obsNodeCount,obsElement%unknownJ(n_t),srcElement,n_s,        &
                  srcNodeCount,srcElement%unknownJ(n_s),commonRegions,obsJsourceJ,gMatrix)
      ENDIF
      IF (obsSourceFlag(2)) THEN
       obsIndex = 1
       srcIndex = 2
       CALL localToGlobal(modelIndex,obsElement,n_t,obsNodeCount,obsElement%unknownJ(n_t),srcElement,n_s,        &
                  srcNodeCount,srcElement%unknownM(n_s),commonRegions,obsJsourceM,gMatrix)
      ENDIF
      IF (obsSourceFlag(3)) THEN
       obsIndex = 2
       srcIndex = 1
       CALL localToGlobal(modelIndex,obsElement,n_t,obsNodeCount,obsElement%unknownM(n_t),srcElement,n_s,        &
                  srcNodeCount,srcElement%unknownJ(n_s), commonRegions,obsMsourceJ,gMatrix)
      ENDIF
      IF (obsSourceFlag(4)) THEN
       obsIndex = 2
       srcIndex = 2
       CALL localToGlobal(modelIndex,obsElement,n_t,obsNodeCount,obsElement%unknownM(n_t),srcElement,n_s,        &
                  srcNodeCount,srcElement%unknownM(n_s), commonRegions,obsMsourceM,gMatrix)
      ENDIF
     ENDDO
    ENDDO
   ENDIF
   CALL ObservationArrayStatic%destroy(obsArray)
  ENDDO
 ENDDO
!
!$OMP END DO
!
!>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>> Threads Join Here
!$OMP END PARALLEL
!
 CALL deleteLocals(obsJsourceJ,obsJsourceM,obsMsourceJ,obsMsourceM)
!
! Add lumped loads
 CALL addLumpedLoads(project,omega,gMatrix)
!
 END SUBROUTINE fillNormal
```

Simple OMP parallelization of the GEMINI Solver v2.0 FORTRAN 2000 matrix fill routine is run on the test cases in Table 25 to solve for the output currents.

Table 25: Test cases run using simple OMP parallelization of GEMINI Solver v2.0

Frequency, f (GHz)	Edges, N (# unknowns)	Solution Test	OMP Parallelism	# MPI Processes	# OMP Threads	Performance Comparison
0.1499	1083	EFIE	Simple	1	1,2,4,8,16	Yes
0.2998	4455	EFIE	Simple	8,16,32	1,2,4,8,16	Yes
0.8994	41,415	EFIE	Simple	8,16,32	1,2,4,8,16	Yes

The matrix fill time, speedup, efficiency, and speedup-efficiency product (SEP) for each OMP threading case with one MPI process on N = 1083 unknowns are shown in Table 26. The highest, most efficient speedup (SEP = 2.2) occurs for 8 threads. Graphs of speedup and efficiency vs. threads are shown in Figure 38 and Figure 39, respectively.

Table 26: GEMINI Solver v2.0 Serial and Parallel Execution Times for N = 1083

OMP Threading	N (unknowns)	# MPI Processes	# OMP Threads	Execution Time (s)	Speedup S	Efficiency E	SEP
Serial	1083	1	1	7.91180	1.00	100.0%	1.0
Parallel	1083	1	2	5.43418	1.46	72.8%	1.1
Parallel	1083	1	4	3.18102	2.49	62.2%	1.5
Parallel	1083	1	8	1.88971	4.19	52.3%	2.2
Parallel	1083	1	16	2.06925	3.82	23.9%	0.9

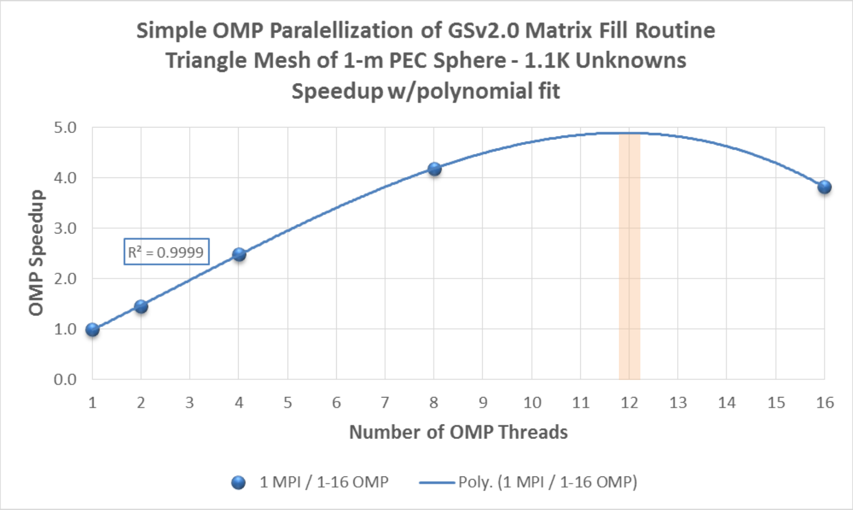


Figure 38: Speedup vs number of threads for N = 1083

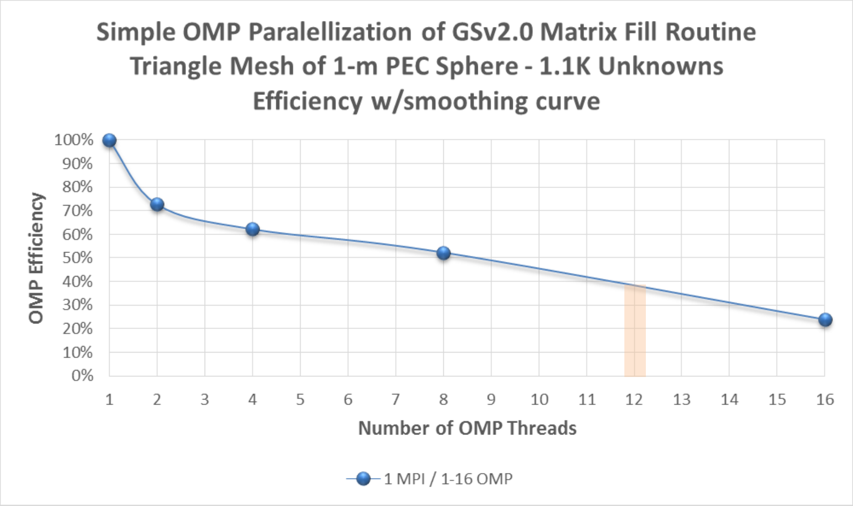


Figure 39: Efficiency vs number of threads for N = 1083

The highest speedup, projected from Figure 38, would occur for 12 threads. However, its SEP ≈ 1.9 is less than that for 8 threads.

The matrix fill time, speedup, efficiency, and SEP for each OMP threading case with 8, 16, and 32 MPI process on N = 4455 unknowns are shown in Table 27. The highest, most efficient speedup (largest SEP) occurs for 8 threads. Graphs of speedup and efficiency vs. number of threads are shown in Figure 40 and Figure 41, respectively.

Table 27: GEMINI Solver v2.0 Serial and Parallel Execution Times for N = 4455

OMP Threading	N (unknowns)	# MPI Processes	# OMP Threads	Execution Time (s)	Speedup S	Efficiency E	SEP
Serial	4455	8	1	35.80556	1.00	100.0%	1.0
Parallel	4455	8	2	24.24482	1.48	73.8%	1.1
Parallel	4455	8	4	13.61293	2.63	65.8%	1.7
Parallel	4455	8	8	8.74592	4.09	51.2%	2.1
Parallel	4455	8	16	10.20651	3.51	21.9%	0.8
Serial	4455	16	1	28.71364	1.00	100.0%	1.0
Parallel	4455	16	2	18.81764	1.53	76.3%	1.2
Parallel	4455	16	4	10.20070	2.81	70.4%	2.0
Parallel	4455	16	8	6.46964	4.44	55.5%	2.5
Parallel	4455	16	16	7.61884	3.77	23.6%	0.9
Serial	4455	32	1	15.70761	1.00	100.0%	1.0
Parallel	4455	32	2	11.32628	1.39	69.3%	1.0
Parallel	4455	32	4	6.60075	2.38	59.5%	1.4
Parallel	4455	32	8	4.17962	3.76	47.0%	1.8
Parallel	4455	32	16	4.86951	3.23	20.2%	0.7

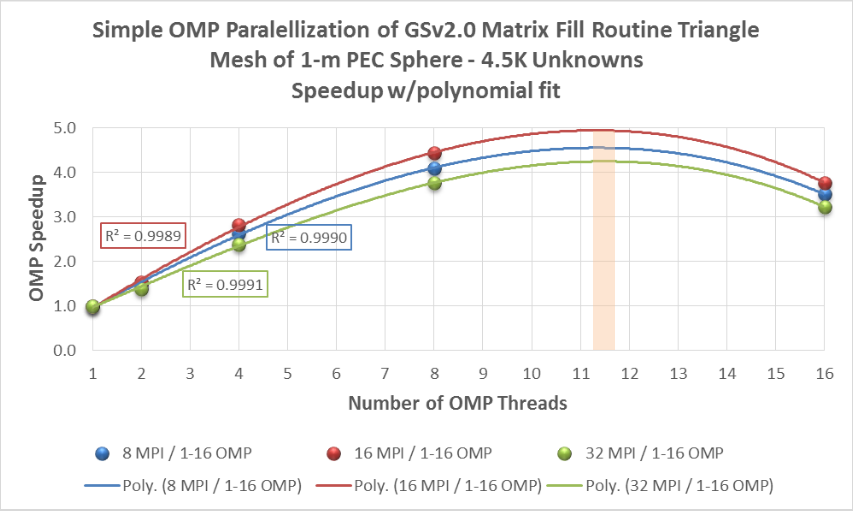


Figure 40: Speedup vs number of threads for N = 4455

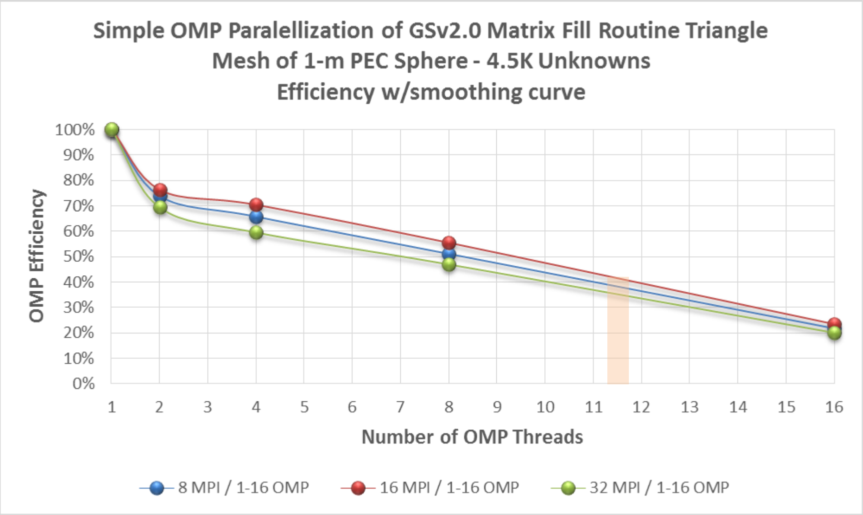


Figure 41: Efficiency vs number of threads for N = 4455

The highest speedup, projected from Figure 40, would occur for 11½ threads. However, its SEP ≈ 1.8, SEP ≈ 2.0, and SEP ≈ 1.5 for 8, 16, and 32 MPI processes, respectively, are less than those corresponding to 8 threads.

The matrix fill time, speedup, efficiency, and SEP for each OMP threading case with 8, 16, and 32 MPI process on N = 41,415 unknowns are shown in Table 28. The highest, most efficient speedup occurs for 4 or 8 threads. Graphs of speedup and efficiency vs. number of threads are shown in Figure 42 and Figure 43, respectively.

Table 28: GEMINI Solver v2.0 Serial and Parallel Execution Times for N = 41,415

OMP Threading	N (unknowns)	# MPI Processes	# OMP Threads	Execution Time (min)	Speedup S	Efficiency E	SEP
Serial	41,415	8	1	42.654	1.00	100.0%	1.0
Parallel	41,415	8	2	31.058	1.37	68.7%	0.9
Parallel	41,415	8	4	17.452	2.44	61.1%	1.5
Parallel	41,415	8	8	12.747	3.35	41.8%	1.4
Parallel	41,415	8	16	14.063	3.03	19.0%	0.6
Serial	41,415	16	1	30.583	1.00	100.0%	1.0
Parallel	41,415	16	2	23.712	1.29	64.5%	0.8
Parallel	41,415	16	4	13.161	2.32	58.1%	1.4
Parallel	41,415	16	8	8.966	3.41	42.6%	1.5
Parallel	41,415	16	16	10.771	2.84	17.7%	0.5
Serial	41,415	32	1	18.157	1.00	100.0%	1.0
Parallel	41,415	32	2	14.305	1.27	63.5%	0.8
Parallel	41,415	32	4	7.661	2.37	59.3%	1.4
Parallel	41,415	32	8	5.815	3.12	39.0%	1.2
Parallel	41,415	32	16	6.145	2.95	18.5%	0.5

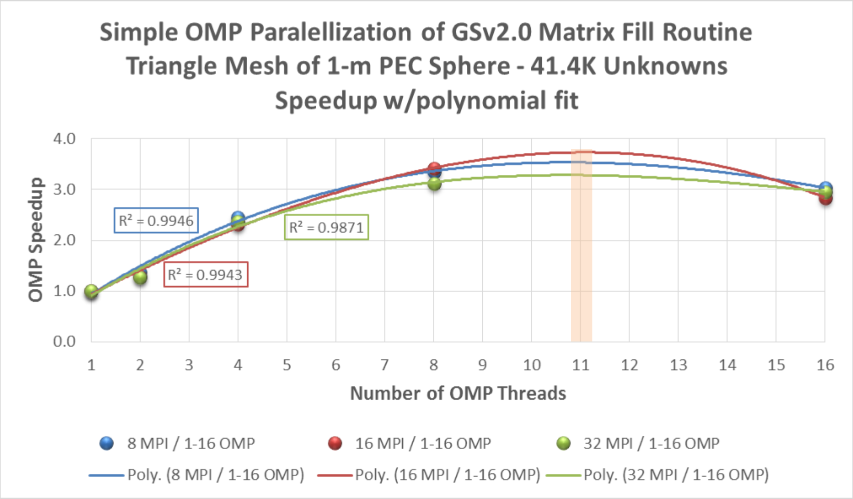


Figure 42: Speedup vs number of threads for N = 41,415

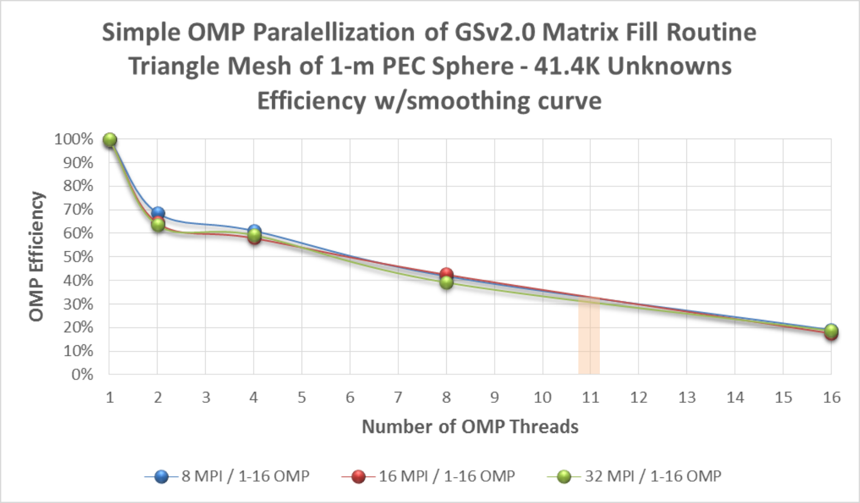


Figure 43: Efficiency vs number of threads for N = 41,415

The highest speedup, projected from Figure 42, would occur for 11 threads. However, its SEP ≈ 1.1, SEP ≈ 1.2, and SEP ≈ 1.0 for 8, 16, and 32 MPI processes, respectively, are less than those corresponding 4 or 8 threads.

The speedup-efficiency product (SEP) bears some discussion. Consider parallel cases with 16 MPI process on N = 4455 unknowns with the results given in Table 29.

Table 29: GSv2.0 Parallel Execution S, E, and SEP for N = 4455

OMP Threading	N (unknowns)	# MPI Processes	# OMP Threads	Speedup S	Efficiency E	SEP
Parallel	4455	16	1	1.00	100.0%	1.0
Parallel	4455	16	4	2.81	70.4%	2.0
Parallel	4455	16	8	4.44	55.5%	2.5
Parallel	4455	16	11½	5*	41%**	2.1

*Estimated from fit to graph in Figure 40. ** Estimated from fit to graph in Figure 41.

The SEP takes into account both speedup and efficiency. A single thread always yields the ideal efficiency 100%. If 4 threads were 100% efficient, the speedup would be 4; however, the actual speedup is only 2.81, or 70.4% efficient. Parallel execution with 4 threads has 29.6% inefficiency and it gets worse as the number of threads increases. Although 11½ threads yield 5x speedup, the efficiency is only 41%. To compare parallel execution cases, the *effective* speedup is measured using the speedup-efficiency product.

4.3.2. Incorporating Nested OMP Parallelism into GEMINI Solver

In the case of nested OMP parallelism, a team of parent threads is created to perform the iterations of the outer loop over observations elements. For each parent thread, a team of daughter threads is created to perform the iterations of the inner loop over sources elements. Each daughter thread uses the selected observation element of its parent thread and its own source element to compute a local observation-source element interaction matrix, which in turn is used by the daughter thread to update the global matrix. Figure 44 shows outer-inner loop nested parallelization of the matrix fill routine.

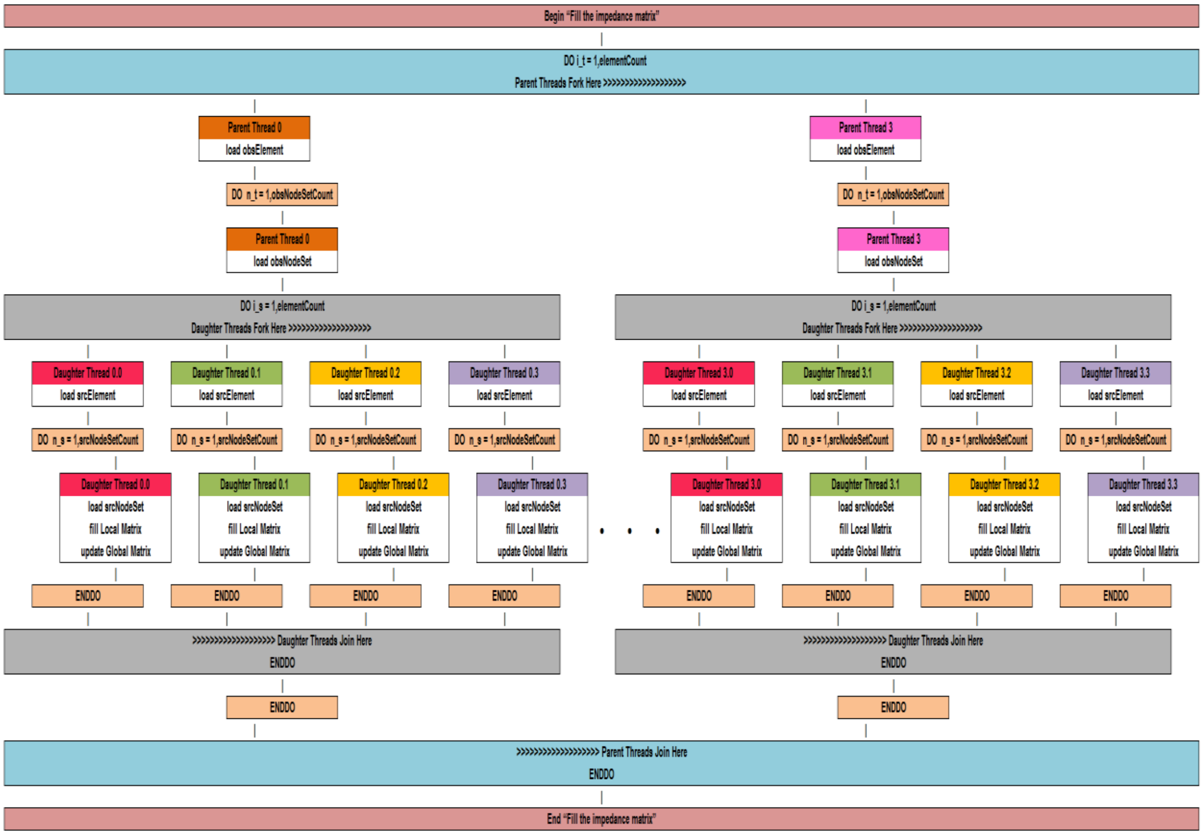


Figure 44: Nested parallelization of GEMINI Solver

The integration of nested OMP parallelism into the FORTRAN 2000 matrix fill routine is displayed in Table 30 and Table 31 on the following pages. The routine contains two nested parallel regions with enclosed parallel loops.

Table 30: Nested OMP parallelization of FORTRAN 2000 matrix fill routine, part 1

```
 SUBROUTINE fillNormal(project,freqIndex,gMatrix)
!
 < Declarations Omitted >
 elementCount = project%elements%listSize()
 freqPointer => project%frequencies%at(freqIndex)
 modelIndex = freqPointer%modelIndex
 omega = 2.0_dk*Constants%pi*freqPointer%frequency
!
! Refresh obsBases for each observation element and scrBases for each source element
 DO ip = 1,elementCount
   oE => project%elements%at(ip)
   obsBasis => project%obsBases%at(oE%type() + 1)
   sE => project%elements%at(ip)
   srcBasis => project%srcBases%at(sE%type() + 1)
 ENDDO
!
! Create a team of parent threads to execute code in parallel
!$OMP PARALLEL                                                                                              &
!$OMP SHARED(project,gMatrix,freqIndex, elementCount,freqPointer,modelIndex,omega)                          &
!$OMP PRIVATE(i_t,n_t,obsBasis,obsElement,obsNodeSetCount,obsNodeSet,obsNodeCount,obsArray)                 &
!$OMP NUM_THREADS(obsNTHREADS)
!>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>> Parent Threads Fork Here
! Direct team of parent threads to execute iterations of outer observation element loop in parallel using dynamic scheduling
!$OMP DO SCHEDULE(DYNAMIC, CHUNK)
!
! Loop over observation elements
 DO i_t = 1,elementCount
   obsElement => project%elements%at(i_t)
   IF (.NOT. obsElement%contributesToRow .OR. obsElement%isGhost()) CYCLE
   obsBasis => project%obsBases%at(obsElement%type() + 1)
   obsNodeSetCount = SIZE(obsElement%nodeSets(:))
!
! Loop over node sets for this element
   DO n_t = 1,obsNodeSetCount
     obsNodeSet => obsElement%nodeSets(n_t)%object
     obsNodeCount = obsElement%nodeCount(n_t)
     CALL ObservationArrayStatic%create(obsElement,n_t,obsBasis,obsArray)
!
! Create a team of daughter threads to execute code in parallel
!$OMP PARALLEL                                                                                              &
!$OMP SHARED(project,gMatrix,freqIndex, elementCount,freqPointer,modelIndex,omega)                          &
!$OMP SHARED (i_t,n_t,obsBasis,obsElement,obsNodeSetCount,obsNodeSet,obsNodeCount,obsArray)                 &
!$OMP PRIVATE(i_s,n_s,srcBasis,srcElement,srcNodeSetCount,srcNodeSet,srcNodeCount,commonRegions,regionCount) &
!$OMP PRIVATE(obsSourceFlag,obsJsourceJ,obsJsourceM,obsMsourceJ,obsMsourceM,obsIndex,srcIndex)              &
!$OMP NUM_THREADS(srcNTHREADS)
!>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>> Daughter Threads Fork Here
! Direct team of daughter threads to execute iterations of inner source element loop in parallel using dynamic scheduling
!$OMP DO SCHEDULE(DYNAMIC, CHUNK)
!
! Loop over source elements
       DO i_s = 1,elementCount
         srcElement => project%elements%at(i_s)
         IF (.NOT. srcElement%contributesToColumn) CYCLE
         srcBasis => project%srcBases%at(srcElement%type() + 1)
         srcNodeSetCount = SIZE(srcElement%nodeSets(:))
!
! Loop over node sets for this element
         DO n_s = 1,srcNodeSetCount
           srcNodeSet => srcElement%nodeSets(n_s)%object
           IF (srcNodeSet%equation == NodeSetEquations%E_FEM) CYCLE
           srcNodeCount = srcElement%nodeCount(n_s)
           CALL CommonRegionStatic%create(obsNodeSet,srcNodeSet,commonRegions)
           regionCount = CommonRegionStatic%regionCount
           IF (regionCount == 0) CYCLE
           IF (obsNodeSet%equation == NodeSetEquations%HYBRID_SOURCES .OR.                                  &
               srcNodeSet%equation == NodeSetEquations%HYBRID_SOURCES) THEN
             CommonRegionStatic%regionCount = 1
             regionCount = 1
           ENDIF
```

Table 31: Nested OMP parallelization of FORTRAN 2000 matrix fill routine, part 2

```
!
! Create element-element interaction matrix
      CALL createLocals(regionCount,obsNodeCount,srcNodeCount,obsJsourceJ,obsJsourceM,obsMsourceJ,obsMsourceM)
!
! Find element to element interaction
      CALL LocalMatrix%fill(project,omega,obsElement,n_t,obsArray,srcElement,n_s,commonRegions,             &
                   obsSourceFlag,obsJsourceJ,obsJsourceM,obsMsourceJ,obsMsourceM)
!
! Thin material contribution
      IF (srcNodeSet%equation == NodeSetEquations%THIN_PEC_EFIE .OR.                                         &
        srcNodeSet%equation == NodeSetEquations%THIN_PEC_MFIE .OR.                                           &
        srcNodeSet%equation == NodeSetEquations%THIN_PEC_CFIE) THEN
!
       CALL LocalMatrix%fill(project,omega,obsElement,n_t, obsArray,srcElement,n_s,                          &
                    commonRegions,obsSourceFlag,obsJsourceJ)
      ENDIF
!
! Place element interactions into global matrix
      IF (obsSourceFlag(1)) THEN
       obsIndex = 1
       srcIndex = 1
       CALL localToGlobal(modeIndex,obsElement,n_t,obsNodeCount,obsElement%unknownJ(n_t),srcElement,n_s,     &
                   srcNodeCount,srcElement%unknownJ(n_s),commonRegions,obsJsourceJ,gMatrix)
      ENDIF
      IF (obsSourceFlag(2)) THEN
       obsIndex = 1
       srcIndex = 2
       CALL localToGlobal(modeIndex,obsElement,n_t,obsNodeCount,obsElement%unknownJ(n_t),srcElement,n_s,     &
                   srcNodeCount,srcElement%unknownM(n_s),commonRegions,obsJsourceM,gMatrix)
      ENDIF
      IF (obsSourceFlag(3)) THEN
       obsIndex = 2
       srcIndex = 1
       CALL localToGlobal(modeIndex,obsElement,n_t,obsNodeCount,obsElement%unknownM(n_t),srcElement,n_s,     &
                   srcNodeCount,srcElement%unknownJ(n_s), commonRegions,obsMsourceJ,gMatrix)
      ENDIF
      IF (obsSourceFlag(4)) THEN
       obsIndex = 2
       srcIndex = 2
       CALL localToGlobal(modeIndex,obsElement,n_t,obsNodeCount,obsElement%unknownM(n_t),srcElement,n_s,     &
                   srcNodeCount,srcElement%unknownM(n_s), commonRegions,obsMsourceM,gMatrix)
      ENDIF
     ENDDO
    ENDDO
!
!$OMP END DO
!
!>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>> Daughter Threads Join Here
!$OMP END PARALLEL
!
   ENDIF
   CALL ObservationArrayStatic%destroy(obsArray)
  ENDDO
 ENDDO
!
!$OMP END DO
!
!>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>> Parent Threads Join Here
!$OMP END PARALLEL
!
 CALL deleteLocals(obsJsourceJ,obsJsourceM,obsMsourceJ,obsMsourceM)
!
! Add lumped loads
 CALL addLumpedLoads(project,omega,gMatrix)
!
 END SUBROUTINE fillNormal
```

Simple and nested OMP parallelization of the GEMINI Solver matrix fill routine are each run back-to-back ten times on each test case in Table 32 to compare the performance of the two parallel methods. The execution time for test cases are measured for simple and nested parallelism methods, and speedups compared. Nested parallelism speedup compared to simple parallelism speedup is calculated using the performance comparison ratio given by:

Table 32: Test cases run to compare simple vs. nested OMP parallelism

Frequency, f (GHz)	Edges, N (# unknowns)	Solution Test	OMP Parallelism Comparison	# Outer–Inner OMP Threads
0.2998	4455	EFIE	Simple vs Nested	4-0 vs. 2-2
0.2998	4455	EFIE	Simple vs Nested	16-0 vs. 4-4
0.8994	41,415	EFIE	Simple vs Nested	4-0 vs. 2-2
0.8994	41,415	EFIE	Simple vs Nested	16-0 vs. 4-4

$$R = \frac{S_{nested}}{S_{simple}} = \frac{T_1 / T_{nested}}{T_1 / T_{simple}} \quad \rightarrow \quad R = \frac{T_{simple}}{T_{nested}} \qquad (30)$$

where equation (26) for speedup has been applied. T_{simple} and T_{nested} are the simple and nested execution times required to perform the matrix fill operation, respectively. Performance ratios with 3σ (99¾% interval = R ± 3σ) are shown in Table 33.

Table 33: Performance ratios comparing nested to simple parallelism speedup

Case	Edges, N (# unknowns)	Nodes	# MPI Processes	Simple vs Nested OMP Threads	Average Performance Ratio, R	3σ (99¾%)
1	4455	1	4	4-0 vs. 2-2	0.972	0.040
2	4455	1	1	16-0 vs. 4-4	0.974	0.040
3	41,415	1	4	4-0 vs. 2-2	0.945	0.030
4	41,415	2	8	4-0 vs. 2-2	0.900	0.029
5	41,415	4	16	4-0 vs. 2-2	0.889	0.022
6	41,415	1	1	16-0 vs. 4-4	0.971	0.045
7	41,415	2	2	16-0 vs. 4-4	0.973	0.042
8	41,415	4	4	16-0 vs. 4-4	0.972	0.028

A color graph of the performance ratios with 99¾% confidence intervals for all 8 cases is shown in Figure 45.

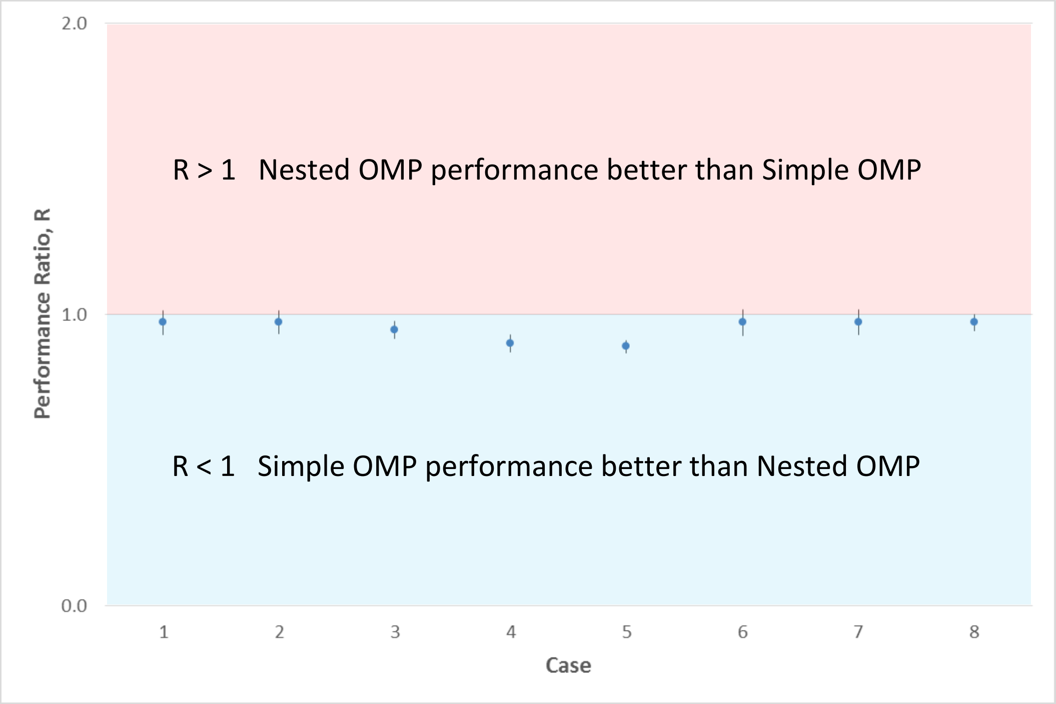


Figure 45: Simple vs. Nested OMP Performance ratios with 99¾% intervals

In all 8 cases, simple OMP threading slightly outperforms nested OMP threading. The following considerations should be taken into account [18]:

- Increased demand for forking and joining of threads at the inner parallel regions requires an extremely efficient thread runtime library and operating system support.
- Synchronization overhead will increase because of the implicit barrier synchronization at inner parallel regions.

Increased forking/joining demand and overhead are responsible for slightly poorer performance in nested OMP parallelism. Any performance gained by threading the inner loop is overshadowed by increased forking/joining demand and overhead. Thus, simple OMP threading will be used in the MPI-OMP hybrid parallelization of GEMINI Solver.

Chapter 5. Testing MPI-OMP Hybrid Parallelization

5.1. Final Implementation

Consider the results from section 3.3.4 on the performance tests for N = 208K using 224 MPI processes for both 16 and 8 MPI processes per node as shown in Table 34.

Table 34: N=208K case using 224 MPI processes for 8 and 16 MPIs/node

Edges, N (# unknowns)	# DLX Nodes	# MPI Processes / DLX Node	# MPI Processes	Idle Cores	Fill Time (hours)	Factor Time (hours)
207,663	14	16	224	0	1.44	2.13
207,663	28	8	224	224	1.27	1.63

Reducing the MPI processes from 16 to 8 per node decreases both the matrix fill and factor times. Performance increase is attributed to the reduction in node memory needed by 8 MPI processes compared to 16. However, reducing the number of MPI process per node results in 224 idle cores. To minimize inefficiencies when reducing the number of MPI processes, the matrix fill routine can incorporate OMP threading to utilize idle cores and increase matrix fill performance. A test done for N = 208K using 224 MPI processes with 8 MPI processes per node and 2 OMP threads yields the results in Table 35.

Table 35: N=208K case using 224 MPI processes with 8 MPIs/node and 2 OMP threads

Edges, N (# unknowns)	# DLX Nodes	# MPI Processes / DLX Node	# MPI Processes	# OMP Threads	Fill Time (hours)	Factor Time (hours)
207,663	28	8	224	2	1.09	1.67

Matrix fill performance improves as expected while matrix factor performance remains essentially constant as expected. Implementation of MPI-OMP hybrid parallelization incorporates the existing MPI parallelization between concurrent MPI processes with simple OMP threading of the matrix fill routine within each MPI process. Figure 46 on the following page illustrates an example of hybrid parallelization using four MPI processes with four simple OMP threads per MPI process.

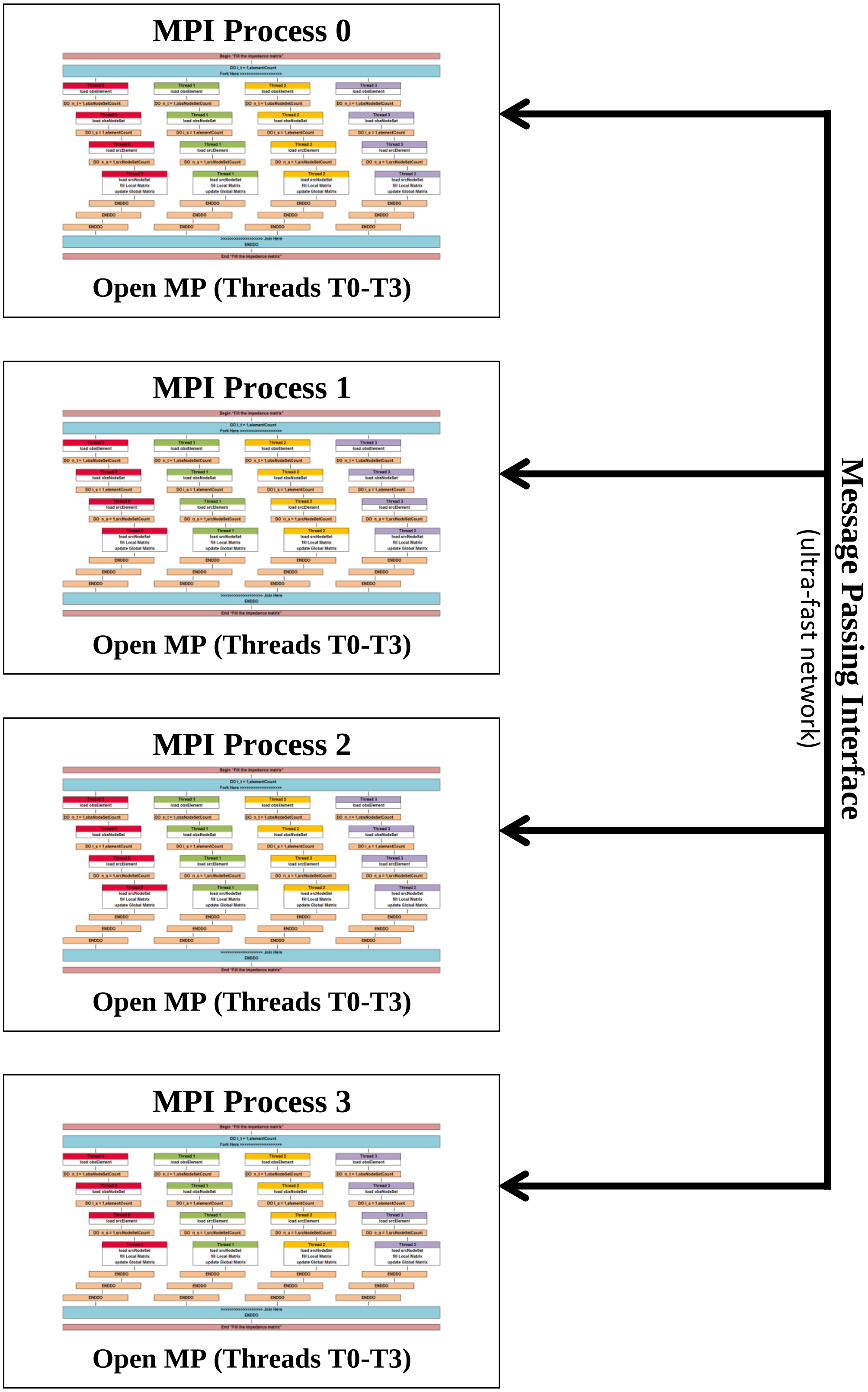


Figure 46: Illustration for 4-MPI / 4-Simple OMP hybrid parallelization

In building the final implementation of MPI-OMP hybrid parallelization of GEMINI Solver, several modifications need to be made to the original v3.0 program to:

- add capability to measure the matrix fill execution time for each MPI process;
- add fine level parallelization to the matrix fill routine with OMP threading;
- resolve race conditions arising from OMP threading.

The following GEMINI Solver program changes are made:

1. Add a wall clock time measurement feature to the matrix fill routine contained within the *solution_nonperiodic a*nd *solution_periodic* subroutines of the **solution_m.f90** module. This tool allows measurement of the matrix fill time for each executing MPI process. Modifications #4 through #6 listed in Appendix B add matrix fill time measurement capability for nonperiodic solutions while modifications #7 through #9 add this capability for periodic solutions.

2. Parallelize the outer loop of the matrix fill routine on the following statements:

   ```
   DO i_t = 1,elementCount
   . . .
   ENDDO
   ```

   contained within the *fillNormal* subroutine of the **globalmatrix_m.f90** module. Modifications #24 and #27 listed in Appendix B add OMP parallelization.

3. Correct the OMP race condition occurring on the following statements:

   ```
   obsBasis => project%obsBases%at(obsElement%type() + 1)
   srcBasis => project%srcBases%at(srcElement%type() + 1)
   ```

   inside the outer observation element loop and inner source element loop, respectively, within the *fillNormal* subroutine of the of the **globalmatrix_m.f90** module. The race condition causes the program to intermittently crash when one OMP thread attempts to deallocate memory that another thread had previously deallocated but had not had time to set the notification flag. Modifications #1 and #23 listed in Appendix B solve the race condition.

4. Resolve the OMP race condition occurring on the following two statements:

   obsElement => project%elements%at(i_t)
   srcElement => project%elements%at(i_s)

   inside the outer observation element loop and inner source element loop, respectively, within the *fillNormal* subroutine of the of the **globalmatrix_m.f90** module. The race condition causes a small intermittent error in the solution to the edge currents when two or more OMP threads write access the same variable simultaneously. Modification #28 listed in Appendix B solves the race condition for the first statement and modification #29 solves it for the second statement. To avoid the race condition, the modifications require use of an $OMP CRITICAL directive to force all threads to work one at a time when executing the above statements. Unfortunately, this directive reduces the parallel performance of the matrix fill routine; however, no other solution can be found with the current structure of GEMINI Solver. In the future design of GEMINI Solver, thread safety needs to be incorporated into subroutines executed by the above statements.

5. Correct the race condition on the following statements:

   value = gMatrix%getValue(rowDOF,columnDOF)
   . . .
   CALL gMatrix%putValue(rowDOF,columnDOF,value)

   within the *localToGlobal* subroutine of the **globalmatrix_m.f90** module. The race condition causes a large error in the solution to the edge currents because two or more OMP threads often write to the gMatrix variable simultaneously. Since gMatrix contains the global matrix, each thread must have sole access when updating this matrix. Modifications #25 and #26 listed in Appendix B solve the race condition. These modifications require use of the $OMP CRITICAL directive to force all threads to execute one at a time when writing to gMatrix. Although this directive reduces the parallel performance of the matrix fill routine; this solution must be implemented to protect the global matrix. No way could be found around the need to force threads to access gMatrix one at a time.

6. Resolve intermittent race conditions caused by the following variables:

    CommonRegionStatic
    HomogenousGreensFunction
    LocalMatrix
    rs, unitNormal, l_vec, jac, srcWghtJacobian, srcArray
    G, Kphi, Kpsi, Pz, Qz, gradG, grad_Kphi, grad_Kpsi, grad_Pz, grad_Qz
    Ga, Gf, curl_Ga, curl_Gf
    xiTemp, wghtTemp
    classPointer

    located within various modules of GEMINI Solver. Modifications #10 through #19 and #21 listed in Appendix B solve the race conditions caused when two or more OMP threads write access one of the above variables simultaneously. These modifications require use of the $OMP TREADPRIVATE directive to ensure each thread receives its own a private copy of each above variable.

Table 36 shows the test cases utilized in the final hybrid MPI-OMP parallelization of GEMINI Solver. The first five test cases include a performance comparison between OMP threading and no OMP threading. The performance comparison for the fifth test case can only be made for 32 nodes. The memory requirements on 28 and 30 nodes do not allow execution without OMP threading. The last two cases have large problem sizes requiring memory sizes that will not execute without OMP threading.

Table 36: Test cases for final implementation of hybrid MPI-OMP GEMINI Solver v3.0

Frequency, f (GHz)	Edges, N (# unknowns)	Solution Test	# DLX Nodes	#MPI/node – #OMP Thread Combination	OMP vs. No OMP Performance Comparison
0.8994	41,415	EFIE	1,2,4	1-16, 2-8, 4-4, 8-2, 16-1	Yes
1.1992	74,211	EFIE	4,8,12	1-16, 2-8, 4-4, 8-2, 16-1	Yes
1.7988	167,652	EFIE	16, 20,24	1-16, 2-8, 4-4, 8-2, 16-1	Yes
2.0000	207,663	EFIE	16, 20,24	2-8, 4-4, 8-2, 16-1	Yes
2.3984	298,863	EFIE	28,30,32	2-8, 4-4, 8-2	Yes
2.5000	326,430	EFIE	30,32	2-8, 4-4	No
2.5624	342,087	EFIE	32	2-8	No

Allocating run time on the DLX requires following certain rules for batch job submission as well as waiting for resources to become available. DLX batch scripts are created, with sanity checks [22], to execute run scripts, which in turn execute GEMINI Solver v3.0 test runs. Sanity checks ensure that batch submission rules are followed. Without sanity checks, batch submissions could remain in que indefinitely, never being allocated run time. Table 37 shows an example DLX batch script with sanity checks.

Table 37: DLX batch script with sanity checks

```
  Linux Batch Script mpgs_batchall.sh

#!/bin/sh

pnodes=$1                                   # number of physical nodes
time=$2                                     # max time in batch que
cper_pnode=16                               # cores per physical node (Compute)
ncores=$(($cper_pnode*$pnodes))             # number of total cores requested

# setup options
rdir="/home/bljo222/gemini_v3/Test"         # SET PATH TO YOUR SBATCH SCRIPT AND OUTPUT FILE
script=$rdir/mpgs_runall-1.sh
outfil="$rdir/mpgs_screenall_p$pnodes-1.txt"

# Select the queue and make sure run time meets requirements
if [ $ncores -ge 512 ]; then
  part="Short"
  if [ $time -gt 1440 ]; then
    echo "Time to long for Short queue. Exiting"
    exit
  fi
elif [ $ncores -ge 65 ]; then
  part="Med"
  if [ $time -gt 10080 ]; then
    echo "Time to long for Med queue. Exiting"
    exit
  fi
elif [ $ncores -ge 16 ]; then
  part="Long"
  if [ $time -gt 43200 ]; then
    echo "Time to long for Long queue. Exiting"
    exit
  fi
else
  part="debug"
fi

# Submit the batch job
echo "sbatch --exclusive -p $part -t $time -N $pnodes -n $ncores -o $outfil $script $pnodes"
sbatch --exclusive --no-requeue -p $part -t $time -N $pnodes -n $ncores -o $outfil $script $pnodes
```

Once DLX resources become available, it is beneficial to run multiple tests while one has the resources. DLX run scrips are created [23] to execute five MPI/node – OMP thread combinations for each test case in Table 36. Table 38 shows an example DLX run script.

Table 38: DLX run script to execute GEMINI Solver v3.0 test sets

```
  Linux Run Script mpgs_runall.sh

#!/bin/sh

pnodes=$1                                         # number of physical nodes imported from ”mpgs_batchall.sh”
source /etc/bashrc                                # may need this to initialize module system
module load mpi/openmpi/intel/1.8.2               # load MPI module
#----------------------------------------------------------------------------------------------------------------------------------
# Run Combo #1 (1 MPI/node & 16 OMP/MPI  processes) with OMP Nested Parallelism NOT Enabled
exe="/home/bljo222/gemini_v3/Test/GEMINI_solver"  # path to GEMINI Solver executable
tper_pnode=1                                      # MPI processes per physical node
cper_vnode=16                                     # OMP threads  used by each MPI process
vnodes=$(($tper_pnode*$pnodes))                   # total number of MPI processes on all nodes
wkdir=/home/bljo222/gemini_v3/Test/solver_p$pnodes\_v$vnodes\_cv$cper_vnode\_tp$tper_pnode
mkdir -p  $wkdir
cd $wkdir
infil="/home/bljo222/gemini_v3/Test/mpgs_inputall-1.txt"
mpirun -n $vnodes --map-by ppr:$tper_pnode:node:pe=$cper_vnode --bind-to core -report-bindings $exe < $infil
#----------------------------------------------------------------------------------------------------------------------------------
# Run Combo #2 (2 MPI/node & 8 OMP/MPI  processes) with OMP Nested Parallelism NOT Enabled
exe="/home/bljo222/gemini_v3/Test/GEMINI_solver"  # path to GEMINI Solver executable
tper_pnode=2                                      # MPI processes per physical node
cper_vnode=8                                      # OMP threads  used by each MPI process
vnodes=$(($tper_pnode*$pnodes))                   # total number of MPI processes on all nodes
wkdir=/home/bljo222/gemini_v3/Test/solver_p$pnodes\_v$vnodes\_cv$cper_vnode\_tp$tper_pnode
mkdir -p  $wkdir
cd $wkdir
infil="/home/bljo222/gemini_v3/Test/mpgs_inputall-1.txt"
mpirun -n $vnodes --map-by ppr:$tper_pnode:node:pe=$cper_vnode --bind-to core -report-bindings $exe < $infil
#----------------------------------------------------------------------------------------------------------------------------------
# Run Combo #3 (4 MPI/node & 4 OMP/MPI  processes) with OMP Nested Parallelism NOT Enabled
exe="/home/bljo222/gemini_v3/Test/GEMINI_solver"  # path to GEMINI Solver executable
tper_pnode=4                                      # MPI processes per physical node
cper_vnode=4                                      # OMP threads  used by each MPI process
vnodes=$(($tper_pnode*$pnodes))                   # total number of MPI processes on all nodes
wkdir=/home/bljo222/gemini_v3/Test/solver_p$pnodes\_v$vnodes\_cv$cper_vnode\_tp$tper_pnode
mkdir -p  $wkdir
cd $wkdir
infil="/home/bljo222/gemini_v3/Test/mpgs_inputall-1.txt"
mpirun -n $vnodes --map-by ppr:$tper_pnode:node:pe=$cper_vnode --bind-to core -report-bindings $exe < $infil
#----------------------------------------------------------------------------------------------------------------------------------
# Run Combo #4 (8 MPI/node & 2 OMP/MPI  processes) with OMP Nested Parallelism NOT Enabled
exe="/home/bljo222/gemini_v3/Test/GEMINI_solver"  # path to GEMINI Solver executable
tper_pnode=8                                      # MPI processes per physical node
cper_vnode=2                                      # OMP threads  used by each MPI process
vnodes=$(($tper_pnode*$pnodes))                   # total number of MPI processes on all nodes
wkdir=/home/bljo222/gemini_v3/Test/solver_p$pnodes\_v$vnodes\_cv$cper_vnode\_tp$tper_pnode
mkdir -p  $wkdir
cd $wkdir
infil="/home/bljo222/gemini_v3/Test/mpgs_inputall-1.txt"
mpirun -n $vnodes --map-by ppr:$tper_pnode:node:pe=$cper_vnode --bind-to core -report-bindings $exe < $infil
#----------------------------------------------------------------------------------------------------------------------------------
# Run Combo #5 (16 MPI/node & 1 OMP/MPI  processes) with OMP Nested Parallelism NOT Enabled
exe="/home/bljo222/gemini_v3/Test/GEMINI_solver"  # path to GEMINI Solver executable
tper_pnode=16                                     # MPI processes per physical node
cper_vnode=1                                      # OMP threads  used by each MPI process
vnodes=$(($tper_pnode*$pnodes))                   # total number of MPI processes on all nodes
wkdir=/home/bljo222/gemini_v3/Test/solver_p$pnodes\_v$vnodes\_cv$cper_vnode\_tp$tper_pnode
mkdir -p  $wkdir
cd $wkdir
infil="/home/bljo222/gemini_v3/Test/mpgs_inputall-1.txt"
mpirun -n $vnodes --map-by ppr:$tper_pnode:node:pe=$cper_vnode --bind-to core -report-bindings $exe < $infil
```

5.2. GEMINI Solver Results

GEMINI Solver v3.0, incorporating hybrid MPI/Simple OMP parallelization, is executed on the DLX cluster for the N = 41.4 K, f = 0.8994 GHz test case. Table 39 shows the average matrix fill time, factor time, and required memory per node for three trials of each configuration: #DLX Nodes **:** #MPI processes/node **:** #OMP threads.

Table 39: GEMINI Solver v3.0 Matrix Fill & Factor Times for N=41.4K, f=0.8994GHz

DLX Nodes	MPIs/node	OMP Threads	# Cores	Matrix Fill Time (min)	Matrix Factor Time (min)	Required Memory/node
1	1	16	16	70.274	164.584	50.1%
1	2	8	16	35.698	89.627	50.4%
1	4	4	16	34.189	45.276	51.0%
1	8	2	16	35.009	22.819	52.2%
1	16	1	16	33.263	11.100	54.6%
2	1	16	32	55.413	96.921	29.8%
2	2	8	32	21.435	48.571	30.1%
2	4	4	32	19.125	24.519	30.8%
2	8	2	32	25.835	11.508	32.1%
2	16	1	32	19.793	5.666	34.8%
4	1	16	64	33.913	56.171	19.3%
4	2	8	64	12.440	27.987	19.7%
4	4	4	64	14.116	12.401	20.3%
4	8	2	64	15.346	5.940	21.7%
4	16	1	64	11.559	2.981	24.4%

Matrix factor times follow earlier trends for MPI only parallelization. Graphs of matrix fill times and required memory usage are shown in Figure 47 and Figure 48, respectively.

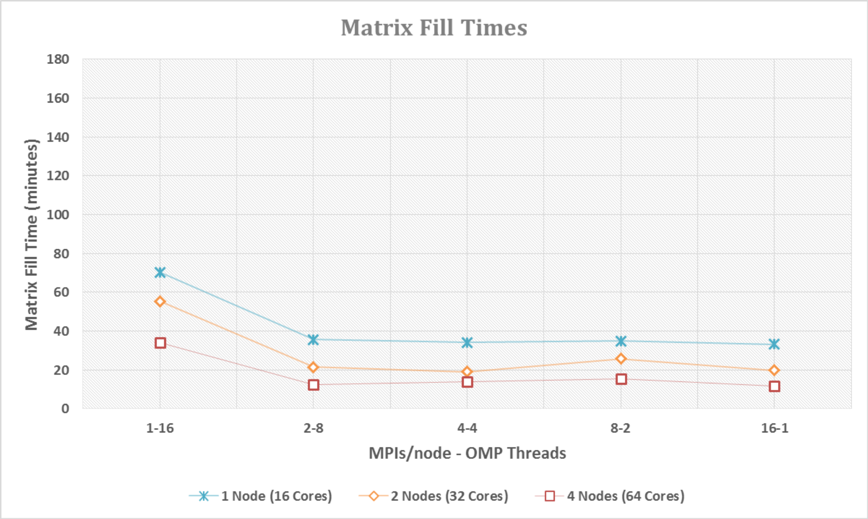


Figure 47: Matrix fill times vs. MPI-OMP combination for N=41.4K, f=0.8994GHz

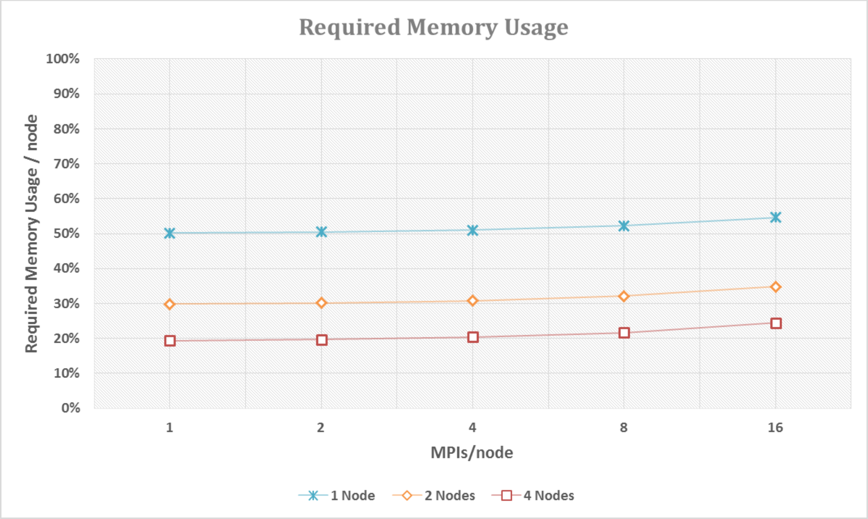


Figure 48: Required memory/node vs. MPIs/node for N=41.4K, f=0.8994GHz

As the number of MPI processes per node is reduced by powers of two, OMP threads are increased by the same factor to compensate and share more of the computational workload. As expected, each line graph in Figure 47 is relatively constant for all MPI-OMP combinations except the 1-16 combination. At least 2 MPI processes per node must execute for OMP threading to be effective. In addition, as the number of MPI processes per node is increased, the required physical memory usage per node should increase as more copies of the mesh are needed. As expected, each line graph in Figure 48 increases as the number of MPI processes increases. When reducing the number of MPI processes, the matrix fill routine incorporates simple OMP threading to utilize idle cores and increase matrix fill performance. Matrix fill time, speedup, and efficiency graphs comparing OMP threading to the same cases without OMP threading are shown in Figure 49 on the next page. Matrix fill time comparing OMP threading to the same cases without OMP threading along with SEP tables are shown in Figure 50 on the page following next. The highest, most efficient speedup occurs for 2 MPI processes and 8 OMP threads with SEP = 1.9, 1.8, and 1.6 for 1, 2, and 4 nodes, respectively. The 2-8 combination requires the least MPI processes per node for effective OMP threading, has the highest, most efficient speedup, and uses the least physical memory per node.

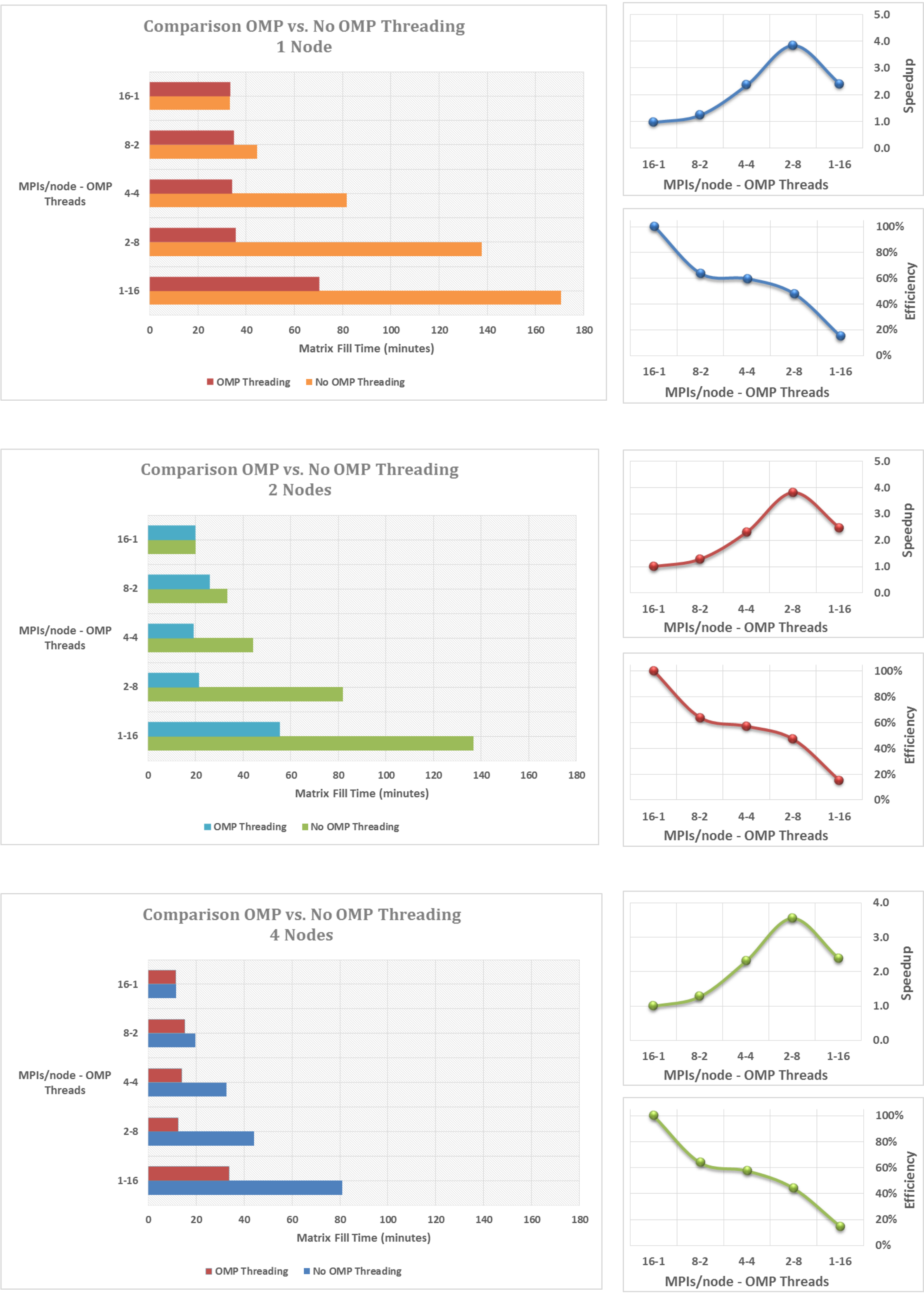
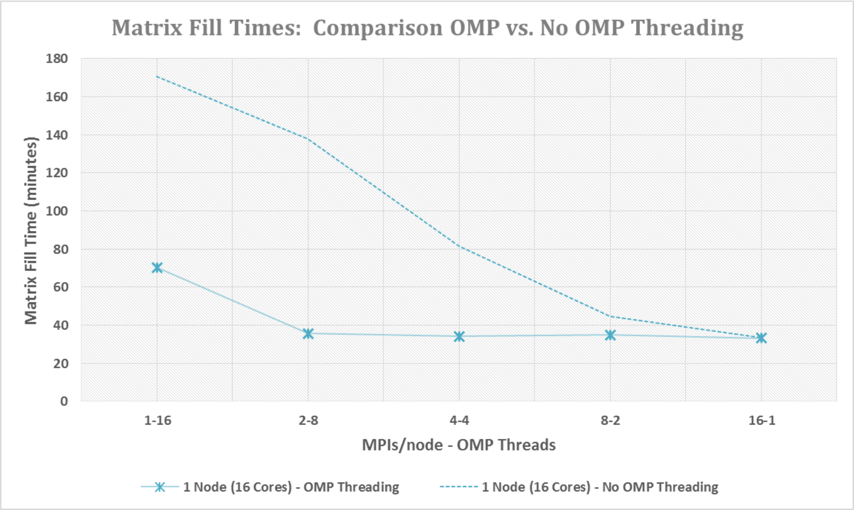
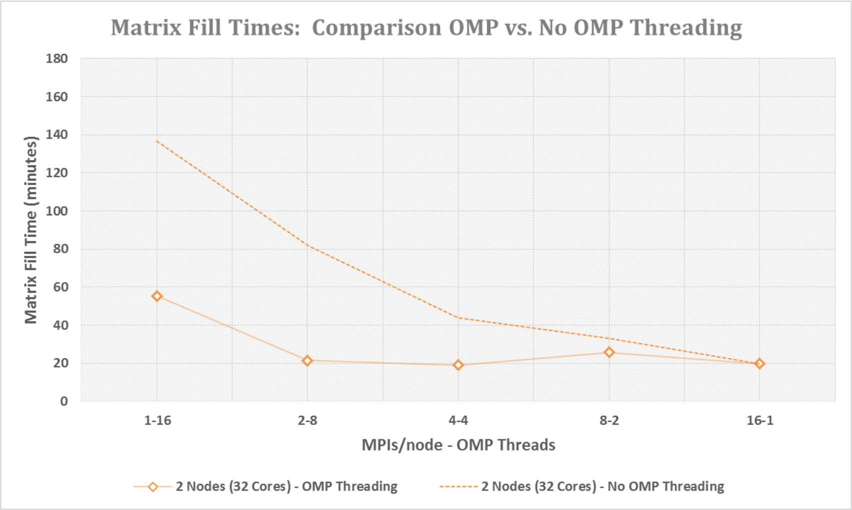


Figure 49: OMP threading vs. No OMP threading for N=41.4K, f=0.8994GHz



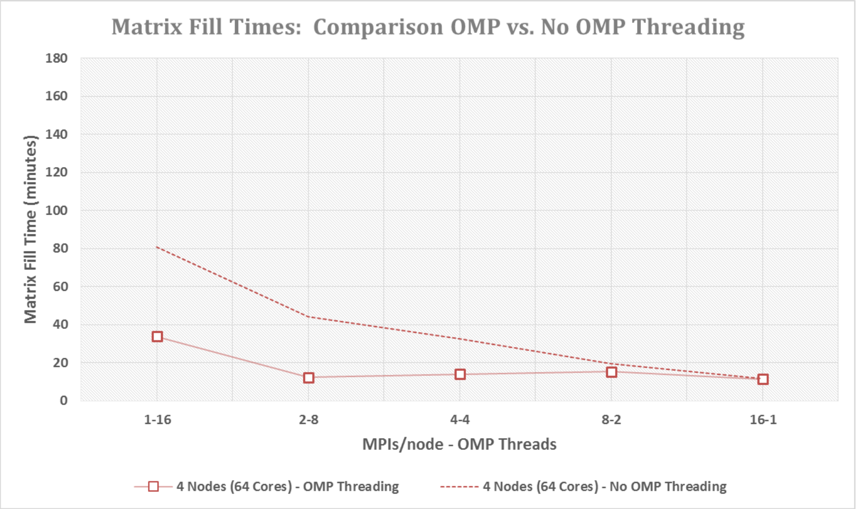
1 Node

MPI-OMP	SEP
16-1	1.0
8-2	0.8
4-4	1.4
2-8	1.9
1-16	0.4



2 Nodes

MPI-OMP	SEP
16-1	1.0
8-2	0.8
4-4	1.3
2-8	1.8
1-16	0.4



4 Nodes

MPI-OMP	SEP
16-1	1.0
8-2	0.8
4-4	1.3
2-8	1.6
1-16	0.4

Figure 50: Matrix Fill Times: OMP vs. No OMP for N=41.4K, f=0.8994GHz

GEMINI Solver v3.0, incorporating hybrid MPI/Simple OMP parallelization, is executed on the DLX cluster for the N = 74.2 K, f = 1.1992 GHz test case. Table 40 shows the average matrix fill time, factor time, and required memory per node for three trials of each configuration: #DLX Nodes **:** #MPI processes/node **:** #OMP threads.

Table 40: GEMINI Solver v3.0 Matrix Fill & Factor Times for N=74.4K, f=1.1992GHz

DLX Nodes	MPIs/node	OMP Threads	# Cores	Matrix Fill Time (min)	Matrix Factor Time (min)	Required Memory/node
2	1	16	32	158.917	523.943	74.9%
2	2	8	32	68.697	258.208	75.4%
2	4	4	32	60.710	129.531	76.6%
2	8	2	32	82.478	63.256	78.7%
2	16	1	32	63.205	31.477	83.1%
4	1	16	64	107.368	284.448	42.3%
4	2	8	64	39.673	138.766	42.7%
4	4	4	64	44.105	65.840	44.2%
4	8	2	64	47.943	32.161	45.9%
4	16	1	64	35.470	16.190	50.7%
8	1	16	128	62.560	158.237	25.8%
8	2	8	128	27.587	71.209	26.0%
8	4	4	128	25.716	33.699	27.1%
8	8	2	128	28.626	16.877	29.4%
8	16	1	128	19.019	8.354	34.4%

Matrix factor times follow earlier trends for MPI only parallelization. Graphs of matrix fill times and required memory/node are shown in Figure 51 and Figure 52, respectively.

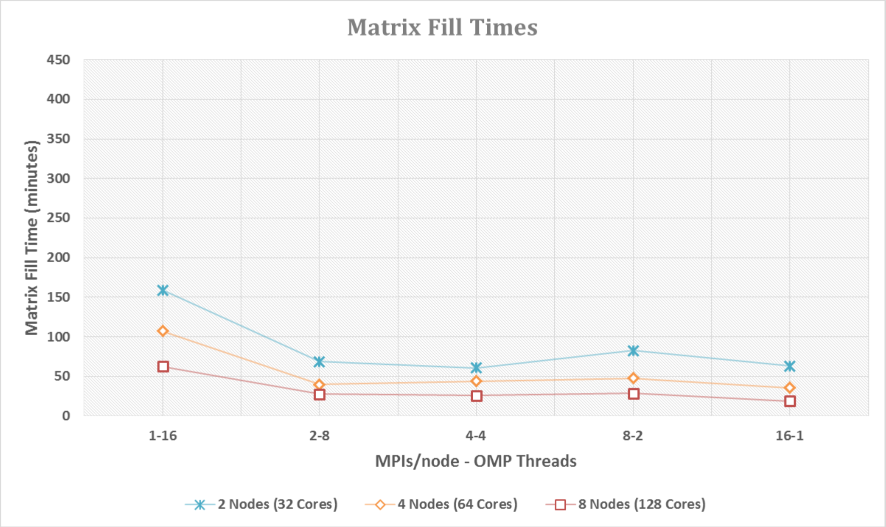


Figure 51: Matrix fill times vs. MPI-OMP combination N=74.2K, f=1.1992GHz

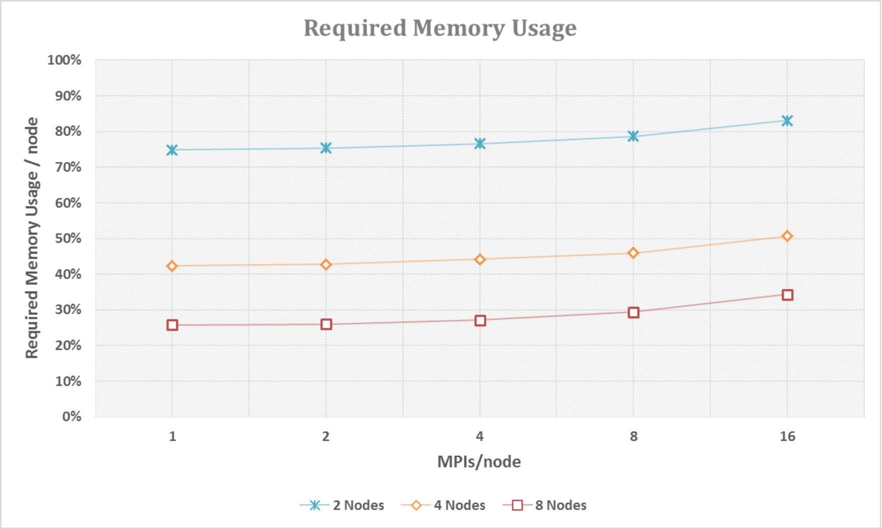


Figure 52: Required memory/node vs. MPIs/node for N=74.2K, f=1.1992GHz

As the number of MPI processes per node is reduced by powers of two, OMP threads are increased by the same factor to compensate and share more of the computational workload. As expected, each line graph in Figure 51 is relatively constant for all MPI-OMP combinations except the 1-16 combination. At least 2 MPI processes per node must execute for OMP threading to be effective. In addition, as the number of MPI processes per node is increased, the required physical memory usage per node should increase as more copies of the mesh are needed. As expected, each line graph in Figure 52 increases as the number of MPI processes increases. When reducing the number of MPI processes, the matrix fill routine incorporates simple OMP threading to utilize idle cores and increase matrix fill performance. Matrix fill time, speedup, and efficiency graphs comparing OMP threading to the same cases without OMP threading are shown in are shown in Figure 53 on the next page. Matrix fill time comparing OMP threading to the same cases without OMP threading along with SEP tables are shown in Figure 54 on the page following next. The highest, most efficient speedup occurs for 2 MPI processes and 8 OMP threads with SEP = 1.9, 1.6, and 1.8 for 2, 4, and 8 nodes, respectively. The 2-8 combination requires the least MPI processes per node for effective OMP threading, has the highest, most efficient speedup, and uses the least physical memory per node.

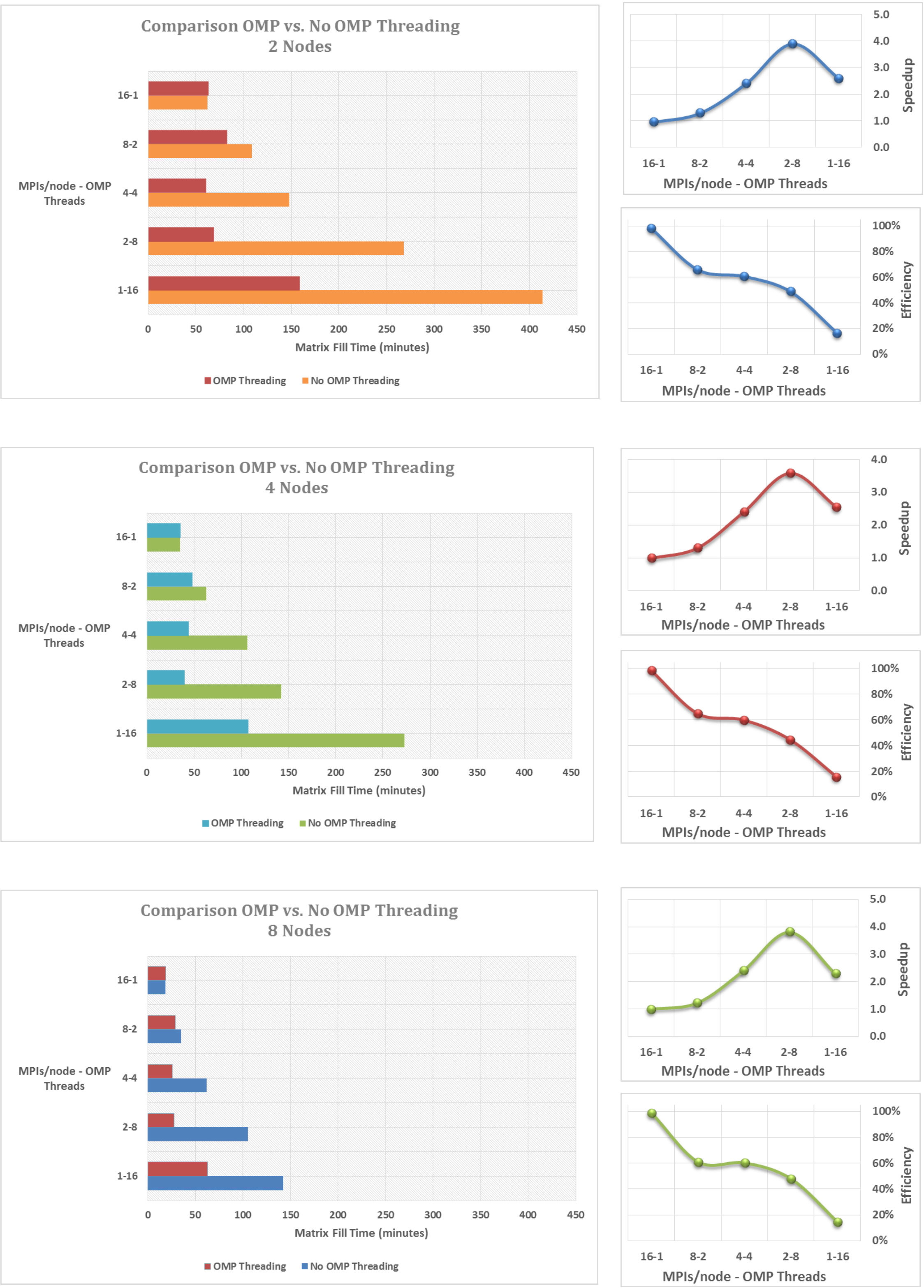
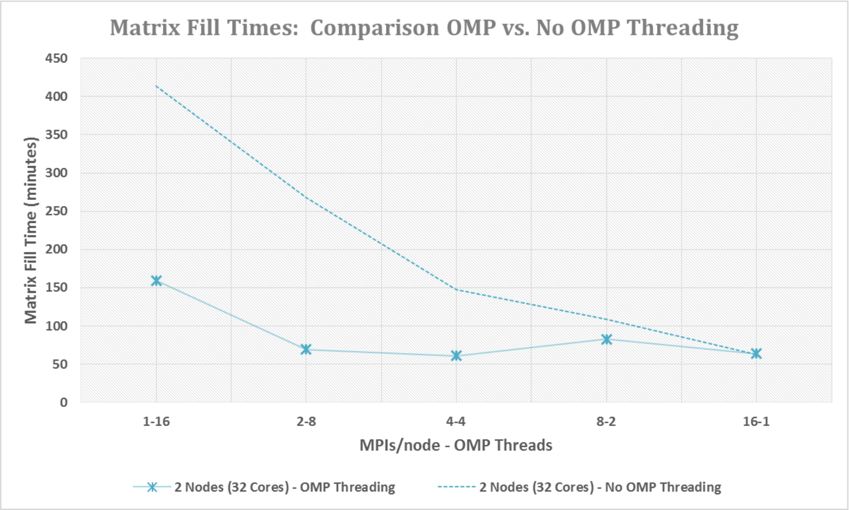
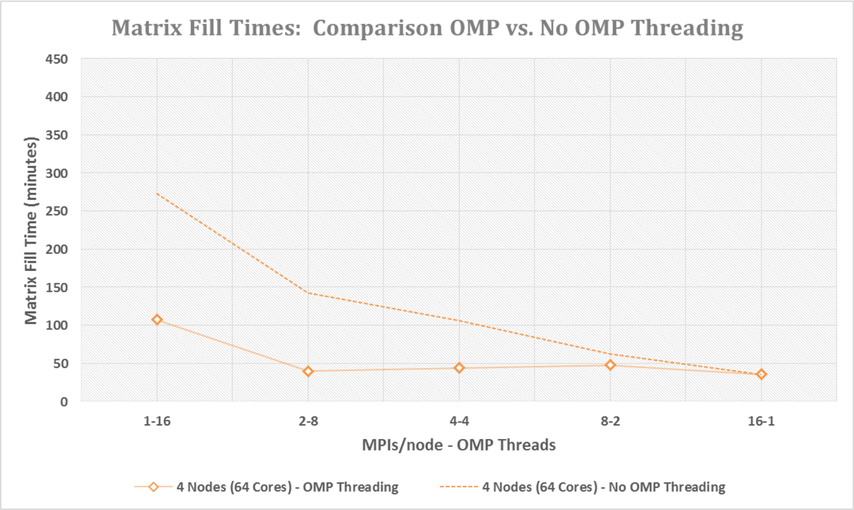


Figure 53: OMP threading vs. No OMP threading for N=74.2K, f=1.1992GHz



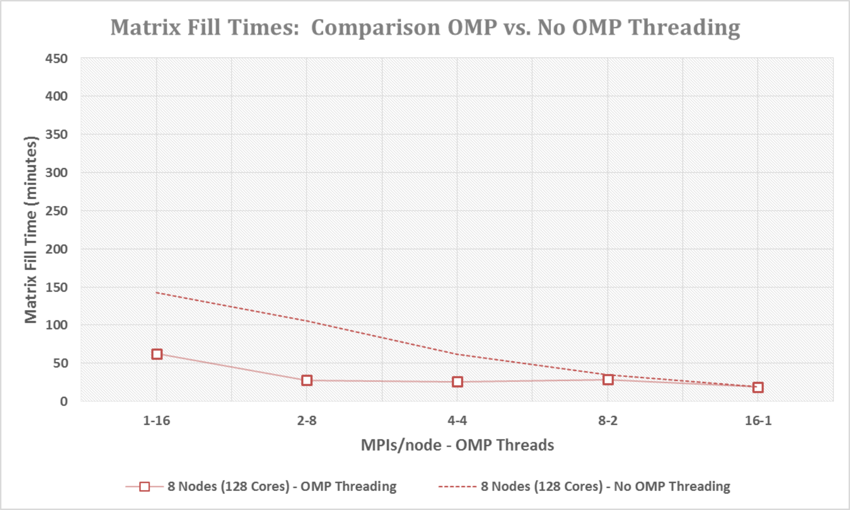
2 Nodes

MPI-OMP	SEP
16-1	1.0
8-2	0.9
4-4	1.5
2-8	1.9
1-16	0.4



4 Nodes

MPI-OMP	SEP
16-1	1.0
8-2	0.8
4-4	1.4
2-8	1.6
1-16	0.4



8 Nodes

MPI-OMP	SEP
16-1	1.0
8-2	0.7
4-4	1.4
2-8	1.8
1-16	0.3

Figure 54: Matrix Fill Times: OMP vs. No OMP for N=74.2K, f=1.1992GHz

GEMINI Solver v3.0, incorporating hybrid MPI/Simple OMP parallelization, is executed on the DLX cluster for the N = 168 K, f = 1.7998 GHz test case. Table 41 shows the average matrix fill time, factor time, and required memory per node for three trials of each configuration: #DLX Nodes **:** #MPI processes/node **:** #OMP threads.

Table 41: GEMINI Solver v3.0 Matrix Fill & Factor Times for N=168K, f=1.7998GHz

DLX Nodes	MPIs/node	OMP Threads	# Cores	Matrix Fill Time (hr)	Matrix Factor Time (hr)	Required Memory/node
16	1	16	256	3.924	12.911	52.3%
16	2	8	256	1.394	6.194	53.2%
16	4	4	256	1.291	3.117	55.5%
16	8	2	256	1.298	1.537	60.5%
16	16	1	256	0.956	0.773	70.3%
20	1	16	320	3.443	10.204	43.2%
20	2	8	320	1.166	5.102	45.3%
20	4	4	320	0.972	2.510	47.2%
20	8	2	320	1.115	1.251	51.3%
20	16	1	320	0.688	0.640	62.7%
24	1	16	384	2.912	8.667	37.4%
24	2	8	384	0.945	4.253	38.6%
24	4	4	384	0.853	2.117	41.0%
24	8	2	384	0.878	1.062	46.7%
24	16	1	384	0.603	0.537	56.5%

Matrix factor times follow earlier trends for MPI only parallelization. Graphs of matrix fill times and required memory/node are shown in Figure 55 and Figure 56, respectively.

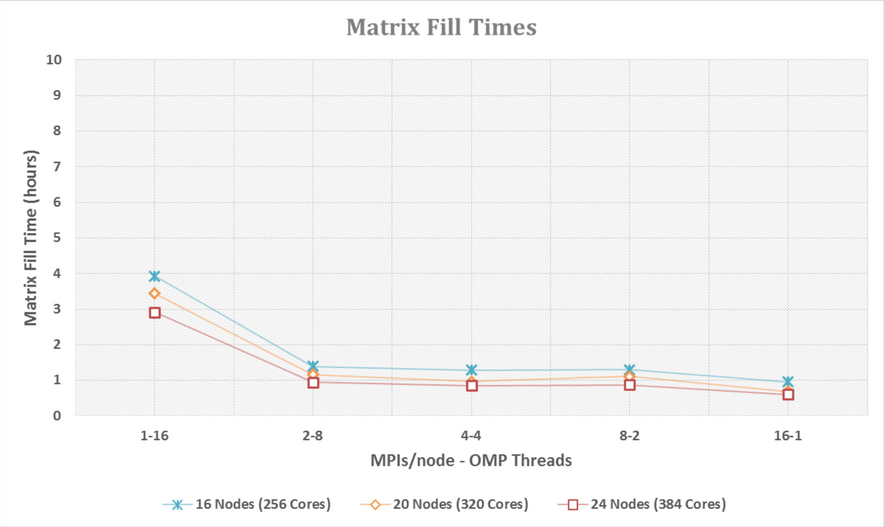


Figure 55: Matrix fill times vs. MPI-OMP combination for N=168K, f=1.7998GHz

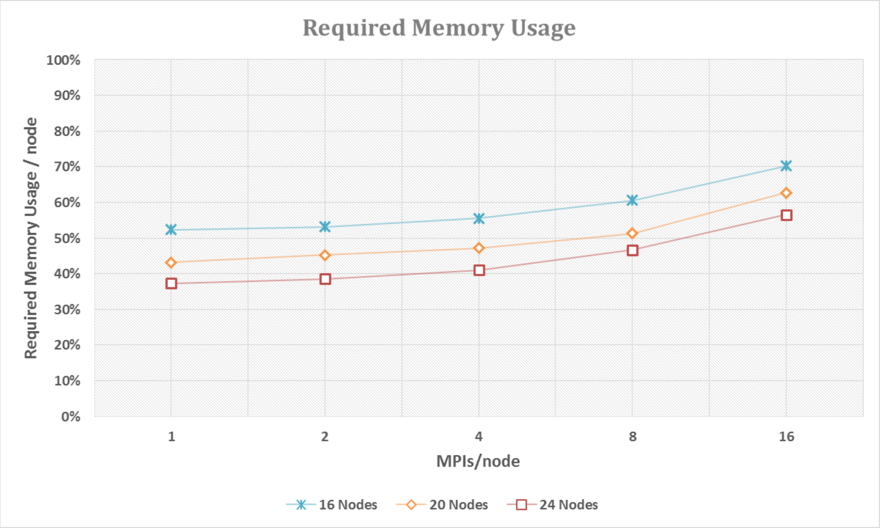


Figure 56: Required memory/node vs. MPIs/node for N=168K, f=1.7998GHz

As the number of MPI processes per node is reduced by powers of two, OMP threads are increased by the same factor to compensate and share more of the computational workload. As expected, each line graph in Figure 55 is relatively constant except the 1-16 combination. At least 2 MPI processes per node must execute for OMP threading to be effective. In addition, as the number of MPI processes per node is increased, the required physical memory usage per node should increase as more copies of the mesh are needed. As expected, each line graph in Figure 56 increases as the number of MPI processes increases. When reducing the number of MPI processes, the matrix fill routine incorporates simple OMP threading to utilize idle cores and increase matrix fill performance. Matrix fill time, speedup, and efficiency graphs comparing OMP threading to the same cases without OMP threading are shown in Figure 57 on the next page. Matrix fill time comparing OMP threading to the same cases without OMP threading along with SEP tables are shown in Figure 58 on the page following next. The highest, most efficient speedup occurs for 2 MPI processes and 8 OMP threads with SEP = 2.0, 1.8, and 1.9 for 16, 20, and 24 nodes, respectively. The 2-8 combination requires the least MPI processes per node for effective OMP threading, has the highest, most efficient speedup, and uses the least physical memory per node.

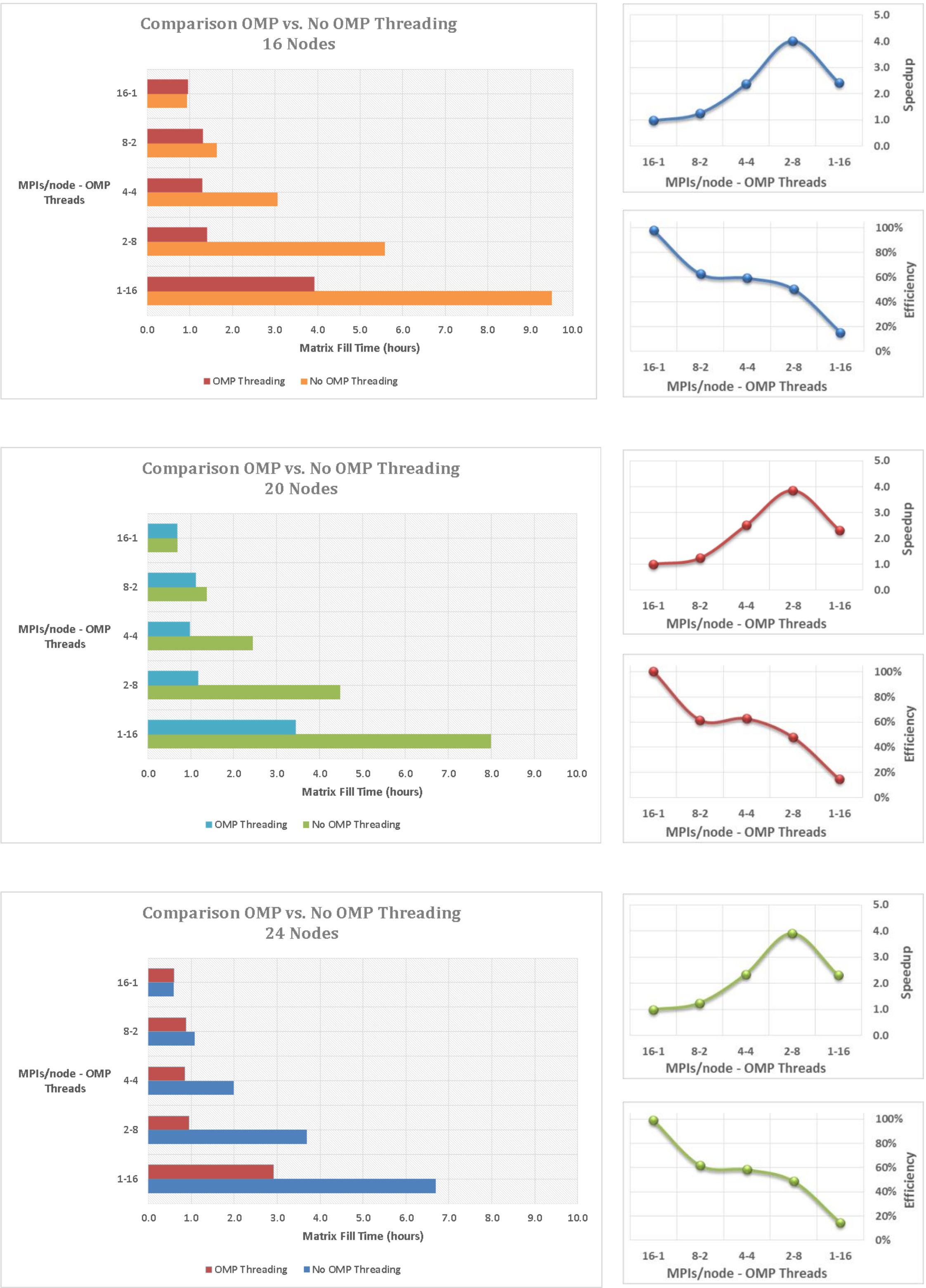
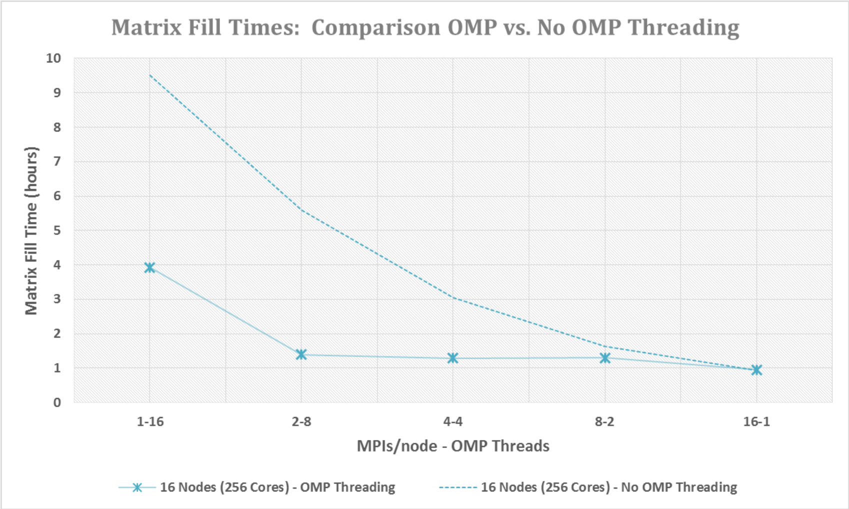
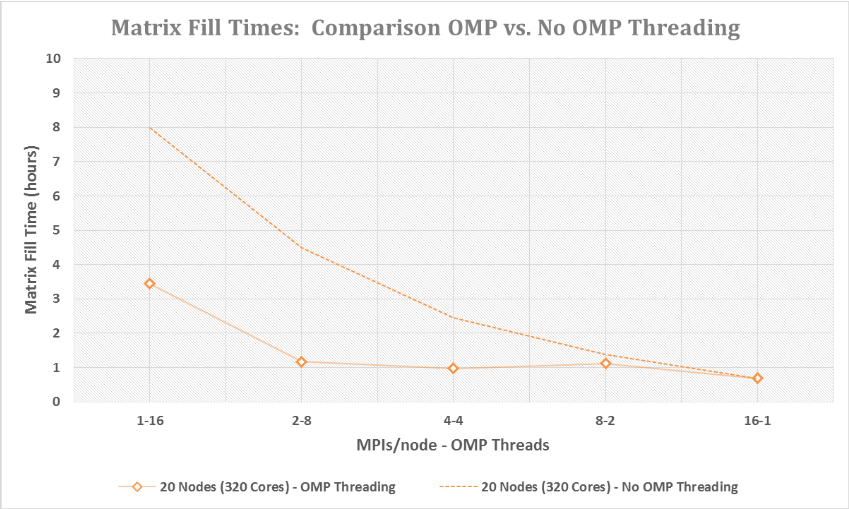


Figure 57: OMP threading vs. No OMP threading for N=168K, f=1.7998GHz



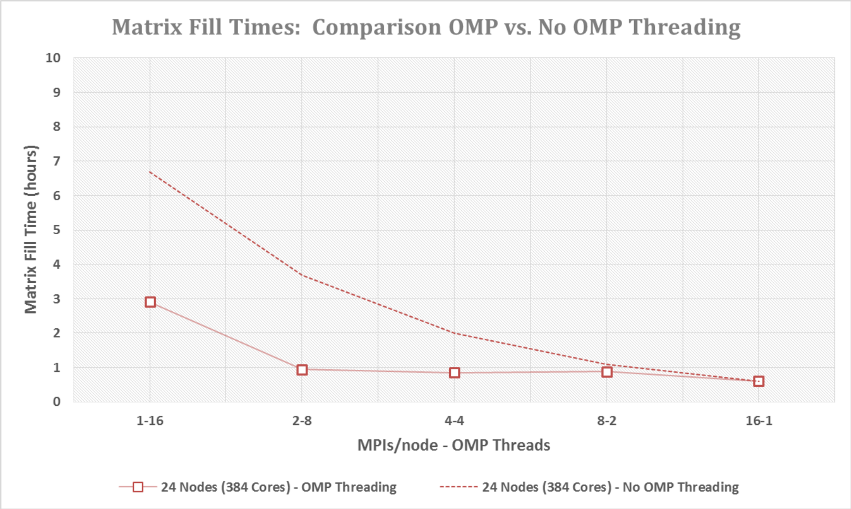
16 Nodes

MPI-OMP	SEP
16-1	1.0
8-2	0.8
4-4	1.4
2-8	2.0
1-16	0.4



20 Nodes

MPI-OMP	SEP
16-1	1.0
8-2	0.8
4-4	1.6
2-8	1.8
1-16	0.3



24 Nodes

MPI-OMP	SEP
16-1	1.0
8-2	0.8
4-4	1.4
2-8	1.9
1-16	0.3

Figure 58: Matrix Fill Times: OMP vs. No OMP for N=168K, f=1.7998GHz

GEMINI Solver v3.0, incorporating hybrid MPI/Simple OMP parallelization, is executed on the DLX cluster for the N = 208 K, f = 2.0000 GHz test case. Table 42 shows the average matrix fill time, factor time, and required memory per node for three trials of each configuration: #DLX Nodes **:** #MPI processes/node **:** #OMP threads.

Table 42: GEMINI Solver v3.0 Matrix Fill & Factor Times for N=208K, f=2.0000GHz

DLX Nodes	MPIs/node	OMP Threads	# Cores	Matrix Fill Time (hr)	Matrix Factor Time (hr)	Required Memory/node
16	1*	16	256	–	–	–
16	2	8	256	2.121	6.194	75.9%
16	4	4	256	1.967	5.831	78.8%
16	8	2	256	2.004	2.909	84.9%
16	16	1	256	1.447	1.643	96.9%
20	1*	16	320	–	–	–
20	2	8	320	1.796	9.463	63.6%
20	4	4	320	1.507	4.667	66.1%
20	8	2	320	1.684	2.335	72.1%
20	16	1	320	1.060	1.204	84.5%
24	1*	16	384	–	–	–
24	2	8	384	1.463	7.865	55.1%
24	4	4	384	1.301	3.930	58.1%
24	8	2	384	1.362	1.976	64.0%
24	16	1	384	0.912	1.002	75.8%

***The total MPI processes required for 208 K unknowns was not attainable with 1 MPI/node**

Matrix factor times follow earlier trends for MPI only parallelization. Graphs of matrix fill times and required memory/node are shown in Figure 59 and Figure 60, respectively.

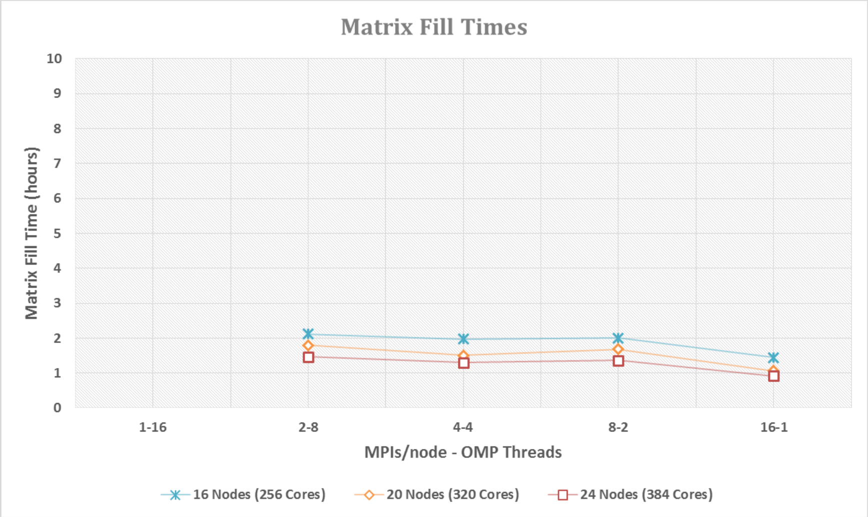


Figure 59: Matrix fill times vs. MPI-OMP combination for N=208K, f=2.0000GHz

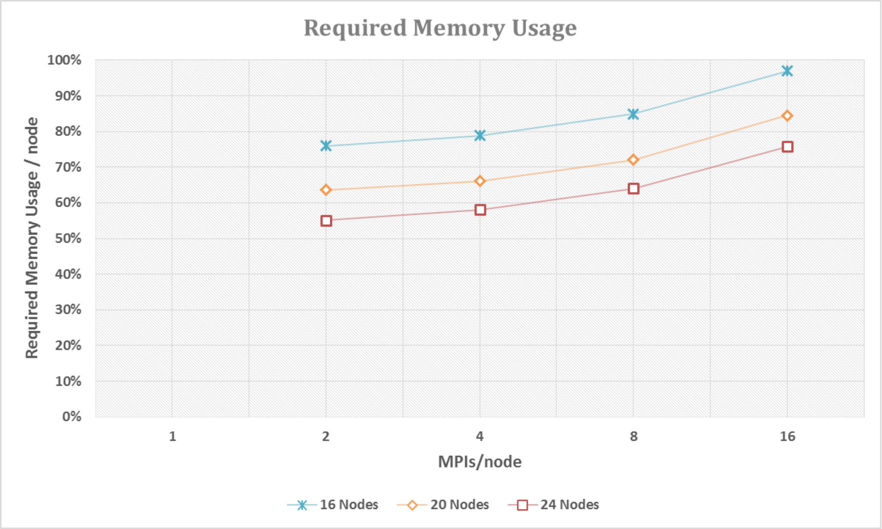


Figure 60: Required memory/node vs. MPIs/node for N=208K, f=2.0000GHz

As the number of MPI processes per node is reduced by powers of two, OMP threads are increased by the same to compensate and share more of the computational workload. As expected, each line graph in Figure 59 is relatively constant for all MPI-OMP combinations utilized. However, the required MPI processes for 208 K unknowns were not attainable with the 1-16 combination. At least 2 MPI processes per node must execute. In addition, as the number of MPI processes per node is increased, the required physical memory usage per node should increase as more copies of the mesh are needed. As expected, each line graph in Figure 60 increases as the number of MPI processes increases. When reducing the number of MPI processes, the matrix fill routine incorporates simple OMP threading to utilize idle cores and increase matrix fill performance. Matrix fill time, speedup, and efficiency graphs comparing OMP threading to the same cases without OMP threading are shown in are shown in Figure 61 on the next page. Matrix fill time comparing OMP threading to the same cases without OMP threading along with SEP tables are shown in Figure 62 on the page following next. The highest, most efficient speedup occurs for 2 MPI processes and 8 OMP threads with SEP = 2.2, 2.0, and 1.9 for 16, 20, and 24 nodes, respectively. The 2-8 combination requires the least MPI processes per node for effective OMP threading, has the highest, most efficient speedup, and uses the least physical memory per node.

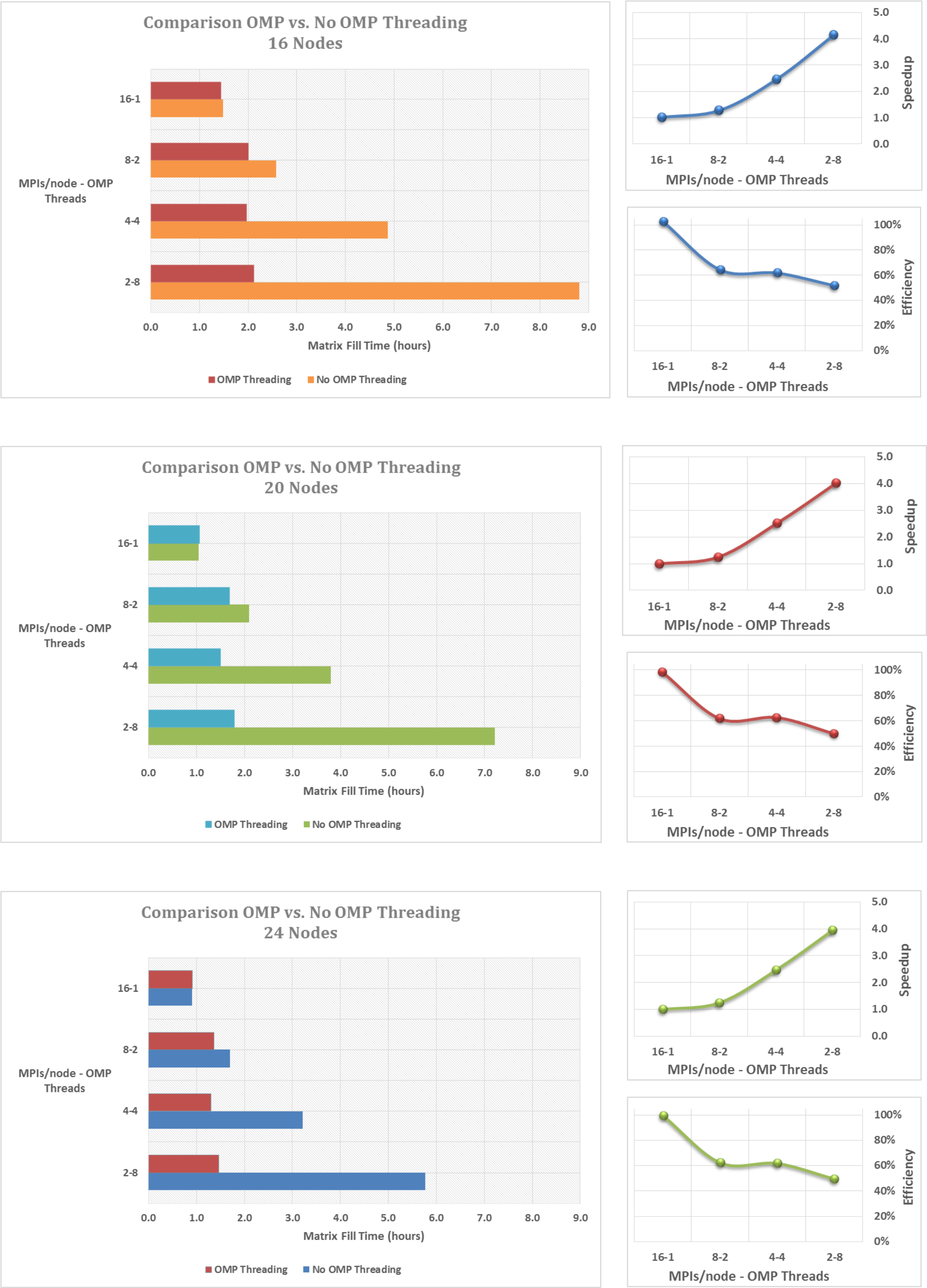
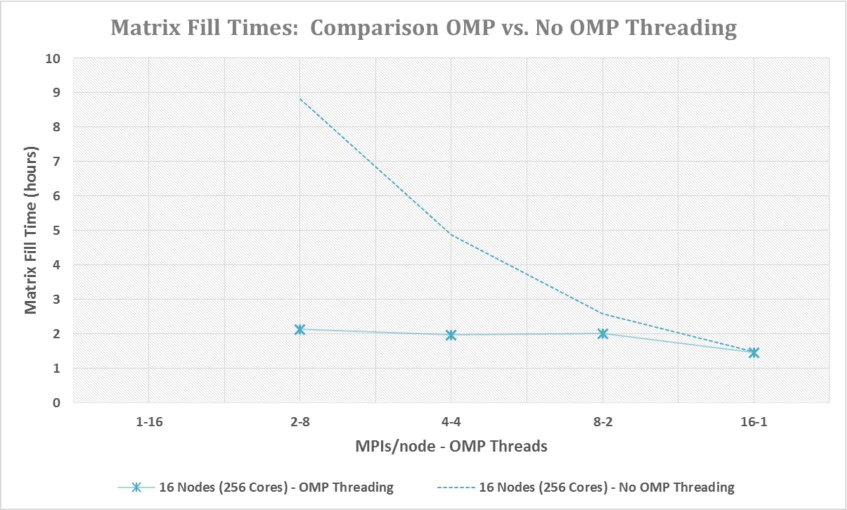
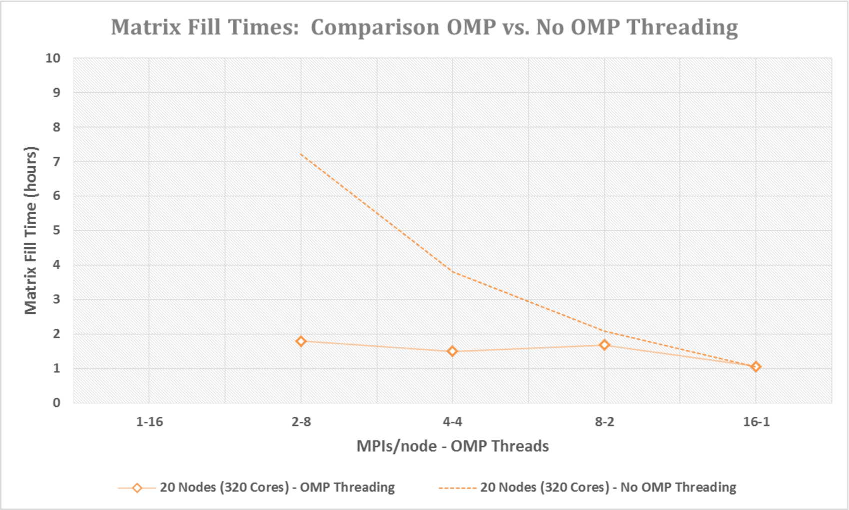


Figure 61: OMP threading vs. No OMP threading for N=208K, f=2.0000GHz



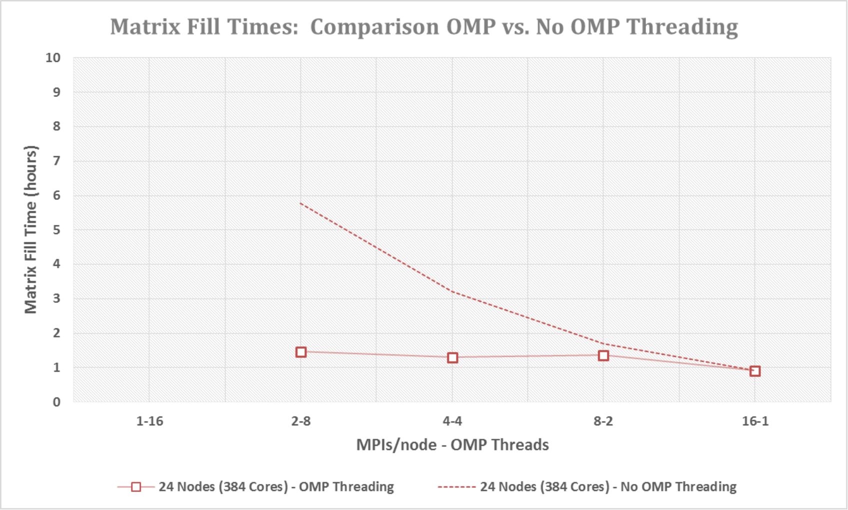
16 Nodes

MPI-OMP	SEP
16-1	1.0
8-2	0.8
4-4	1.5
2-8	2.2



20 Nodes

MPI-OMP	SEP
16-1	1.0
8-2	0.8
4-4	1.6
2-8	2.0



24 Nodes

MPI-OMP	SEP
16-1	1.0
8-2	0.8
4-4	1.5
2-8	1.9

Figure 62: Matrix Fill Times: OMP vs. No OMP for N=208K, f=2.0000GHz

GEMINI Solver v3.0, incorporating hybrid MPI/Simple OMP parallelization, is executed on the DLX cluster for the N = 299 K, f = 2.3984 GHz test case. Table 43 shows the average matrix fill time, factor time, and required memory per node for three trials of each configuration: #DLX Nodes **:** #MPI processes/node **:** #OMP threads.

Table 43: GEMINI Solver v3.0 Matrix Fill & Factor Times for N=299K, f=2.3984GHz

DLX Nodes	MPIs/node	OMP Threads	# Cores	Matrix Fill Time (hr)	Matrix Factor Time (hr)	Required Memory/node
28	1*	16	448	–	–	–
28	2	8	448	3.072	19.794	88.8%
28	4	4	448	2.502	9.819	93.0%
28	8	2	448	2.627	6.120	99.6%
28	16†	1	448	–	–	–
30	1*	16	480	–	–	–
30	2	8	480	2.861	18.503	83.0%
30	4	4	480	2.276	9.163	87.2%
30	8	2	480	2.681	5.049	95.6%
30	16†	1	480	–	–	–
32	1*	16	512	–	–	–
32	2	8	512	2.788	18.981	79.3%
32	4	4	512	2.179	8.601	84.4%
32	8	2	512	2.611	4.435	91.7%
32	16†	1	512	–	–	–

***The total MPI processes required for 299 K unknowns was not attainable with 1 MPI/node**
†The total memory required per node for 299 K unknowns was exceeded for 16 MPIs/node

Matrix factor times follow earlier trends for MPI only parallelization. Graphs of matrix fill times and required memory/node are shown in Figure 63 and Figure 64.

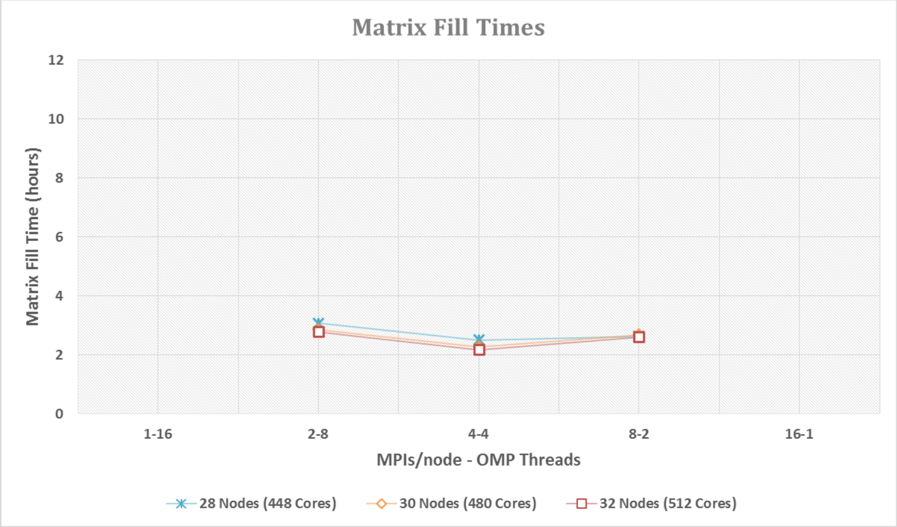


Figure 63: Matrix fill times vs. MPI-OMP combination for N=299K, f=2.3984GHz

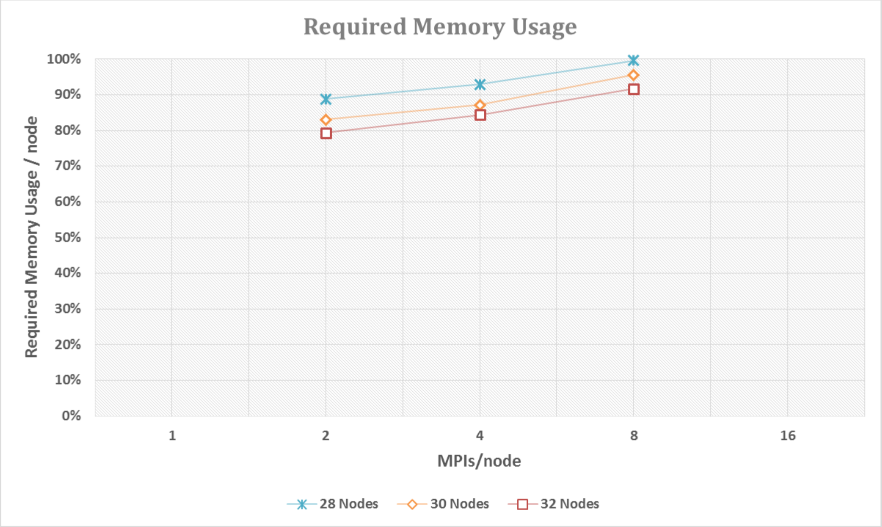


Figure 64: Required memory/node vs. MPIs/node for N=299K, f=2.3984GHz

As the number of MPI processes per node is reduced by powers of two, OMP threads are increased by the same to compensate and share more of the computational workload. As expected, each line graph in Figure 63 is relatively constant for all MPI-OMP combinations utilized. However, the required MPI processes for 299 K unknowns were not attainable with the 1-16 combination and the total memory required per node for 299 K unknowns was exceeded for the 16-1 combination. In this case, at least 2 MPI processes but no more than 8 MPI processes can execute on a node. In addition, as the number of MPI processes per node is increased, the required physical memory usage per node should increase as more copies of the mesh are needed. As expected, each line graph in Figure 64 increases as the number of MPI processes increases. When reducing the number of MPI processes, the matrix fill routine incorporates simple OMP threading to utilize idle cores and increase matrix fill performance. Matrix fill time, speedup, and efficiency graphs comparing OMP threading to the same cases without OMP threading are shown in Figure 65 on the next page. Matrix fill time comparing OMP threading to the same cases without OMP threading along with SEP tables are shown in Figure 66 on the page following next. The highest, most efficient speedup occurs for 2 MPI processes and 8 OMP threads with SEP = 1.5 for 32 nodes. GEMINI Solver would not execute with less than 32 nodes in single threading mode (no OMP threading). The 2-8 combination requires the least MPI processes per node for effective OMP threading, has the highest, most efficient speedup, and uses the least physical memory per node.

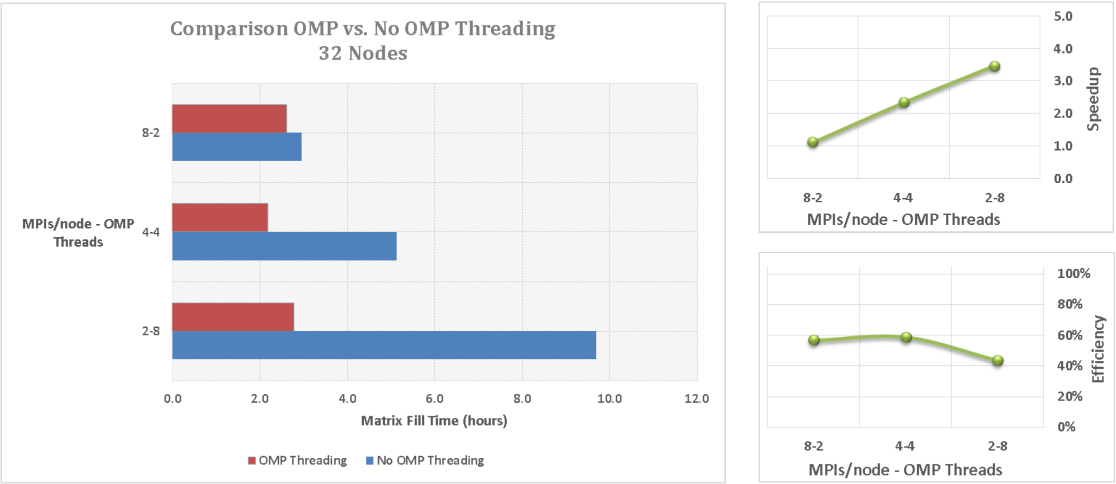
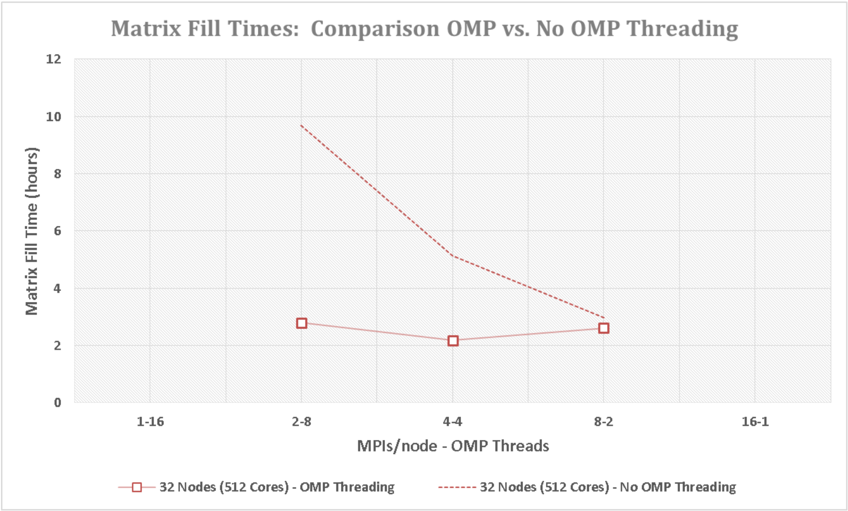


Figure 65: OMP threading vs. No OMP threading for N=299K, f=2.3984GHz



32 Nodes

MPI-OMP	SEP
8-2	0.6
4-4	1.4
2-8	1.5

Figure 66: Matrix Fill Times: OMP vs. No OMP for N=299K, f=2.3984GHz

GEMINI Solver v3.0, incorporating hybrid MPI/Simple OMP parallelization, is executed on the DLX cluster for the N = 326 K, f = 2.5000 GHz test case. Table 44 shows the average matrix fill time, factor time, and required memory per node for three trials of each configuration: #DLX Nodes **:** #MPI processes/node **:** #OMP threads.

Table 44: GEMINI Solver v3.0 Matrix Fill & Factor Times for N=326K, f=2.5000GHz

DLX Nodes	MPIs/node	OMP Threads	# Cores	Matrix Fill Time (hr)	Matrix Factor Time (hr)	Required Memory/node
30	1*	16	480	–	–	–
30	2	8	480	3.440	25.070	97.5%
30	4	4	480	2.851	13.115	99.6%
30	8†	2	480	–	–	–
30	16†	1	480	–	–	–
32	1*	16	512	–	–	–
32	2	8	512	3.370	22.700	92.1%
32	4	4	512	2.592	11.503	97.4%
32	8†	2	512	–	–	–
32	16†	1	512	–	–	–

***The total MPI processes required for 326 K unknowns was not attainable with 1 MPI/node**
†The total memory required per node for 326 K unknowns was exceeded for 8 and 16 MPIs/node

Matrix factor times agree with earlier values for MPI only parallelization. Graphs of matrix fill times and required memory/node are shown in Figure 67. For the maximum allowable 32 nodes on the DLX cluster, the total memory required per node is >92%.

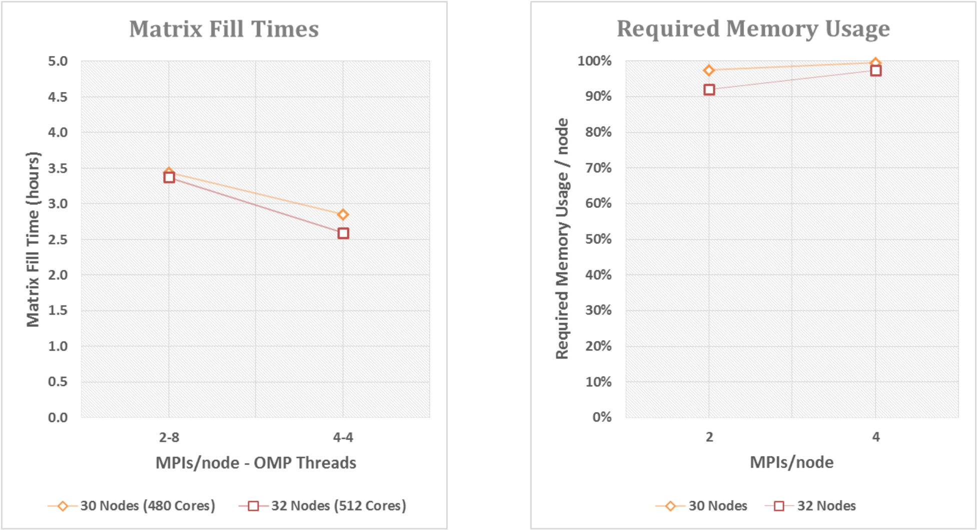


Figure 67: Matrix fill times & memory usage for N=326K, f=2.5000GHz

GEMINI Solver v3.0, incorporating hybrid MPI/Simple OMP parallelization, is executed on the DLX cluster for the N = 342 K, f = 2.5624 GHz test case. Table 45 shows the average matrix fill time, factor time, and required memory per node for three trials of each configuration: #DLX Nodes **:** #MPI processes/node **:** #OMP threads.

Table 45: GEMINI Solver v3.0 Matrix Fill & Factor Times for N=342K, f=2.5624GHz

DLX Nodes	MPIs/node	OMP Threads	# Cores	Matrix Fill Time (hr)	Matrix Factor Time (hr)	Required Memory/node
32	1*	16	512	–	–	–
32	2	8	512	4.371	31.489	99.5%
32	4†	4	512	–	–	–
32	8†	2	512	–	–	–
32	16†	1	512	–	–	–

***The total MPI processes required for 342 K unknowns was not attainable with 1 MPI/node**

†The total memory required per node for 342 K unknowns was exceeded for 4, 8, and 16 MPIs/node

Matrix factor times agree with earlier values for MPI only parallelization. Graphs of matrix fill times and required memory/node are shown in Figure 68. For the maximum allowable 32 nodes on the DLX cluster, the total memory required per node is ≈100%.

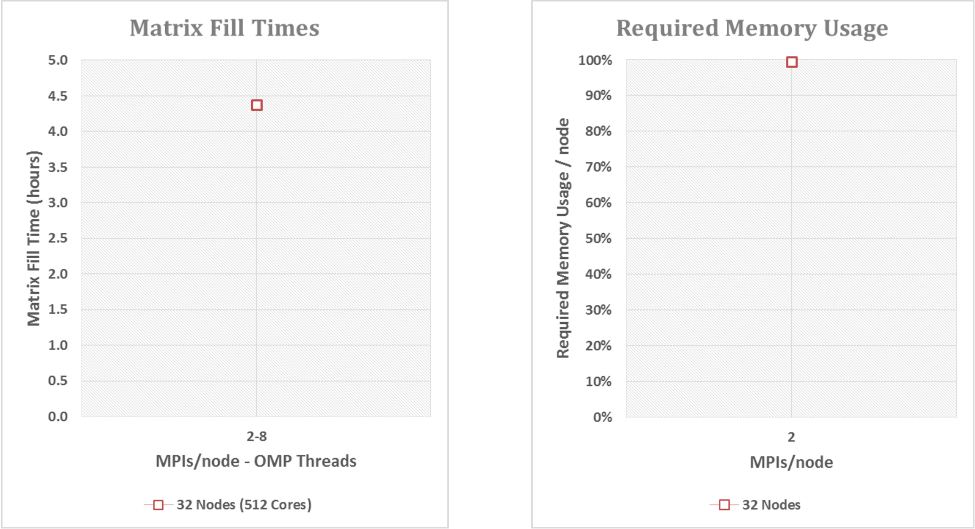


Figure 68: Matrix fill times & memory usage for N=342K, f=2.5624GHz

Test cases with N > 342 K unknowns will not run on the DLX cluster because the total memory required per node exceeds 100%.s

5.3. GEMINI Post Results

Comparisons of the RCS pattern generated by GEMINI Post are made to the Mie Series using the Chi-Square goodness-of-fit (GOF) [15] given by equation (25). Table 46 displays the goodness-of-fit (GOF) between GEMINI Post RCS values and the Mie Series using N_{max} terms. The bistatic angle resolution is given by Δangle.

Table 46: GEMINI Post RCS results fit to Mie Series

Frequency, f (GHz)	Edges, N (# unknowns)	Solution Test	Quality of Fit	N_{max} Terms	Δangle (°)	σ_{VV} χ^2 GOF	σ_{HH} χ^2 GOF
0.8994	41,415	EFIE	Excellent	19	1	0.013	0.015
1.1992	74,211	EFIE	Excellent	25	1	0.018	0.019
1.7988	167,652	EFIE	Excellent	38	1	0.026	0.025
2.0000	207,663	EFIE	Excellent	42	0.5	0.055	0.054
2.3984	298,863	EFIE	Excellent	51	0.5	0.061	0.060
2.5000	326,430	EFIE	Excellent	53	0.5	0.063	0.059
2.5624	342,087	EFIE	Excellent	54	0.5	0.060	0.056

A plot of N_{max} vs. N for the Goodness-of-Fit results is shown in Figure 69. The curve follows the trend $N_{max} \sim \sqrt{N}$.

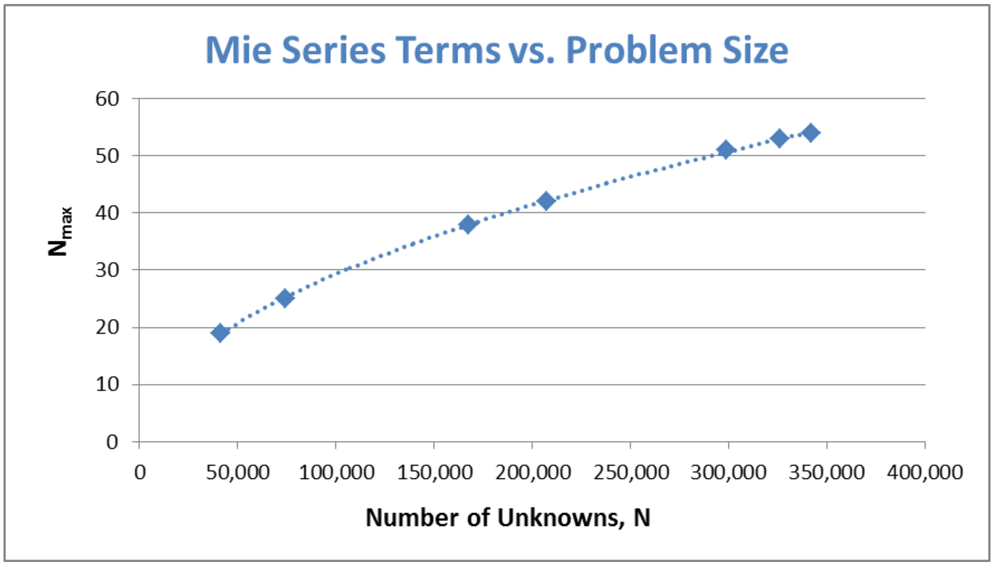


Figure 69: N_{max} vs. Number of Unknowns

Appendix A holds the plots and goodness-of-fit statistics for all cases listed in Table 46.

Chapter 6. Conclusion & Future Direction

6.1. Conclusion

The highest, most efficient speedup for the test cases listed in Table 36 occurs for the combination of 2 MPI processes and 8 OMP threads. Table 47 shows the SEP results for all 2-8 MPI-OMP combinations for which comparison runs could be performed.

Table 47: Results for best MPI-OMP combinations of matrix fill routine

Frequency, f (GHz)	Edges, N (# unknowns)	Solution Test	Best MPI – OMP Combination	# DLX Nodes	Average SEP
0.8994	41,415	EFIE	2-8	1,2,4	1.8
1.1992	74,211	EFIE	2-8	4,8,12	1.8
1.7988	167,652	EFIE	2-8	16,20,24	1.9
2.0000	207,663	EFIE	2-8	16,20,24	2.0
2.3984	298,863	EFIE	2-8	32	1.5

The 2-8 combination affords the following advantages:

- minimum MPI processes needed per node for effective OMP threading;
- highest, most efficient speedup;
- least physical memory usage per node.

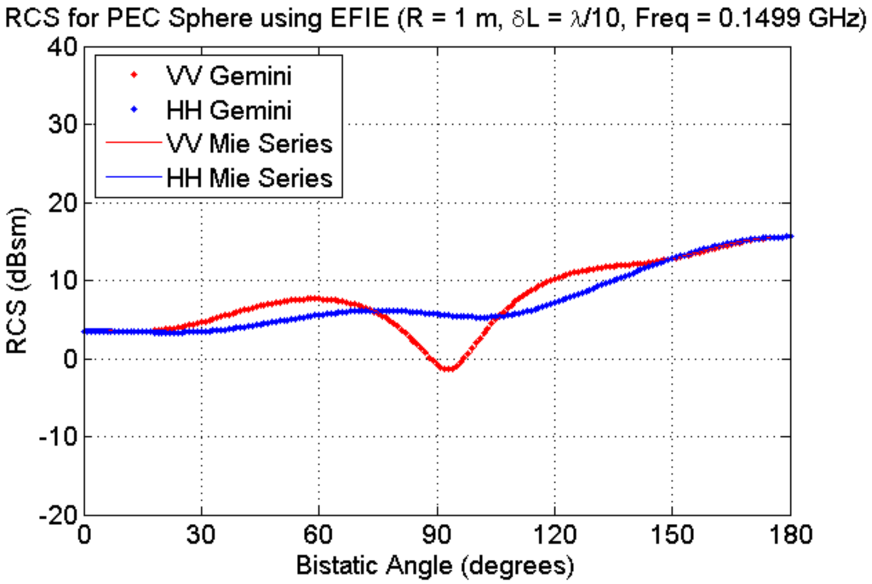
The largest problem size of 342 K unknowns could only be executed on 32 nodes with the 2-8 combination. DLX policy allows a maximum of 32 nodes per user. For problem sizes beyond 342 K, more than 32 nodes will be needed.

6.2. Future Direction

To achieve the highest, most efficient speedup in the GEMINI Solver matrix fill routine, utilize hybrid MPI-OMP parallelization with 2 MPI processes and 8 OMP threads per node. In addition, explore the use of high power clusters with more than 16 cores per node and test different MPI-OMP combinations to find the highest, most efficient speedup combinations. Furthermore, solve the OMP race condition requiring modifications #25 and #26 listed in Appendix B without using an OMP CRITICAL directive. Finally, explore the utilization of hybrid MPI-OMP parallelization within the GEMINI Solver matrix solver routine to supplement the current integration of the University of Kentucky MFD library of advanced solution methods.

Appendix A GEMINI Post RCS Results Fit to Mie Series

Figure 70 shows VV (φ = 0°, 0° ≤ θ ≤ 180°) and HH (ϕ = 90°, 0° ≤ θ ≤ 180°) plots of GEMINI Post EFIE RCS results (dB referenced to 1 m^2) for f = 0.1499 GHz with fits to the Mie Series (dB referenced to 1 m^2). Chi-Square (χ^2) goodness-of-fit (GOF) statistics indicate Gemini results are in excellent agreement with the Mie Series. VV and HH linear scatterplots show a nearly perfect correlation between Gemini RCS values, y_i, and Mie Series values, y_i*.



Frequency (GHz)
0.1499
Unknowns
1,083
Solution
EFIE
Nmax Terms
4
Agreement
Excellent
σ_{VV} χ^2 GOF
0.103
σ_{HH} χ^2 GOF
0.080

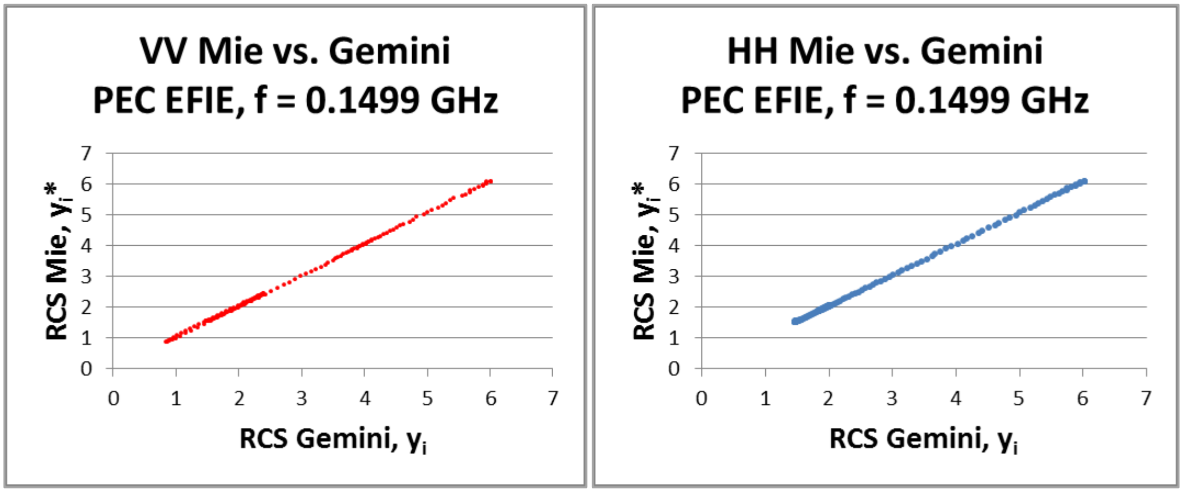
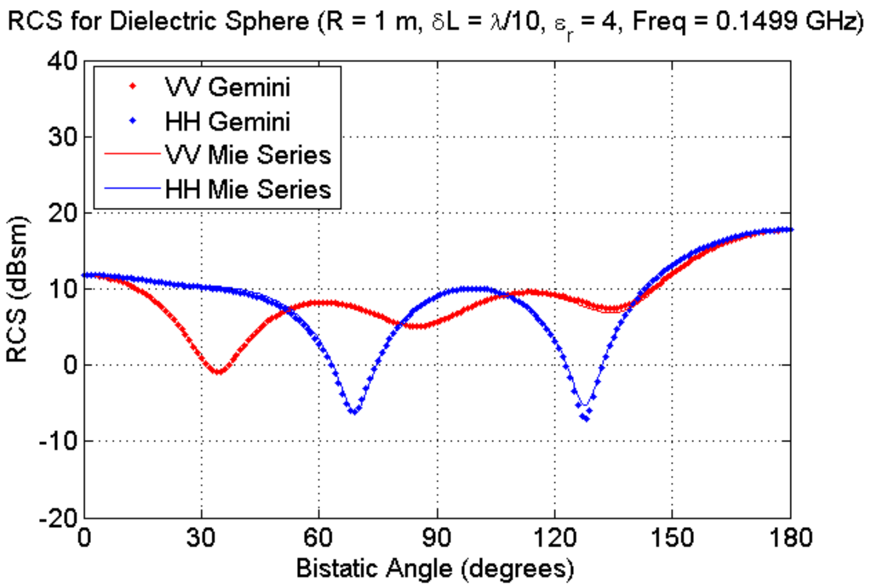


Figure 70: (Upper-Left) Gemini EFIE RCS results vs. Mie Series for f=0.1499 GHz
(Upper-Right) Chi-Square Goodness-of-Fit statistics
(Lower) Linear scatter plots of Mie Series vs. Gemini EFIE RCS result

Figure 71 shows VV ($\phi = 0°$, $0° \leq \theta \leq 180°$) and HH ($\phi = 90°$, $0° \leq \theta \leq 180°$) plots of GEMINI Post Dielectric RCS results (dB referenced to 1 m^2) for f = 0.1499 GHz with fits to the Mie Series (dB referenced to 1 m^2). Chi-Square (χ^2) goodness-of-fit (GOF) statistics indicate Gemini results are in good agreement with the Mie Series. VV and HH linear scatterplots show a good correlation between Gemini RCS values, y_i, and Mie Series values, y_i*. However, a minor lack of fit is observed by the scatter in the linear values up to ≈ 4 m^2.



Frequency (GHz)
0.1499
Unknowns
2,166
Solution
DIELECTRIC
Nmax Terms
4
Agreement
Good
σ_{VV} χ^2 GOF
0..475
σ_{HH} χ^2 GOF
0.351

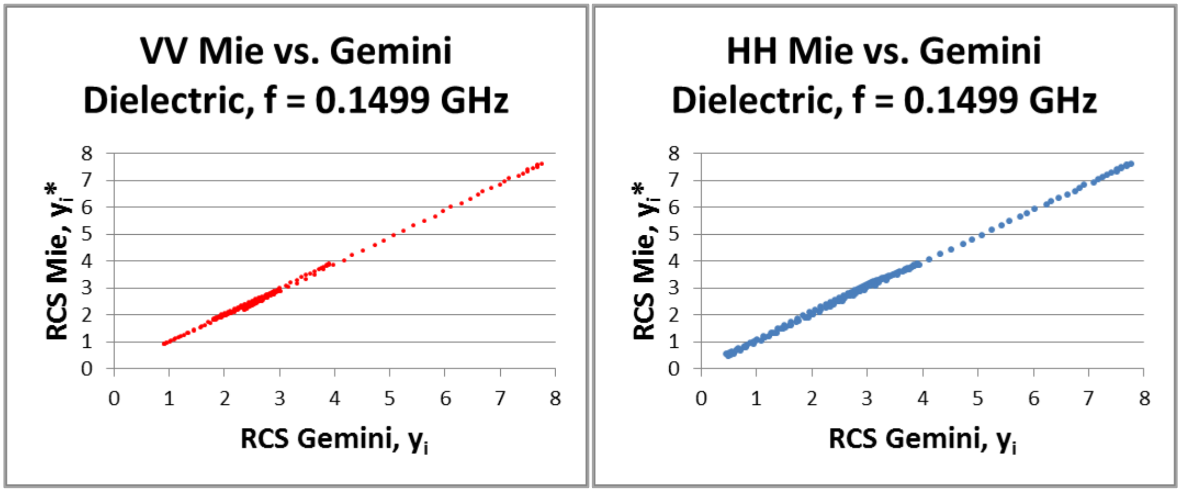
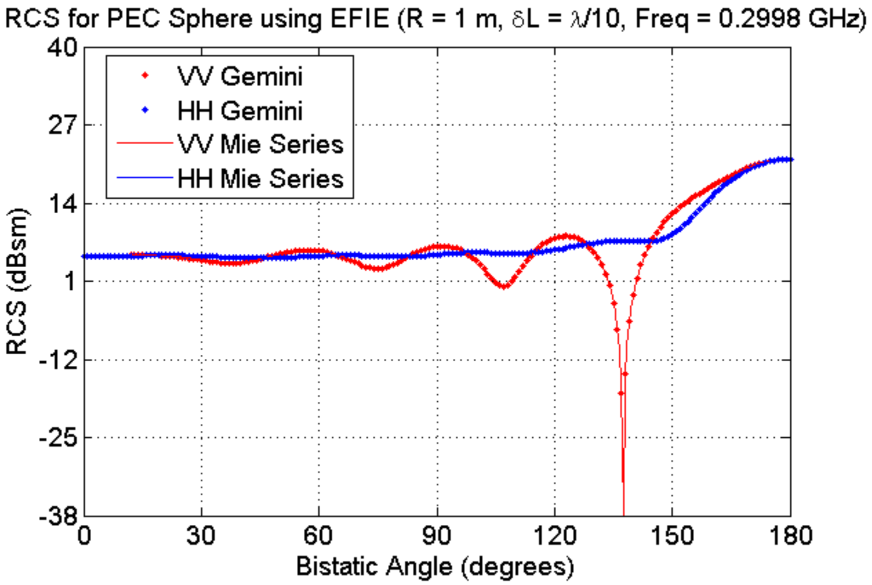


Figure 71: (Upper-Left) Gemini Dielectric RCS results vs. Mie Series for f=0.1499 GHz
(Upper-Right) Chi-Square Goodness-of-Fit statistics
(Lower) Linear scatter plots of Mie Series vs. Gemini Dielectric RCS results

Figure 72 shows VV (ϕ = 0°, 0° ≤ θ ≤ 180°) and HH (ϕ = 90°, 0° ≤ θ ≤ 180°) plots of GEMINI Post EFIE RCS results (dB referenced to 1 m^2) for f = 0.2998 GHz with fits to the Mie Series (dB referenced to 1 m^2). Chi-Square (χ^2) goodness-of-fit (GOF) statistics indicate Gemini results are in excellent agreement with the Mie Series. VV and HH linear scatterplots show a nearly perfect correlation between Gemini RCS values, y_i, and Mie Series values, y_i*.



Frequency (GHz)	0.2998
Unknowns	4,455
Solution	EFIE
Nmax Terms	7
Agreement	Excellent
σ_{VV} χ^2 GOF	0.008
σ_{HH} χ^2 GOF	0.007

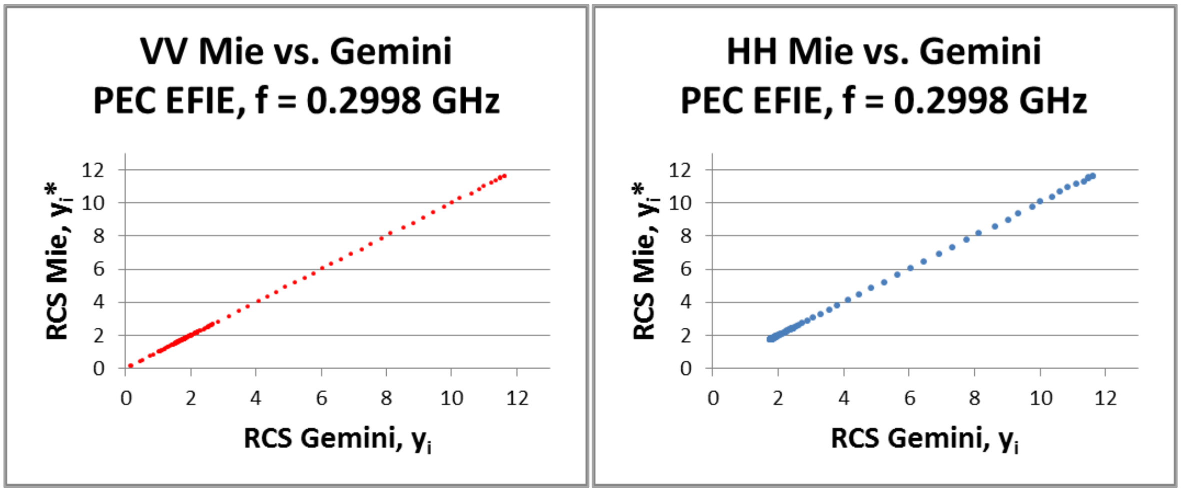
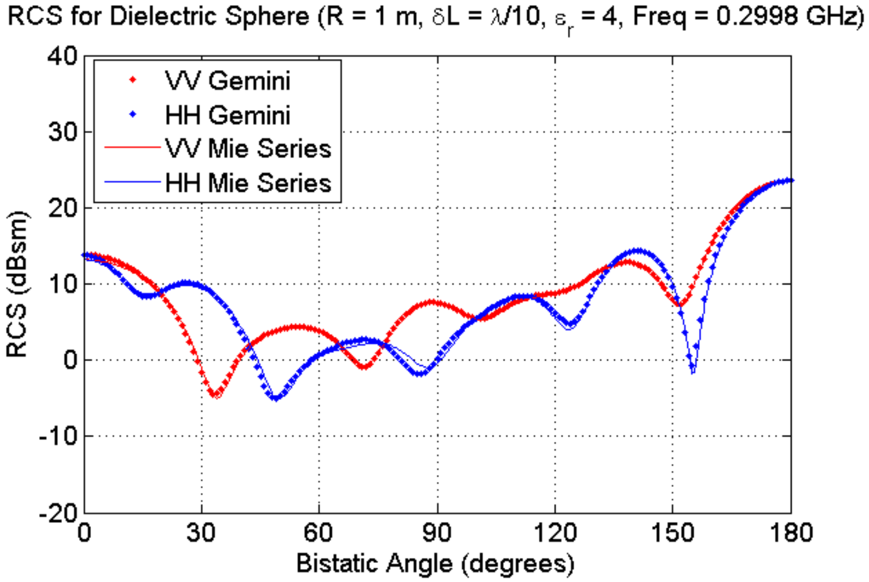


Figure 72: (Upper-Left) Gemini EFIE RCS results vs. Mie Series for f=0.2998 GHz
(Upper-Right) Chi-Square Goodness-of-Fit statistics
(Lower) Linear scatter plots of Mie Series vs. Gemini EFIE RCS results

Figure 73 shows VV (ϕ = 0°, 0° ≤ θ ≤ 180°) and HH (ϕ = 90°, 0° ≤ θ ≤ 180°) plots of GEMINI Post Dielectric RCS results (dB referenced to 1 m^2) for f = 0.2998 GHz with fits to the Mie Series (dB referenced to 1 m^2). Chi-Square (χ^2) goodness-of-fit (GOF) statistics indicate Gemini results are in good agreement with the Mie Series. VV and HH linear scatterplots show a good correlation between Gemini RCS values, y_i, and Mie Series values, y_i*. However, a slight lack of fit is observed by the scatter in the linear values up to ≈ 5 m^2.



Frequency (GHz)
0.2998
Unknowns
8,910
Solution
DIELECTRIC
Nmax Terms
7
Agreement
Good
σ_{VV} χ^2 GOF
0.204
σ_{HH} χ^2 GOF
0.321

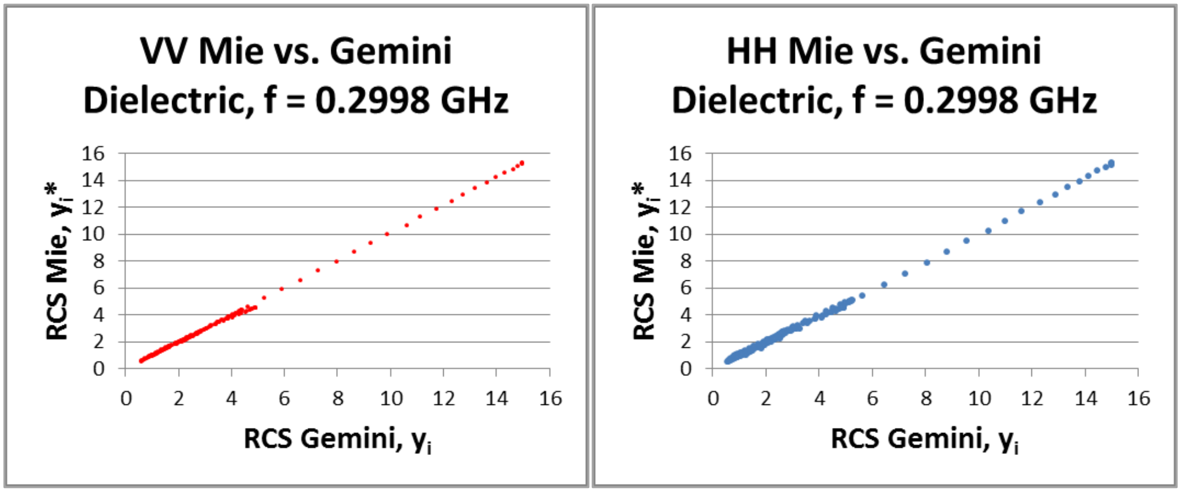
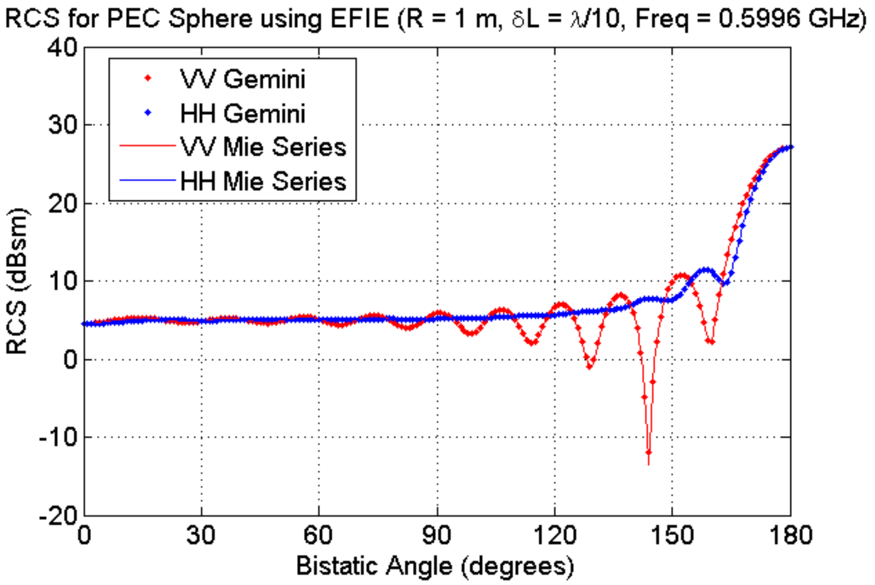


Figure 73: (Upper-Left) Gemini Dielectric RCS results vs. Mie Series for f=0.2998 GHz (Upper-Right) Chi-Square Goodness-of-Fit statistics (Lower) Linear scatter plots of Mie Series vs. Gemini Dielectric RCS results

Figure 74 shows VV ($\phi = 0°$, $0° \leq \theta \leq 180°$) and HH ($\phi = 90°$, $0° \leq \theta \leq 180°$) plots of GEMINI Post EFIE RCS results (dB referenced to 1 m^2) for f = 0.5996 GHz with fits to the Mie Series (dB referenced to 1 m^2). Chi-Square (χ^2) goodness-of-fit (GOF) statistics indicate Gemini results are in excellent agreement with the Mie Series. VV and HH linear scatterplots show a nearly perfect correlation between Gemini RCS values, y_i, and Mie Series values, y_i*.



Frequency (GHz)	Unknowns	Solution	Nmax Terms	Agreement	σ_{VV} χ^2 GOF	σ_{HH} χ^2 GOF
0.5996	18,162	EFIE	13	Excellent	0.009	0.010

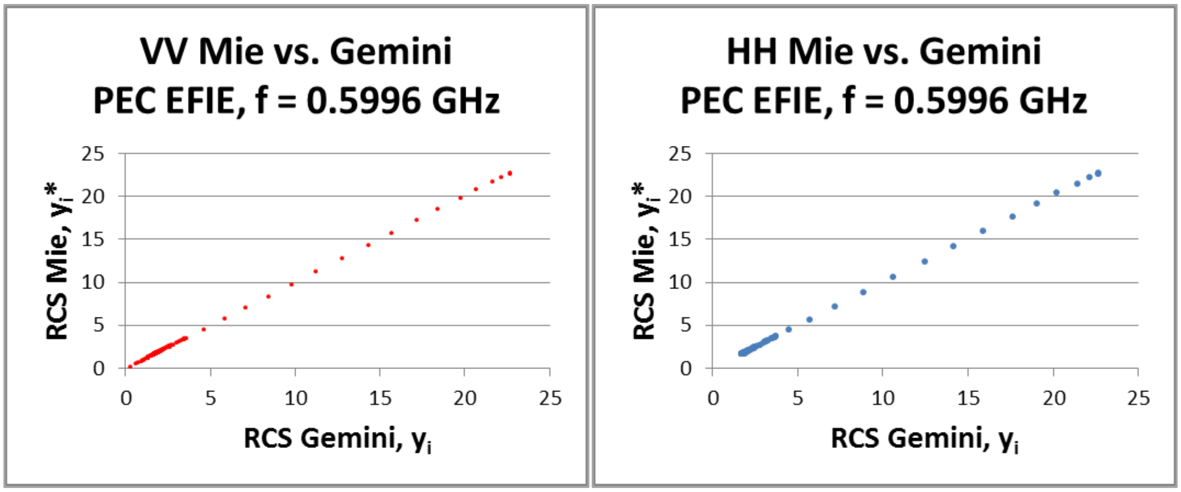
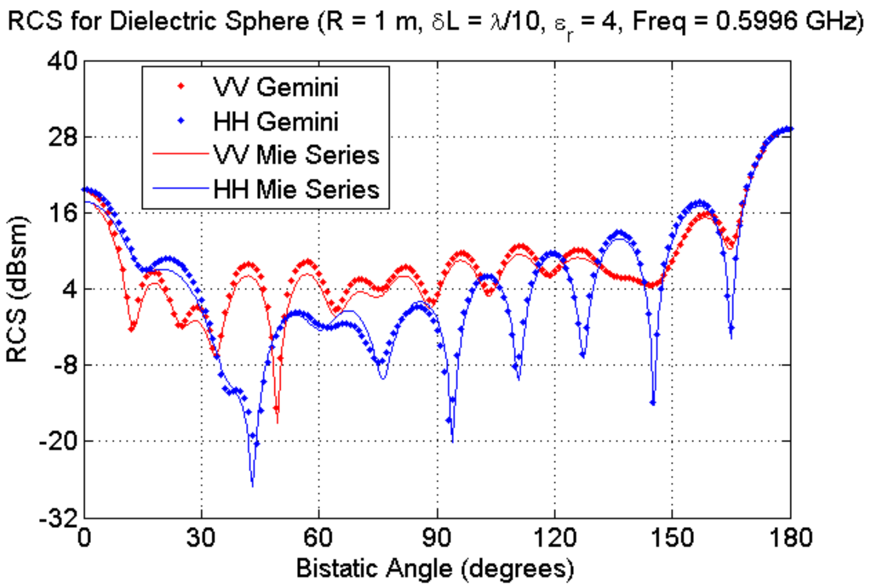


Figure 74: (Upper-Left) Gemini EFIE RCS results vs. Mie Series for f=0.5996 GHz
(Upper-Right) Chi-Square Goodness-of-Fit statistics
(Lower) Linear scatter plots of Mie Series vs. Gemini EFIE RCS results

Figure 75 shows VV (ϕ = 0°, 0° ≤ θ ≤ 180°) and HH (ϕ = 90°, 0° ≤ θ ≤ 180°) plots of GEMINI Post Dielectric RCS results (dB referenced to 1 m^2) for f = 0.5996 GHz with fits to the Mie Series (dB referenced to 1 m^2). Chi-Square (χ^2) goodness-of-fit (GOF) statistics indicate Gemini results are in fair agreement with the Mie Series. VV and HH linear scatterplots show a fair correlation between Gemini RCS values, y_i, and Mie Series values, y_i*. However, a clear lack of fit is observed by the scatter in the linear values up to ≈ 10 m^2.



Frequency (GHz)	0.5996
Unknowns	36,324
Solution	DIELECTRIC
Nmax Terms	13
Agreement	Fair
σ_{VV} χ^2 GOF	2.123
σ_{HH} χ^2 GOF	2.308

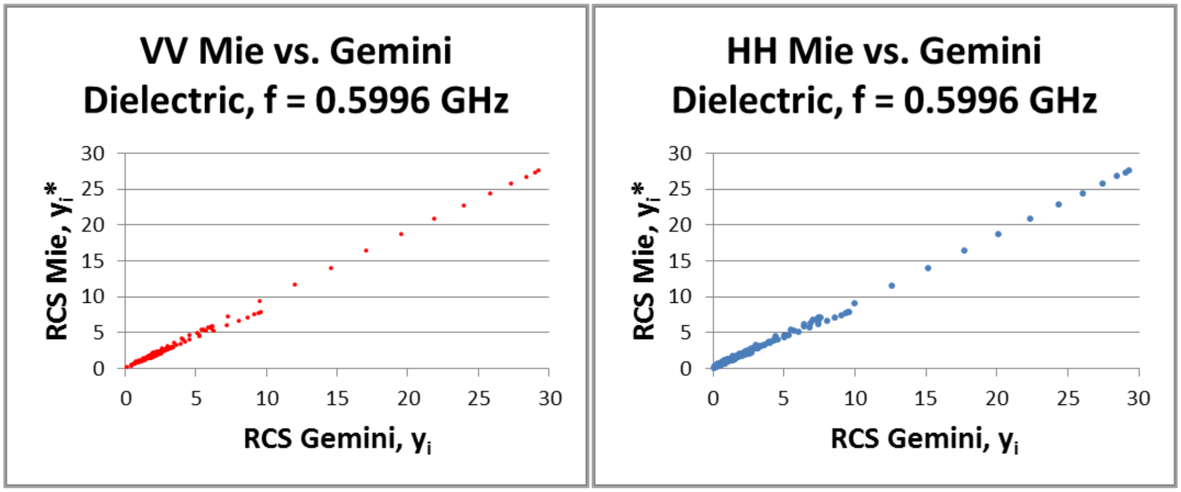
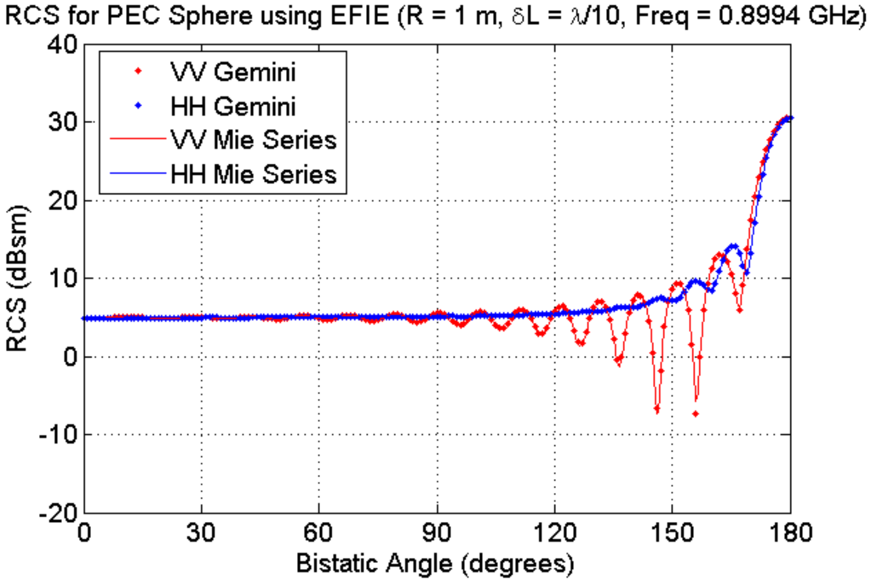


Figure 75: (Upper-Left) Gemini Dielectric RCS results vs. Mie Series for f=0.5996 GHz (Upper-Right) Chi-Square Goodness-of-Fit statistics (Lower) Linear scatter plots of Mie Series vs. Gemini Dielectric RCS results

Figure 76 shows VV (ϕ = 0°, 0° ≤ θ ≤ 180°) and HH (ϕ = 90°, 0° ≤ θ ≤ 180°) plots of GEMINI Post EFIE RCS results (dB referenced to 1 m^2) for f = 0.8994 GHz with fits to the Mie Series (dB referenced to 1 m^2). Chi-Square (χ^2) goodness-of-fit (GOF) statistics indicate Gemini results are in excellent agreement with the Mie Series. VV and HH linear scatterplots show a nearly perfect correlation between Gemini RCS values, y_i, and Mie Series values, y_i*.



Frequency (GHz)	0.8994
Unknowns	41,415
Solution	EFIE
Nmax Terms	19
Agreement	Excellent
σ_{VV} χ^2 GOF	0.013
σ_{HH} χ^2 GOF	0.015

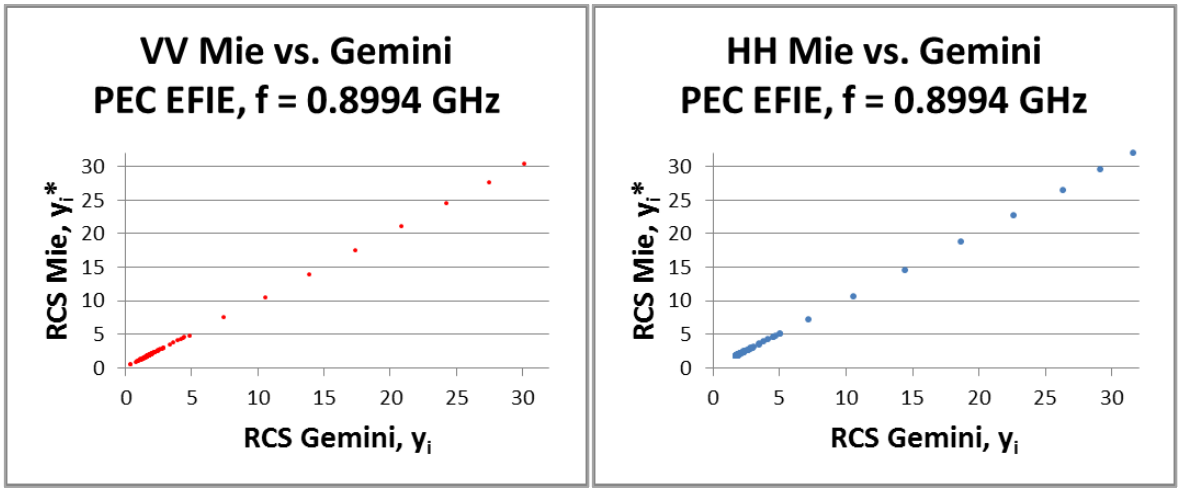
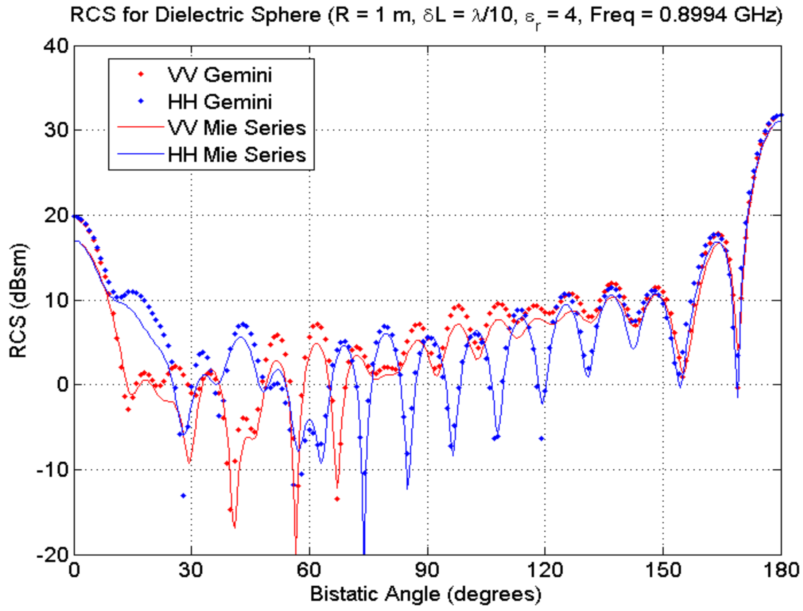


Figure 76: (Upper-Left) Gemini EFIE RCS results vs. Mie Series for f=0.8994 GHz
(Upper-Right) Chi-Square Goodness-of-Fit statistics
(Lower) Linear scatter plots of Mie Series vs. Gemini EFIE RCS results

Figure 77 shows VV ($\phi = 0°$, $0° \leq \theta \leq 180°$) and HH ($\phi = 90°$, $0° \leq \theta \leq 180°$) plots of GEMINI Post Dielectric RCS results (dB referenced to 1 m^2) for f = 0.8994 GHz with fits to the Mie Series (dB referenced to 1 m^2). Chi-Square (χ^2) goodness-of-fit (GOF) statistics indicate Gemini results are in poor agreement with the Mie Series. The Mie Series values tend to underestimate the Gemini values. VV and HH linear scatterplots show a significant lack of fit by the scatter observed in the linear values up to $\approx 10\ m^2$.



Frequency (GHz)	0.8994
Unknowns	82,830
Solution	DIELECTRIC
Nmax Terms	19
Agreement	Poor
σ_{VV} χ^2 GOF	4.115
σ_{HH} χ^2 GOF	4.604

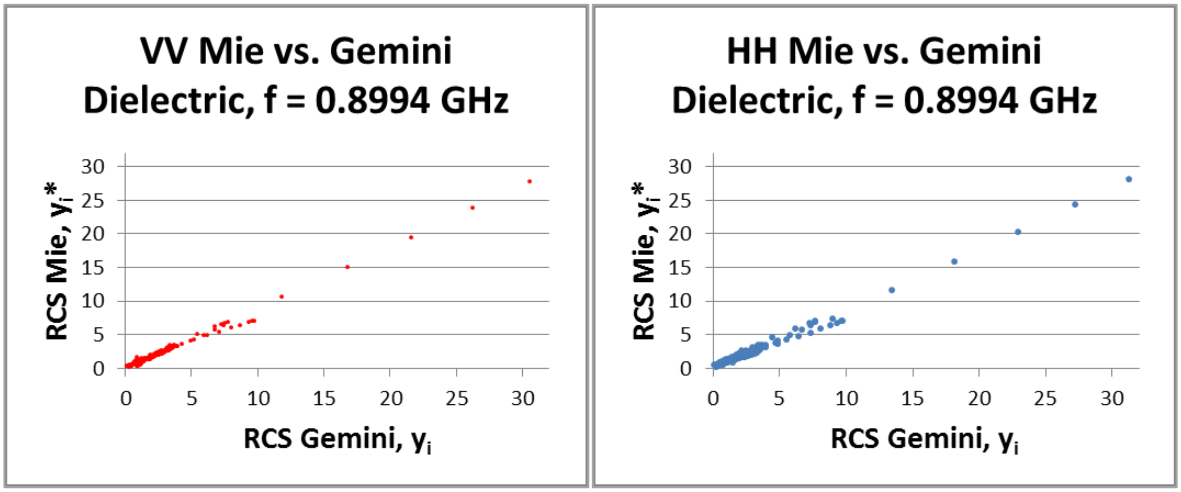
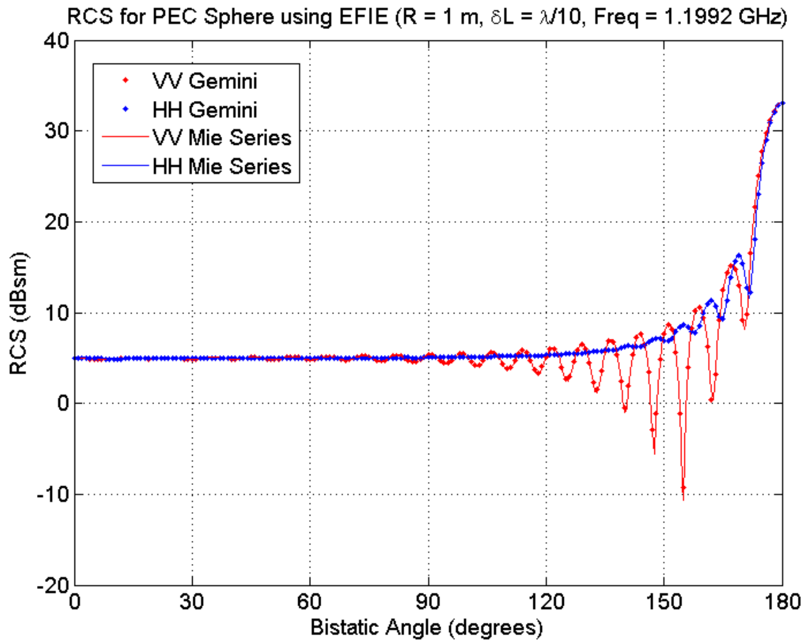


Figure 77: (Upper-Left) Gemini Dielectric RCS results vs. Mie Series for f=0.8994 GHz (Upper-Right) Chi-Square Goodness-of-Fit statistics (Lower) Linear scatter plots of Mie Series vs. Gemini Dielectric RCS results

Figure 78 shows VV ($\phi = 0°$, $0° \leq \theta \leq 180°$) and HH ($\phi = 90°$, $0° \leq \theta \leq 180°$) plots of GEMINI Post EFIE RCS results (dB referenced to 1 m^2) for f = 1.1992 GHz with fits to the Mie Series (dB referenced to 1 m^2). Chi-Square (χ^2) goodness-of-fit (GOF) statistics indicate Gemini results are in excellent agreement with the Mie Series. VV and HH linear scatterplots show a nearly perfect correlation between Gemini RCS values, y_i, and Mie Series values, y_i*.



Frequency (GHz)	1.1992
Unknowns	74,211
Solution	EFIE
Nmax Terms	25
Agreement	Excellent
σ_{VV} χ^2 GOF	0.018
σ_{HH} χ^2 GOF	0.019

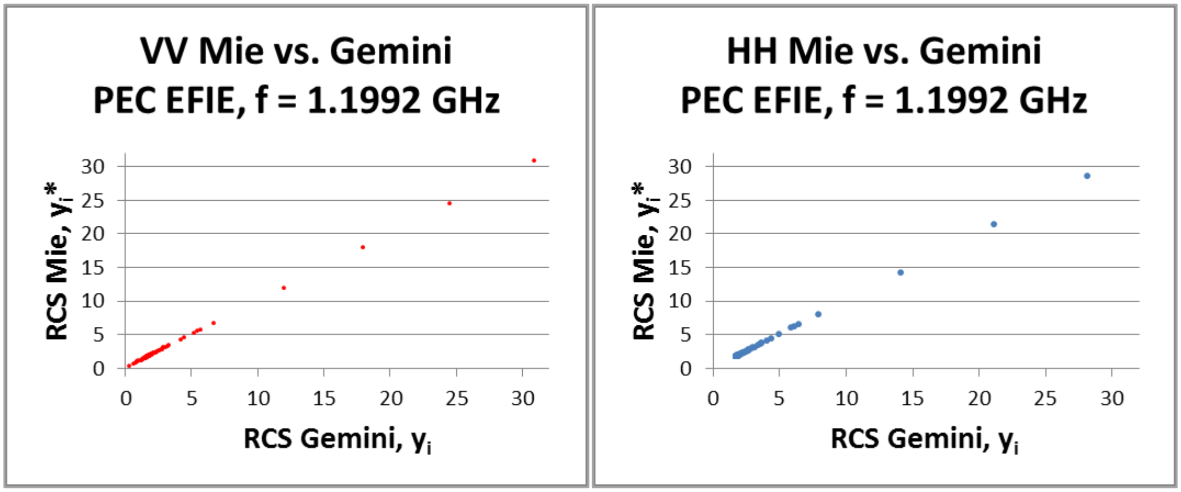
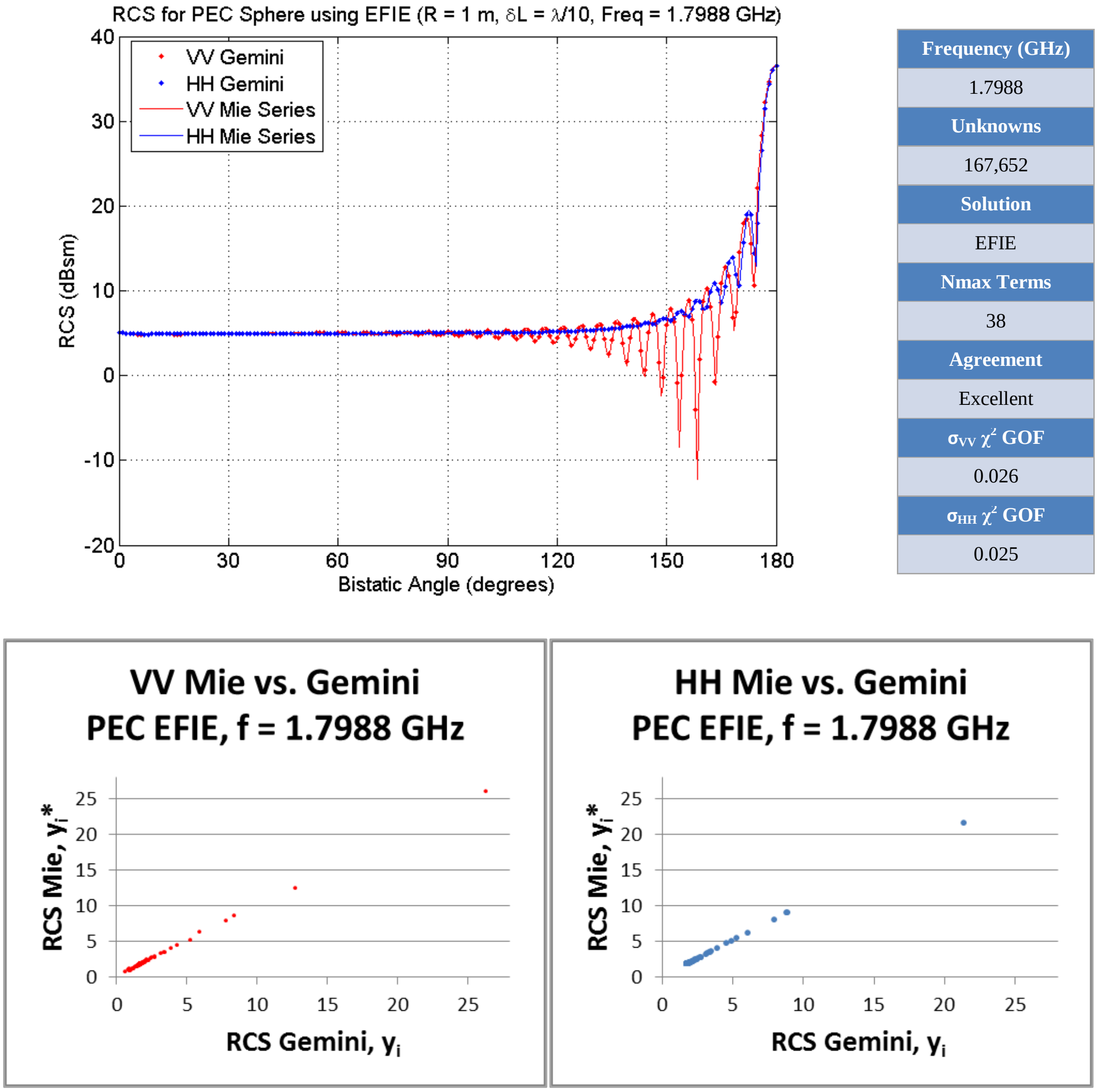


Figure 78: (Upper-Left) Gemini EFIE RCS results vs. Mie Series for f=1.1992 GHz (Upper-Right) Chi-Square Goodness-of-Fit statistics (Lower) Linear scatter plots of Mie Series vs. Gemini EFIE RCS results

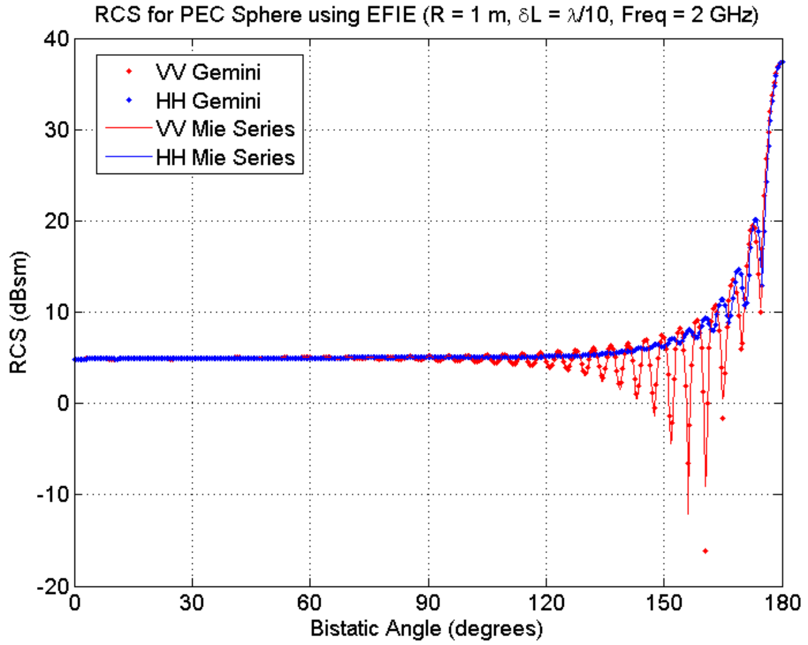
Figure 79 shows VV (ϕ = 0°, 0° ≤ θ ≤ 180°) and HH (ϕ = 90°, 0° ≤ θ ≤ 180°) plots of GEMINI Post EFIE RCS results (dB referenced to 1 m^2) for f = 1.7988 GHz with fits to the Mie Series (dB referenced to 1 m^2). Chi-Square (χ^2) goodness-of-fit (GOF) statistics indicate Gemini results are in excellent agreement with the Mie Series. VV and HH linear scatterplots show a nearly perfect correlation between Gemini RCS values, y_i, and Mie Series values, y_i*.

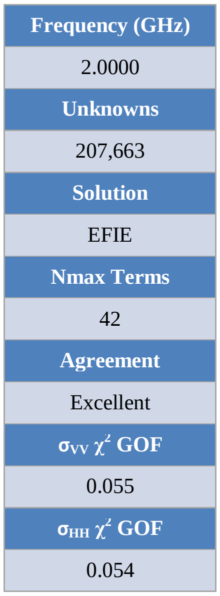


Frequency (GHz)
1.7988
Unknowns
167,652
Solution
EFIE
Nmax Terms
38
Agreement
Excellent
σ_{VV} χ^2 GOF
0.026
σ_{HH} χ^2 GOF
0.025

Figure 79: (Upper-Left) Gemini EFIE RCS results vs. Mie Series for f=1.7988 GHz (Upper-Right) Chi-Square Goodness-of-Fit statistics (Lower) Linear scatter plots of Mie Series vs. Gemini EFIE RCS results

Figure 80 shows VV ($\phi = 0°$, $0° \leq \theta \leq 180°$) and HH ($\phi = 90°$, $0° \leq \theta \leq 180°$) plots of GEMINI Post EFIE RCS results (dB referenced to 1 m^2) for f = 2.0000 GHz with fits to the Mie Series (dB referenced to 1 m^2). Chi-Square (χ^2) goodness-of-fit (GOF) statistics indicate Gemini results are in excellent agreement with the Mie Series. VV and HH linear scatterplots show a nearly perfect correlation between Gemini RCS values, y_i, and Mie Series values, y_i*.





Frequency (GHz)
2.0000
Unknowns
207,663
Solution
EFIE
Nmax Terms
42
Agreement
Excellent
σ_{VV} χ^2 GOF
0.055
σ_{HH} χ^2 GOF
0.054

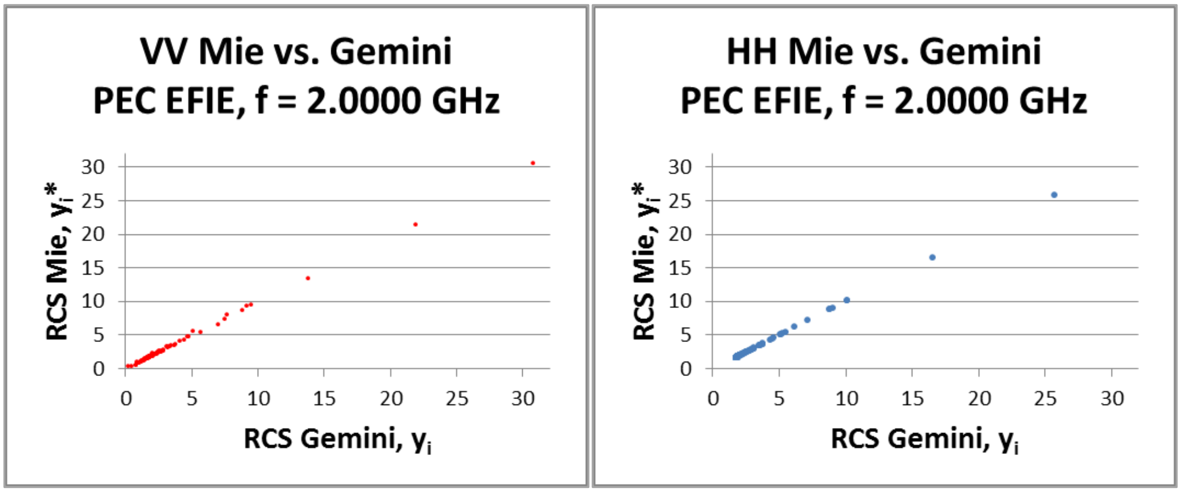
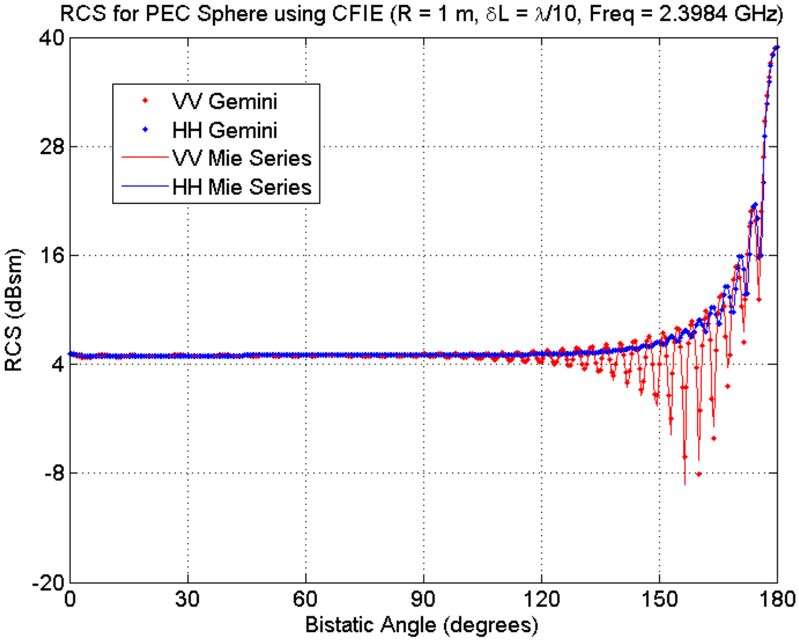


Figure 80: (Upper-Left) Gemini EFIE RCS results vs. Mie Series for f=2.0000 GHz (Upper-Right) Chi-Square Goodness-of-Fit statistics (Lower) Linear scatter plots of Mie Series vs. Gemini EFIE RCS results

Figure 81 shows VV (ϕ = 0°, 0° ≤ θ ≤ 180°) and HH (ϕ = 90°, 0° ≤ θ ≤ 180°) plots of GEMINI Post EFIE RCS results (dB referenced to 1 m^2) for f = 2.3984 GHz with fits to the Mie Series (dB referenced to 1 m^2). Chi-Square (χ^2) goodness-of-fit (GOF) statistics indicate Gemini results are in excellent agreement with the Mie Series. VV and HH linear scatterplots show a nearly perfect correlation between Gemini RCS values, y_i, and Mie Series values, y_i*.



Frequency (GHz)
2.3984
Unknowns
298,863
Solution
EFIE
Nmax Terms
51
Agreement
Excellent
σ_{VV} χ^2 GOF
0.061
σ_{HH} χ^2 GOF
0.060

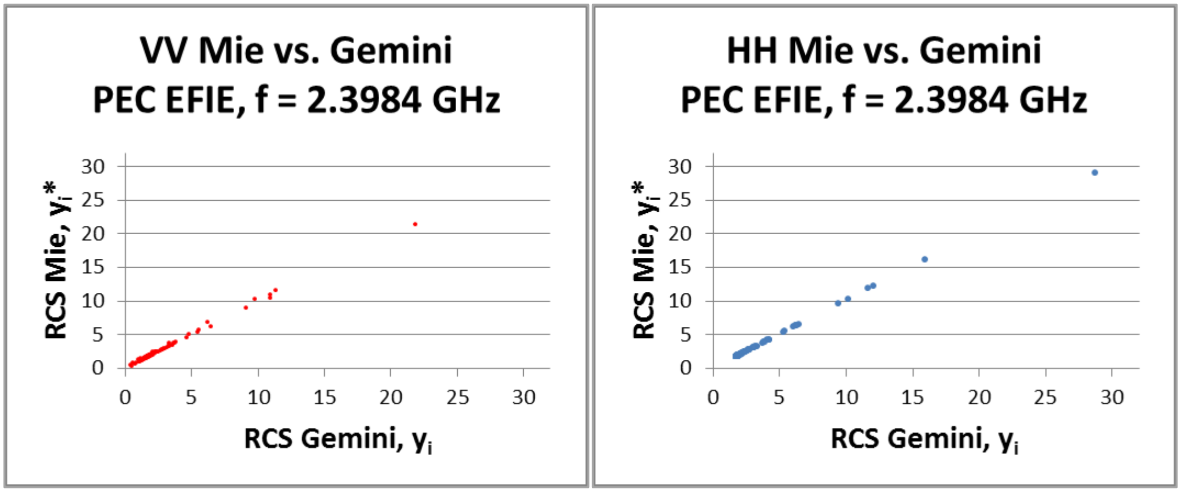
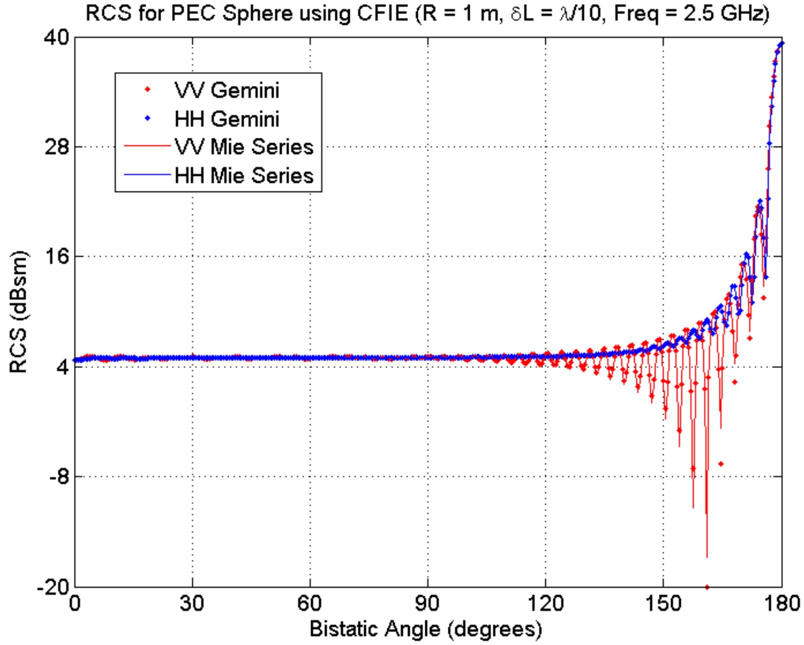


Figure 81: (Upper-Left) Gemini EFIE RCS results vs. Mie Series for f=2.3984 GHz (Upper-Right) Chi-Square Goodness-of-Fit statistics (Lower) Linear scatter plots of Mie Series vs. Gemini EFIE RCS results

Figure 82 shows VV (ϕ = 0°, 0° ≤ θ ≤ 180°) and HH (ϕ = 90°, 0° ≤ θ ≤ 180°) plots of GEMINI Post EFIE RCS results (dB referenced to 1 m^2) for f = 2.5000 GHz with fits to the Mie Series (dB referenced to 1 m^2). Chi-Square (χ^2) goodness-of-fit (GOF) statistics indicate Gemini results are in excellent agreement with the Mie Series. VV and HH linear scatterplots show a nearly perfect correlation between Gemini RCS values, y_i, and Mie Series values, y_i*.



Frequency (GHz)
2.5000
Unknowns
326,430
Solution
EFIE
Nmax Terms
53
Agreement
Excellent
σ_{VV} χ^2 GOF
0.063
σ_{HH} χ^2 GOF
0.059

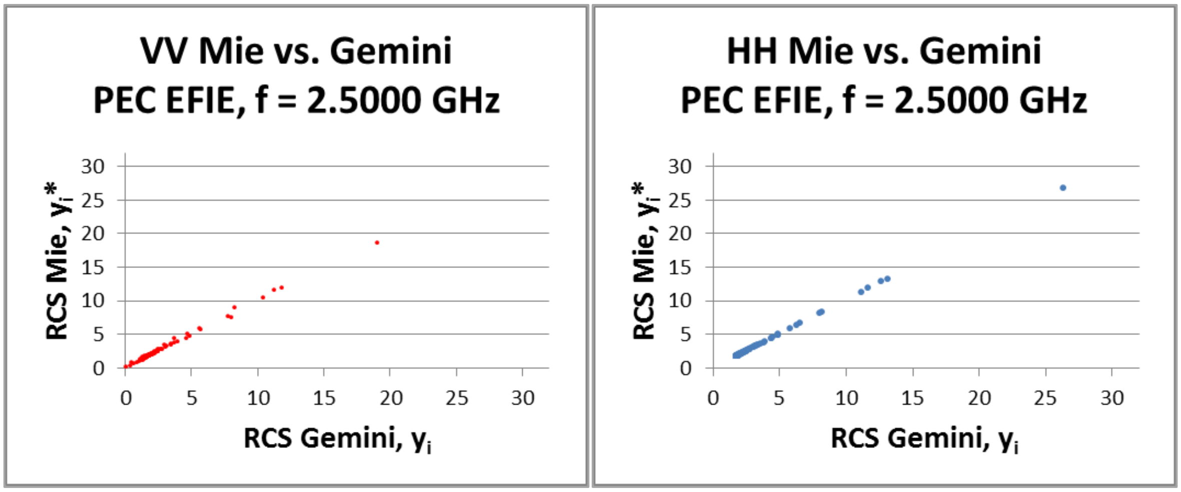
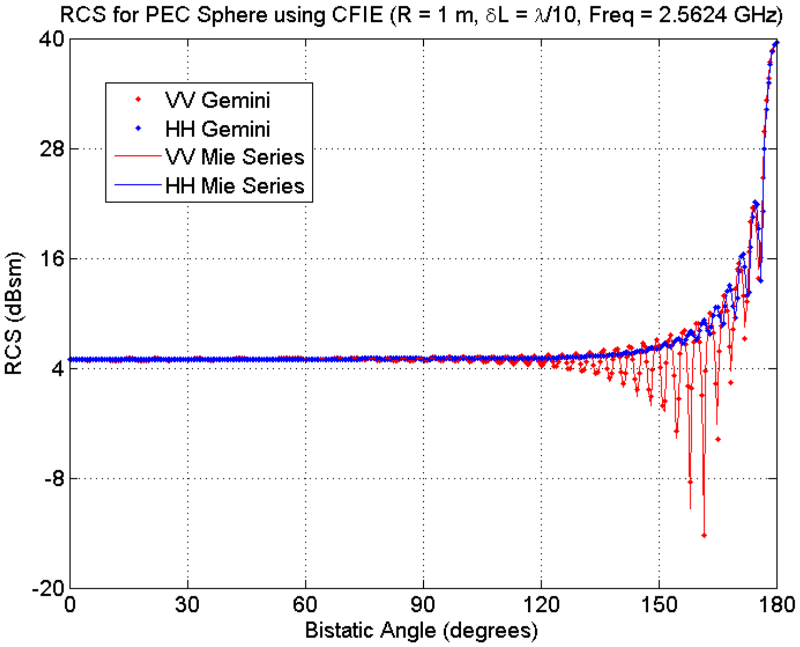


Figure 82: (Upper-Left) Gemini EFIE RCS results vs. Mie Series for f=2.5000 GHz (Upper-Right) Chi-Square Goodness-of-Fit statistics (Lower) Linear scatter plots of Mie Series vs. Gemini EFIE RCS results

Figure 83 shows VV (ϕ = 0°, 0° ≤ θ ≤ 180°) and HH (ϕ = 90°, 0° ≤ θ ≤ 180°) plots of GEMINI Post EFIE RCS results (dB referenced to 1 m^2) for f = 2.5624 GHz with fits to the Mie Series (dB referenced to 1 m^2). Chi-Square (χ^2) goodness-of-fit (GOF) statistics indicate Gemini results are in excellent agreement with the Mie Series. VV and HH linear scatterplots show a nearly perfect correlation between Gemini RCS values, y_i, and Mie Series values, y_i*.



Frequency (GHz)
2.5624
Unknowns
342,087
Solution
EFIE
Nmax Terms
54
Agreement
Excellent
σ_{VV} χ^2 GOF
0.060
σ_{HH} χ^2 GOF
0.056

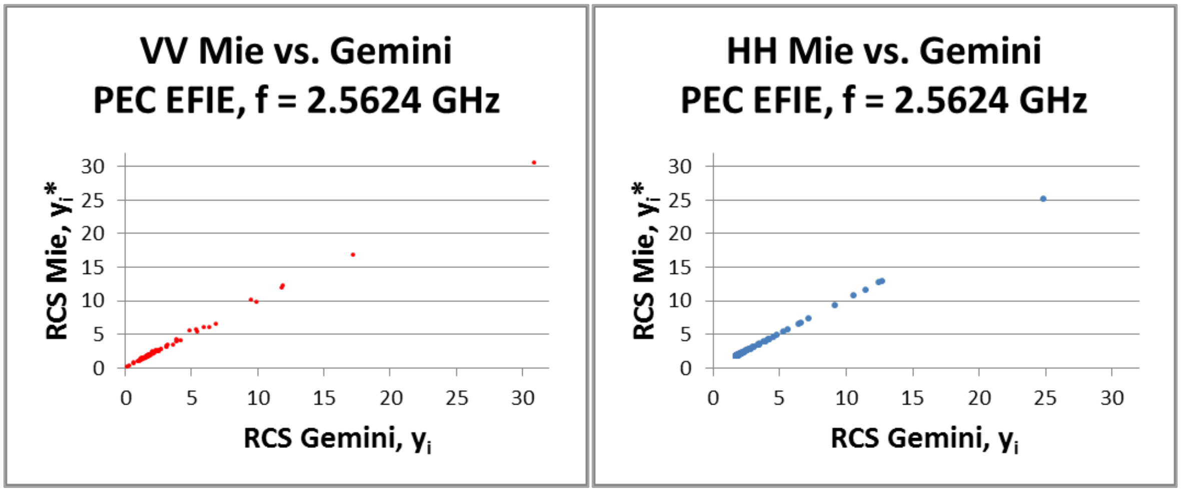


Figure 83: (Upper-Left) Gemini EFIE RCS results vs. Mie Series for f=2.5624 GHz (Upper-Right) Chi-Square Goodness-of-Fit statistics (Lower) Linear scatter plots of Mie Series vs. Gemini EFIE RCS results

Appendix B Final Implementation: Changes To GEMINI Solver v3.0

MODIFICATION #1

```
MODULE:  project_m.f90
SUBROUTINE:  createBasisLists
ENTRY AT ORIGINAL CODE LINE 831
!
!>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
!>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
!
!  Addition:  Nathan Champagne’s recommendation to refresh project obsBases and
!             scrBases to avoid race condition.
!
   CALL this%obsBases%refreshArray()
   CALL this%srcBases%refreshArray()
!
!>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
!>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
!
```

MODIFICATION #2

```
MODULE:  list_m.f90
ENTRY AT ORIGINAL CODE LINE 46
!
!>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
!>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
!
!  Modification:  Nathan Champagne’s recommendation to make refreshArray
!                 nonprivate
!
      PROCEDURE :: refreshArray
!      PROCEDURE, PRIVATE :: refreshArray
!
!>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
!>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
!
```

MODIFICATION #3

```
MODULE:  solution_m.f90
ENTRY AT ORIGINAL CODE LINE 25
!
!>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
!>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
!
!  Addition: Use OMP Library
!
   USE OMP_LIB
!
!>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
!>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
!
```

MODIFICATION #4

MODULE: solution_m.f90
SUBROUTINE: solution_nonperiodic
ENTRY AT ORIGINAL CODE LINE 119

```
!
!>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
!>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
!
!  Addition: Declare wall clock time
!
   REAL (KIND=8) :: WTIME
!
!>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
!>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
!
```

MODIFICATION #5

MODULE: solution_m.f90
SUBROUTINE: solution_nonperiodic
ENTRY AT ORIGINAL CODE LINE 222

```
!
!>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
!>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
!
!  Addition: Get initial wall clock time
!
      WTIME = OMP_GET_WTIME()
!
!>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
!>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
!
```

MODIFICATION #6

MODULE: solution_m.f90
SUBROUTINE: solution_nonperiodic
ENTRY AT ORIGINAL CODE LINE 223

```
!
!>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
!>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
!
!  Addition: Calculate and display "Elapsed Time" for global matrix fill
!
      WTIME = OMP_GET_WTIME() - WTIME
      WRITE (*,'(A,ES10.3E3)') '          Elapsed Time = ',WTIME
!
!>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
!>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
!
```

MODIFICATION #7

MODULE: solution_m.f90
SUBROUTINE: solution_periodic
ENTRY AT ORIGINAL CODE LINE 333

```
!
!>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
!>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
!
!  Addition: Declare wall clock time
!
   REAL (KIND=8) :: WTIME
!
!>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
!>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
!
```

MODIFICATION #8

MODULE: solution_m.f90
SUBROUTINE: solution_periodic
ENTRY AT ORIGINAL CODE LINE 450

```
!
!>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
!>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
!
!  Addition: Get initial wall clock time
!
      WTIME = OMP_GET_WTIME()
!
!>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
!>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
!
```

MODIFICATION #9

MODULE: solution_m.f90
SUBROUTINE: solution_periodic
ENTRY AT ORIGINAL CODE LINE 451

```
!
!>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
!>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
!
!  Addition: Calcualte and display "Elapsed Time" for global matrix fill
!
      WTIME = OMP_GET_WTIME() - WTIME
      WRITE (*,'(A,ES10.3E3)') '          Elapsed Time = ',WTIME
!
!>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
!>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
!
```

MODIFICATION #10

MODULE: commonregion_m.f90

ENTRY AT ORIGINAL CODE LINE 75

```
!
!>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
!>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
!
!  Addition: A private copy of CommonRegionStatic for each OMP thread
!
!$OMP THREADPRIVATE(CommonRegionStatic)
!
!>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
!>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
!
```

MODIFICATION #11

MODULE: homogeneousgreensfunction_m.f90

ENTRY AT ORIGINAL CODE LINE 43

```
!
!>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
!>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
!
!  Addition: A private copy of HomogenousGreensFunction for each OMP thread
!
!$OMP THREADPRIVATE(HomogenousGreensFunction)
!
!>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
!>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
!
!
```

MODIFICATION #12

MODULE: localmatrix_m.f90

ENTRY AT ORIGINAL CODE LINE 84

```
!
!>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
!>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
!
!  Addition: A private copy of LocalMatrix for each OMP thread
!
!$OMP THREADPRIVATE(LocalMatrix)
!
!>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
!>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
!
```

MODIFICATION #13

MODULE: localmatrix_m.f90
SUBROUTINE: fillMoM
ENTRY AT ORIGINAL CODE LINE 141

```
!
!>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
!>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
!
!  Addition: A private copy of variables for each OMP thread
!
!$OMP THREADPRIVATE(rs,unitNormal,l_vec,jac,srcWghtJacobian,srcArray)
!$OMP THREADPRIVATE(G,Kphi,Kpsi,Pz,Qz,gradG,grad_Kphi,grad_Kpsi,grad_Pz,grad_Qz)
!$OMP THREADPRIVATE(Ga,Gf,curl_Ga,curl_Gf)
!
!>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
!>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
!
```

MODIFICATION #14

MODULE: localmatrix_m.f90
SUBROUTINE: fillMoMThin
ENTRY AT ORIGINAL CODE LINE 642

```
!
!>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
!>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
!
!  Addition: A private copy of each variable for each OMP thread
!
!$OMP THREADPRIVATE(rs,unitNormal,l_vec,jac,srcWghtJacobian,srcArray)
!$OMP THREADPRIVATE(G,Kphi,Kpsi,Pz,Qz,gradG,grad_Kphi,grad_Kpsi,grad_Pz,grad_Qz)
!$OMP THREADPRIVATE(Ga,Gf,curl_Ga,curl_Gf)
!
!>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
!>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
!
```

MODIFICATION #15

MODULE: brickelement_m.f90
ENTRY AT ORIGINAL CODE LINE 288

```
!
!>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
!>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
!
!  Addition: A private copy of each variable for each OMP thread
!
!$OMP THREADPRIVATE(xiTemp,wghtTemp)
!
!>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
!>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
!
```

MODIFICATION #16

MODULE: prismelement_m.f90

ENTRY AT ORIGINAL CODE LINE 272

```
!
!>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
!>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
!
!  Addition: A private copy of each variable for each OMP thread
!
!$OMP THREADPRIVATE(xiTemp,wghtTemp)
!
!>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
!>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
!
```

MODIFICATION #17

MODULE: quadrilateralelement_m.f90

ENTRY AT ORIGINAL CODE LINE 220

```
!
!>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
!>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
!
!  Addition: A private copy of each variable for each OMP thread
!
!$OMP THREADPRIVATE(xiTemp,wghtTemp)
!
!>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
!>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
!
```

MODIFICATION #18

MODULE: triangleelement_m.f90

ENTRY AT ORIGINAL CODE LINE 217

```
!
!>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
!>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
!
!  Addition: A private copy of each variable for each OMP thread
!
!$OMP THREADPRIVATE(xiTemp,wghtTemp)
!
!>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
!>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
!
```

MODIFICATION #19

MODULE: wireelement_m.f90
ENTRY AT ORIGINAL CODE LINE 113

```
!
!>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
!>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
!
!  Addition: A private copy of each variable for each OMP thread
!
!$OMP THREADPRIVATE(xiTemp,wghtTemp)
!
!>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
!>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
!
```

MODIFICATION #20

MODULE: globalmatrix_m.f90
ENTRY AT ORIGINAL CODE LINE 21

```
!
!>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
!>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
!
!  Addition: Use OMP Library
!
   USE OMP_LIB
!
!>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
!>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
!
```

MODIFICATION #21

MODULE: globalmatrix_m.f90
SUBROUTINE: fillNormal
ENTRY AT ORIGINAL CODE LINE 64

```
!
!>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
!>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
!
!  Modification:  Make classPointer keep it's value between calls to fillNormal
!  Addition:  Make classPointer OMP Thread Private
!
!   CLASS(*), POINTER :: classPointer
   CLASS(*), POINTER, SAVE :: classPointer => null()
!$OMP THREADPRIVATE(classPointer)
!
!>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
!>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
!
```

MODIFICATION #22

MODULE: globalmatrix_m.f90

SUBROUTINE: fillNormal

ENTRY AT ORIGINAL CODE LINE 65

```
!
!>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
!>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
!
!  Addition: Variables needed for OMP threading
!
   INTEGER :: n_i,nodeSetCount,nodeCount,iNode
   INTEGER :: obsTID, srcTID, obsNTHREADS, srcNTHREADS,CHUNK, numUnknowns
   LOGICAL :: OMP_NESTED_FLAG
   CLASS(ElementType), POINTER :: element
   CLASS(NodeSetType), POINTER :: nodeSet
   TYPE(MatrixParametersType), POINTER :: matrixParameter
   matrixParameter => project%matrixParameters%at(1)
   numUnknowns = matrixParameter%numUnknowns
!
!>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
!>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
!
```

MODIFICATION #23

MODULE: globalmatrix_m.f90

SUBROUTINE: fillNormal

ENTRY AT ORIGINAL CODE LINE 77

```
!
!>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
!>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
!
!  Addition: Nathan Champagne’s recommendation to reduce the number of OMP
!            Critical Region directives needed to solve race conditions
!
   DO i = 1,elementCount
      element => project%elements%at(i)
      nodeSetCount = SIZE(element%nodeSets(:))
      DO n_i = 1,nodeSetCount
         nodeSet => element%nodeSets(n_i)%object
         nodeCount = element%nodeCount(n_i)
         DO iNode = 1,nodeCount
            IF(nodeSet%unknownFlags(1)) THEN
               CALL element%unknownJ(n_i)%node(iNode)%dofIds%refreshArray()
            ENDIF
            IF(nodeSet%unknownFlags(2)) THEN
               CALL element%unknownM(n_i)%node(iNode)%dofIds%refreshArray()
            ENDIF
         ENDDO
      ENDDO
   ENDDO
!
!>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
!>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
!
```

MODIFICATION #24

MODULE: globalmatrix_m.f90
SUBROUTINE: fillNormal
ENTRY AT ORIGINAL CODE LINE 77

```
!
!>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
!>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
!
!  Addition: Set up for Open MP parallelization of outer observation element loop
!
!  Set chuck size per thread
!
   CHUNK=1
!
!  Disable nested parallelism
!
   OMP_NESTED_FLAG=.FALSE.
   CALL OMP_SET_NESTED(OMP_NESTED_FLAG)
!
!  Display parallel environment
!
   IF (MPIWrapper%myIndex == 0) THEN
     WRITE (*,'(10X,A,I6,A,I2,A,I3)') 'Unknowns = ',numUnknowns,               &
                       '   Chunk =  ', CHUNK,                                   &
                       '   MPI processes = ',MPIWrapper%numberOfProcessors
      IF (OMP_GET_NESTED()) THEN
         WRITE (*,'(10X,A,L5)') 'OMP Nested Parallelism ENABLED!'
      ELSE
         WRITE (*,'(10X,A,L5)') 'OMP Nested Parallelism NOT enabled!'
      ENDIF
   ENDIF
!
!  Outer observation element loop forks here
!
!$OMP PARALLEL                                                                   &
!$OMP SHARED(project,gMatrix,freqIndex)                                          &
!$OMP SHARED(elementCount,freqPointer,modeIndex,omega)                           &
!$OMP PRIVATE(obsTID,i_t,n_t)                                                    &
!$OMP PRIVATE(obsElement,obsBasis,obsNodeSetCount,obsNodeSet,obsNodeCount)       &
!$OMP PRIVATE(obsArray)                                                          &
!$OMP PRIVATE(srcTID,i_s,n_s)                                                    &
!$OMP PRIVATE(srcElement,srcBasis,srcNodeSetCount,srcNodeSet,srcNodeCount)       &
!$OMP PRIVATE(commonRegions,regionCount)                                         &
!$OMP PRIVATE(obsSourceFlag,obsJsourceJ,obsJsourceM,obsMsourceJ,obsMsourceM)     &
!$OMP PRIVATE(obsIndex,srcIndex)
!
!  Dynamic Scheduling
!
!$OMP DO SCHEDULE(DYNAMIC, CHUNK)
!
!>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
!>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
!
```

MODIFICATION #25

MODULE: globalmatrix_m.f90
SUBROUTINE: fillNormal
ENTRY AT ORIGINAL CODE LINE 81

```
!
!>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
!>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
!
!  Addition: OMP Critical Region required to eliminate race condition in
!            obsElement => project%elements%at(i_t)
!
!$OMP CRITICAL
      classPointer => project%elements%at(i_t)
      SELECT TYPE(classPointer)
      CLASS IS(ElementType)
         obsElement => classPointer
      END SELECT
!$OMP END CRITICAL
!
!>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
!>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
!
```

MODIFICATION #26

MODULE: globalmatrix_m.f90
SUBROUTINE: fillNormal
ENTRY AT ORIGINAL CODE LINE 123 (FEM) & LINE 200 (MoM)

```
!
!>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
!>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
!
!  Addition: OMP Critical Region required to eliminate race condition in
!            srcElement => project%elements%at(i_s)
!
!$OMP CRITICAL
      classPointer => project%elements%at(i_t)
      SELECT TYPE(classPointer)
      CLASS IS(ElementType)
         srcElement=> classPointer
      END SELECT
!$OMP END CRITICAL
!
!>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
!>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
!
```

MODIFICATION #27

```
MODULE:  globalmatrix_m.f90
SUBROUTINE:  fillNormal
ENTRY AT ORIGINAL CODE LINE 368
!
!>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
!>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
!
!  Addition: Outer observation element loop joins here
!
!$OMP END PARALLEL DO
!
!>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
!>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
!
```

MODIFICATION #28

```
MODULE:  globalmatrix_m.f90
SUBROUTINE:  localToGlobal
ENTRY AT ORIGINAL CODE LINE 463
!
!>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
!>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
!
!  Modification: Initialize value to zero rather than getting value from gMatrix
!                to avoid OMP race condition
!
                  value = 0.0_dk
!                  value = gMatrix%getValue(rowDOF,columnDOF)
!
!>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
!>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
!
```

MODIFICATION #29

```
MODULE:  globalmatrix_m.f90
SUBROUTINE:  localToGlobal
ENTRY AT ORIGINAL CODE LINE 478
!
!>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
!>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
!
!  Modification: Update global matrix in OMP Critical Region to avoid OMP race
!                condition
!
!$OMP CRITICAL
                  value = value + gMatrix%getValue(rowDOF,columnDOF)
                  CALL gMatrix%putValue(rowDOF,columnDOF,value)
!$OMP END CRITICAL
!
!>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
!>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
!
```

References

[1] N. J. Champagne, R. M. Sharpe and J. W. Rockway, "EIGER: Electromagnetic Interactions GEneRalized," in *Department of Defense High Performance Computing Modernization Program Users Group Conference 2001*, Biloxi, 2001.

[2] NASA, "Computational Electromagnetics Laboratory," [Online]. Available: https://www.nasa.gov/centers/johnson/engineering/human_space_vehicle_systems/computational_electromagnetics_laboratory/index.html. [Accessed 11 February 2017].

[3] D. R. Wilton, "Computational Methods," in *Scattering: Scattering and Inverse Scattering in Pure and Applied Science*, London, Academic Press, 2002, pp. 316-365.

[4] X. Xu and R. J. Adams, "Sparse matrix factorization using overlapped localizing LOGOS modes on a shifted grid," *IEEE Transactions on Antennas and Propagation*, vol. 60, no. 3, pp. 1414-1424, 2012.

[5] K. F. Warnick, "Integral Equations and the Method of Moments," in *Numerical Methods for Engineering*, 1st ed., Provo, Scitech Publishing, 2011, pp. 151-209.

[6] S. A. Rao, D. R. Wilton and A. W. Glisson, "Electromagnetic Scattering by Surfaces of Arbitrary Shape," *IEEE Transactions on Antennas and Propagation*, vol. 30, no. 3, pp. 409-417, 1982.

[7] K. F. Warnick, "Introduction," in *Numerical Methods for Engineering*, 1st ed., Provo, Scitech Publishing, 2011, pp. 1-35.

[8] W. L. Stutzman and G. A. Thiele, "CEM for Antennas: The Method of Moments," in *Antenna Theory and Design*, 3rd ed., Hoboken, John Wiley & Sons, Inc., 2013, pp. 641-645.

[9] Argonne National Laboratory, "MPICH: A High-Performance, Portable Implementation of MPI Version 1.4.1p1," 24 August 2011. [Online]. Available: http://www.mpich.org/static/tarballs/1.4.1p1/. [Accessed 13 July 2014].

[10] Open MPI Development Team, "Open MPI: Open Source High Performance Computing," [Online]. Available: https://www.open-mpi.org/. [Accessed 4 March 2017].

[11] University of Kentucky I.T.S., "The Lipscomb High Performance Computing Cluster," [Online]. Available: http://www.uky.edu/its/hpc/hardware. [Accessed 11 February 2017].

[12] OpenMP Architecture Review Board, "OpenMP," [Online]. Available: http://www.openmp.org/. [Accessed 4 March 2017].

[13] Sandia National Laboratory, "The CUBIT Geometry and Mesh Generation Toolkit Version 14.1," 13 January 2014. [Online]. Available: https://cubit.sandia.gov/public/14.1/Cubit-14.1-announcement.html. [Accessed 5 August 2014].

[14] J. B. Grant, "EIGER ANTS (Electromagnetic Interactions GEneRalized Advanced Numerical Tools & Services) v2.7," ANT-S, Livermore, 2000.

[15] G. T. Ruck and et. al., "Spheres," in *Radar Cross Section Handbook*, 1st ed., New York; London, Plenum Press, 1970, pp. 141-202.

[16] W. Gibson, "Scattered Field of a Conducting and Stratified Sphere," 30 April 2013. [Online]. Available: http://www.mathworks.com/matlabcentral/fileexchange/20430-scattered-field-of-a-conducting-and-stratified-spheres/content/mie.zip. [Accessed 16 May 2014].

[17] B. Chapman, G. Jost and R. van der Pas, "Using OpenMP in the Real World," in *Using OpenMP*, 1st ed., Cambridge, The MIT Press, 2008, pp. 191-242.

[18] B. Chapman, G. Jost and R. van der Pas, "Overview of OpenMP," in *Using OpenMP*, 1st ed., Cambridge, The MIT Press, 2008, pp. 23-34.

[19] B. Chapman, G. Jost and R. van der Pas, "Troubleshooting," in *Using OpneMP*, 1st ed., Cambridge, The MIT Press, 2008, pp. 243-276.

[20] B. Chapman, G. Jost and R. van der Pas, "Writing a First OpenMP Program," in *Using OpenMP*, 1st ed., Cambridge, The MIT Press, 2008, pp. 35-50.

[21] B. Chapman, G. Jost and R. van der Pas, "OpenMP Language Features," in *Using OpenMP*, 1st ed., Cambridge, The MIT Press, 2008, pp. 51-124.

[22] J. C. Young, *DLX Batch Script with Sanity Checks*, Lexington: University of Kentucky, 2015.

[23] J. C. Young, *DLX Run Script to Execute GEMINI Solver v3.0 Test Sets*, Lexington: Univeristy of Kentucky, 2015.

Vita

PLACE OF BIRTH	New York City, New York
DEGREES AWARDED	Ph.D., Experimental Nuclear Physics, May 1994 University of Kentucky
	M.S., Experimental Nuclear Physics, December 1991 University of Kentucky
	B.S., Physics, December 1990 University of Kentucky
PROFESSIONAL POSITIONS	Professor of Physics (Retired), 1995-2013 Kentucky Wesleyan College
	Visiting Professor of Physics, 2008-2009 University of Kentucky
	Research Fellow in Medical Physics, 1999-2000 Vanderbilt University Medical Center
SCHOLASTIC HONORS	Teacher of the Year, 2007 Kentucky Wesleyan College
	Achievement in Education, 2004 Kentucky Society of Professional Engineers
	Teacher of the Year, 2003 Kentucky Wesleyan College
	Graduate Fellowship, 1991-1994 National A.N.N. Fellowship Program
	Graduate Fellowship, 1990-1991 University of Kentucky Graduate School
NAME ON FINAL COPY	Buxton L. Johnson, Sr.