# OpenDDA - A Novel High-Performance Computational Framework for the Discrete Dipole Approximation

A Dissertation Submitted in Accordance with
the Requirements for the Degree of
**Doctor of Philosophy**
in the Faculty of Science

**by**


## James Mc Donald


**Supervisors:**

Dr. Aaron Golden
Prof. S. Gerard Jennings


National University of Ireland, Galway
*Ollscoil na hÉireann, Gaillimh*

Computational Astrophysics Laboratory
National University of Ireland, Galway


September 2007

# Approval

"OpenDDA - A Novel High-Performance Computational Framework for the Discrete Dipole Approximation", a dissertation prepared by James Mc Donald and submitted in accordance with the requirements for the Degree of **Doctor of Philosophy** in the Faculty of Science has been approved and accepted by:

| | |
|---|---|
| Dr. Aaron Golden<br>Thesis Supervisor | Date |
| Prof. S. Gerard Jennings<br>Thesis Supervisor | Date |
| External Examiner | Date |
| Internal Examiner | Date |

# Dedication

*This work is dedicated...*

To my family, John, Sarah, Alice, Philip, Sheila, Mum and Dad. We have always been extremely close-knit and I would like to extend my deepest gratitude for their tireless support and encouragement through both the highs and the lows.

To my wingman, my twin brother John, who has always had my back, come what may. With his absolute strength of character, steadfast fortitude and unflinching perseverance, he manages to keep, not only himself, but all those around him, on the straight and narrow. I have always considered myself the DeVito to his Schwarzenegger. Vires quod veneratio.

To my beloved father, John Athelstan Mc Donald, who sadly passed away during the course of this work. Although he never truly got to grips with the modern age and with what I was doing and always thought I would have made a fine welder, he exemplified morality and humility, and often went above and beyond the call of duty in the name of a good cause that most people would have dismissed in an instant. He always had a smile no matter how bleak the outlook and brightened the lives of everyone that knew him. I count myself truly blessed to have had the benefit of his wisdom and guidance and would certainly not be half the man I am today without his influence. Dad, may the wind always be at your back, the sun upon your face and may the wings of destiny carry you aloft to dance with the stars.

# Philosophy

*"Sometimes you're flush and sometimes you're bust, and when you're up, it's never as good as it seems, and when you're down, you never think you'll be up again, but life goes on."* - Fred Jung

*"Regret for the things we did can be tempered by time; it is regret for the things we did not do that is inconsolable."* - Sydney J. Harris

*"Never lose your sense of humour..."* - John Athelstan Mc Donald

# Abstract

## OpenDDA - A Novel High-Performance Computational Framework for the Discrete Dipole Approximation

**James Mc Donald**

Submitted for the Degree of Doctor of Philosophy
September 2007

## Abstract

As the title suggests, this discourse presents a new highly-optimised computational infrastructure for the Discrete Dipole Approximation (DDA). The DDA is an intuitive and extremely flexible numerical method for calculating the optical properties associated with a target of arbitrary geometry, whose dimensions are comparable to or moderately larger than the incident wavelength. The new framework, which has been dubbed OpenDDA, has been meticulously custom-built from scratch in the C language and incorporates a host of new and novel optimisations. During the design and development process, every conceivable effort was made, at every stage, to maximise the performance, portability and maintainability of each of the components and the new framework as a whole.

The infrastructure includes a whole host of iterative methods that have been custom-built to integrate seamlessly with the new highly-optimised DFT kernel that efficiently calculates the matrix-vector products that are required by these iterative schemes. The new DFT kernel performs the requisite 3D DFTs as ensembles of 1D transforms, and by doing so, due to the manipulations that are required to facilitate their use, is able to remove a significant fraction of the constituent 1D transforms which are superfluous. Not only does this optimisation provide a significant reduction in the required memory and a comparably significant improvement in the performance, but it is also permits parallel techniques to be employed to further enhance the performance. Full OpenMP-based shared-memory and MPI-based distributed-memory implementations of the new framework have been created to take proper advantage of the various different architectural environments that exist.

Both possess extremely good scalability and associated performance. For the distributed-memory version, novel parallel memory allocation and domain decomposition schemes have been employed, which, apart from providing a decrease in memory per node that is directly proportional to the number of nodes, also provide inherent load balancing. To support the distributed nature, local transpose and several distributed-transpose algorithms have been custom-designed and integrated into the framework to support the distributed DFTs required by the distributed DFT kernel. On top of the architectural specific nature of the implementations, each also exists in single, double and long double precision.

Apart from the main optimisations associated with the iterative core, the infrastructure also incorporates many miscellaneous advancements. Bit-fielding of the integer storage associated with the spatial and compositional specification of the target provides a significant reduction in the required storage and furthermore, supports an efficient and lightweight distributed-memory implementation. On top of this, OpenDDA also incorporates improved Akima interpolation and SFMT Mersenne Twister pseudorandom number generation. The DFT functionality that underpins the computational efficiency of the framework is provided by the extremely efficient, highly-portable and well known FFTW package (version 3), and for guaranteed portability, the requisite complex number support, that is not native in the ANSI C89 standard, has been custom-built. The framework also includes an extremely intuitive and user-friendly interface for specifying the simulation parameters. This includes a sturdy and forgiving parser and a straightforward human-readable input control file.

The results include a large number of serial, shared-memory and distributed-memory benchmarks that illustrate the profound significance of these optimisations. They include benchmarks for the DFT kernel itself, showing the step-by-step application of the various optimisations to elucidate their respective contribution to the cumulative effect. Benchmarks are also performed to verify the performance gains and high degree of scalability associated with both parallel implementations. The new framework is also put in context via several comparative benchmarks with alternate DDA implementations. Finally, a quick comparison with the output of a pre-existing DDA implementation is performed for operational verification of the new framework.

# Declarations

The work in this thesis is based on research carried out at the Computational Astrophysics Laboratory, Department of Information Technology, National University of Ireland, Galway, Ireland. No part of this thesis has been submitted elsewhere for any other degree or qualification and is all my own work unless referenced to the contrary in the text.

**Copyright © 2007 by James Mc Donald**.

"The copyright of this thesis rests with the author. No quotations from it should be published without the author's prior written consent and information derived from it should be acknowledged".

# Acknowledgements

I would like to sincerely thank my supervisors for giving me the opportunity to work with them and for the freedom to take the helm and chart my own course, no matter the squalls. I am truly grateful for their kind support and encouragement through the years.

On a personal note, I would especially like to acknowledge the multi-talented people that I have had the pleasure of sharing an office with and working alongside over the years, John, Gregg, Ger, Paddy, Mike, Steve, Martin, Padraig, Niall, Rory, Mark, Weston, Seamus, Symeon, Oisin, Imelda, Brendan and the two Ronans. I would also like to give a special mention to the SOF2 gang with whom I had truly great times blowing off steam in *barn*.

I would like to thank the Irish Centre for High-End Computing (ICHEC)[1] and their highly skilled and dedicated Systems Team.

On a separate note, I would like to thank Subversion[2], the version control software, not the act. It is a must for all software development and thesis writing. It saved my bacon on more than one occasion and without it I would have definitely have gone off the proverbial deep end and would now be living as some sort of feral recluse in the hills of Connemara.

This work was supported by the CosmoGrid Project[3], funded under the Programme for Research in Third Level Institutions (PRTLI) administered by the Irish Higher Education Authority under the National Development Plan and with partial support from the European Regional Development Fund.

[1] Irish Centre for High-End Computing, http://www.ichec.ie
[2] Subversion, http://subversion.tigris.org
[3] CosmoGrid Project, http://www.cosmogrid.ie

# Acronyms

| | |
|---|---|
| **%** | Modulo function (C language notation) |
| **(int)()** | Integer cast (C language notation) |
| **1D** | One-dimension(al) |
| **2D** | Two-dimension(al) |
| **3D** | Three-dimension(al) |
| **API** | Application Programming Interface |
| **CG** | Conjugate Gradient |
| **DDA** | Discrete Dipole Approximation |
| **DFT** | Discrete Fourier Transform |
| **FD** | Fourier Domain |
| **FFTW** | Fastest Fourier Transform in the West |
| **iDFT** | inverse Discrete Fourier Transform |
| **IDP** | Interplanetary Dust Particle |
| **ISM** | Interstellar Medium |
| **LHS** | Left-hand side |
| **MGS** | Modified Gram-Schmidt |
| **MPI** | Message Passing Interface |
| **OpenMP** | Open Multi-Processing |
| **PM** | Particulate Matter |
| **RHS** | Right-hand side |
| **SEM** | Scanning Electron Microscope |
| **SFMT** | SIMD-oriented Fast Mersenne Twister |
| **T-to-c** | Toeplitz-to-circulant |

# Contents

**Approval** ii

**Dedication** iii

**Philosophy** iv

**Abstract** v

**Declarations** vii

**Acknowledgements** viii

**Acronyms** ix

**1 Introduction** 1

1.1 General introduction . . . . . 1

1.2 Discrete Dipole Approximation . . . . . 3

1.2.1 Basic Theory . . . . . 3

1.2.2 Applications . . . . . 8

1.2.2.1 Cosmic Dust . . . . . 9

1.2.2.2 Atmospheric Particulates . . . . . 13

1.2.2.3 Surface Features . . . . . 18

1.2.2.4 Biological Cells . . . . . 20

1.3 Reference frames & target rotation . . . . . 24

1.4 Computational motivation . . . . . 28

1.5 Thesis organisation . . . . . 29

**2 Computational Infrastructure** **31**

2.1 Introduction . . . . . 31

2.2 Bit-fielding data . . . . . 32

2.2.1 Bit-fielding of the target occupation data . . . . . 33

2.2.2 Dynamic bit-fielding of the target composition data . . . . . 35

2.3 Data interpolation & extrapolation . . . . . 38

2.4 DFT functionality . . . . . 43

2.5 Complex functionality in C . . . . . 46

2.6 Iterative methods . . . . . 48

2.6.1 Conjugate Gradient basics . . . . . 48

2.6.2 Algorithmic variants & custom-built implementations . . . . . 53

2.6.3 Preconditioning . . . . . 56

2.6.4 Pseudorandom number generation & vector orthonormalisation 58

**3 DFT kernel - Serial implementation** **61**

3.1 Introduction . . . . . 61

3.2 Toeplitz and circulant theory . . . . . 62

3.3 Circulant diagonalisation . . . . . 64

3.3.1 Level-one diagonalisation . . . . . 64

3.3.2 Level-two diagonalisation . . . . . 68

3.3.3 Level-three diagonalisation . . . . . 72

3.4 The nature of the problem . . . . . 75

3.5 Standard DFT algorithm . . . . . 79

3.6 Algorithmic optimisation . . . . . 86

**4 DFT kernel - Parallel Implementation** **95**

4.1 Introduction . . . . . 95

4.2 Shared-memory implementation - OpenMP . . . . . 96

4.2.1 Library routines and compiler directives . . . . . 96

4.2.2 Thread safety . . . . . 99

4.3 Distributed-memory implementation - MPI . . . . . 100
4.3.1 Domain decomposition . . . . . 101
4.3.2 Parallel memory allocation . . . . . 104
4.3.3 Distributed transpose design considerations . . . . . 111
4.3.4 Message passing protocols . . . . . 113
4.3.5 Local transposes . . . . . 115
4.3.6 Exploiting the eager protocol . . . . . 121
4.3.7 Moving beyond the protocols . . . . . 126
4.3.8 Various distributed-transpose algorithms . . . . . 137

**5 Code description 140**
5.1 Introduction . . . . . 140
5.2 Code structure, versions & variants . . . . . 141
5.3 Program control, inputs & outputs . . . . . 144
5.4 Memory estimation & performance timing . . . . . 154
5.5 Compiling & running OpenDDA . . . . . 158

**6 Benchmarking & Analysis 161**
6.1 Benchmarking architectures . . . . . 161
6.2 DFT kernel performance . . . . . 163
6.2.1 Serial performance . . . . . 163
6.2.2 Parallel performance . . . . . 170
6.3 Full system performance . . . . . 183
6.4 Code comparison . . . . . 188
6.5 Result verification . . . . . 199

**7 Conclusions & Future work 201**
7.1 Conclusions . . . . . 201
7.2 Future work . . . . . 206

**Appendices 227**

**A Circulant proofs** **227**

A.1 Show that the downward-shift permutation matrix ${}^{[n]}\psi$ is a fundamental building block for circulant structure . . . . . 227

A.2 Show that the Fourier matrix ${}^{[n]}\mathrm{F}$ diagonalises ${}^{[n]}\psi$ . . . . . 230

A.3 Show that an arbitrary circulant matrix ${}^{[n]}\mathrm{C}$ is diagonalised by the Fourier matrix ${}^{[n]}\mathrm{F}$ . . . . . 231

A.4 Construction of a circulant of level-two and order $[J'][K']$ . . . . . 233

A.5 The diagonalisation of a level-two circulant matrix by the level-two Fourier matrix . . . . . 235

A.6 Construction of a circulant of level-three and order $[P'][J'][K']$ . . . . 236

A.7 The diagonalisation of a level-three circulant matrix by the level-three Fourier matrix . . . . . 238

**B OpenMP library routines and compiler directives** **240**

B.1 OpenMP library routines . . . . . 241

B.2 OpenMP compiler directives . . . . . 241

B.2.1 Clauses . . . . . 242

**C Various iterative methods** **244**

C.1 Conjugate-Gradients: CG . . . . . 245

C.2 Conjugate-Gradients Squared: CGS . . . . . 246

C.3 BiConjugate-Gradients: BiCG . . . . . 247

C.4 BiConjugate-Gradients for symmetric systems: $\mathrm{BiCG}_{sym}$ . . . . . 248

C.5 Stabilised version of the BiConjugate-Gradients: BiCGSTAB . . . . . 249

C.6 Restarted, stabilised version of the BiConjugate-Gradients: RBiCGSTAB 250

C.7 Quasi-minimal residual with coupled two-term recurrences: QMR . . 252

C.8 Quasi-minimal residual for symmetric systems: $\mathrm{QMR}_{sym}$ . . . . . 253

C.9 Transpose-free quasi-minimal residual: TFQMR . . . . . 254

C.10 A variant of BiCGSTAB based on multiple Lanczos starting vectors: $\mathrm{ML(n)BiCGSTAB}_{orig}$ . . . . . 256

C.11 A variant of BiCGSTAB based on multiple Lanczos starting vectors: ML(n)BiCGSTAB . . . . . 258
C.12 A variant of BiCGSTAB based on multiple Lanczos starting vectors: ML(n)BiCGSTAB$_{ss}$ . . . . . 260

**D Full system performance tables 263**

# List of Figures


1.1 Atom in an external electric field. . . . . 4

1.2 Schematic of the electric field and the lines of equipotential (bottom right panel) of a dipole. . . . . 5

1.3 Dipole representation of a tetrahedral and a spherical scatterer, composed of $N_d = 23001$ and $N_d = 65117$ dipoles, respectively. . . . . 6

1.4 A schematic of the general scattering problem. . . . . 8

1.5 Scanning Electron Microscope (SEM) images of cosmic dust particles. 9

1.6 Natural sources of atmospheric particulates. . . . . 14

1.7 The real component, imaginary component, and magnitude of the refractive index for aluminium (Al), silicon (Si), and tungsten (W) in the visible spectrum. . . . . 19

1.8 Scanning Electron Microscope (SEM) images of human blood cells. . 21

1.9 Disparity between the cross section of a cylinder and that of an erythrocyte. . . . . 24

1.10 Laboratory frame → target frame transformation via the three Euler angles $(\phi, \theta, \psi)$. . . . . 25

1.11 Target frame→laboratory frame transformation via the three Euler angles $(-\psi, -\theta, -\phi)$. . . . . 27

2.1 Refractive index versus wavelength ($\mu$m) showing 175 interpolated points. . . . . 42

3.1 Outlines the transformation of the matrix structure for ${}^{[J'][K']}\mathrm{c} \longrightarrow {}^{[J'][K']}\mathrm{C2D}$ 71

3.2 Outlines the transformation of the matrix structure for ${}^{[P'][J'][K']}\mathrm{c} \longrightarrow {}^{[P'][J'][K']}\mathrm{C3D}$. . . . . 74

3.3 The exemplar $K = J = P = 3$ dipole array used to explain the nature of the DDA problem. . . . . 76
3.4 The structure of the nine component matrices of the full interaction matrix is that of a level-three Toeplitz matrix. . . . . 78
3.5 Outlines the T-to-c conversion of the six tensor component arrays, the zero-padding of the three vector component arrays and subsequent DFTs & iDFT for the original algorithm. . . . . 80
3.6 A flowchart for the DFT matrix-vector product as prescribed by the standard algorithm. . . . . 83
3.7 Outlines the implicit T-to-c conversion and simultaneous DFT of the tensor component arrays as implemented in the new algorithm. . . . . 87
3.8 Outlines the zero-padding and the subsequent optimised DFT of the three vector component arrays and also helps to envisage the iDFT where only transforms that contribute to the final requisite answer are performed. . . . . 90
3.9 A flowchart for the DFT matrix-vector product as prescribed by the new algorithm. Includes the serial, OpenMP and MPI implementations. 92

4.1 Domain decomposition implemented using 'cyclic-plane' distribution whereby planes are cyclically assigned to the 3 available processors, *rank=0,1,2*, using the *modulo* function. . . . . 102
4.2 Load balance comparison for block and 'cyclic-plane' decomposition for a simplistic spherical target based on an 18×18×18 computational grid. . . . . 103
4.3 Schematic for parallel memory allocation across 3 nodes showing the allocation in $xy$-plane (a) and in $xz$-plane (b) configurations. . . . . 105
4.4 Flowchart for the algorithm which determines the maximum storage required by a node to implement both $xy$-plane & $xz$-plane configurations. . . . . 107

4.5 Schematic showing the distributed element-wise multiplication and the extra allocation required for each of the six independent tensor components for the MPI version of the DFT matrix-vector kernel. . . . . 109
4.6 Distributed transpose requirements for the DFT kernel. . . . . 112
4.7 Flowchart for the local non-square transpose algorithm. . . . . 116
4.8 Memory layout for the $xy$-plane and $xz$-plane configurations for both the square (a) & non-square (b) local transposes and (c), the memory transformation cycles involved in each transpose. . . . . 117
4.9 Flowchart for the 'in-situ' point-to-point distributed transpose. . . . . 123
4.10 Schematic for the forward $xy$-to-$xz$ 'MPI_Type_vector'-based distributed transpose. . . . . 128
4.11 Schematic for the reverse $xz$-to-$xy$ 'MPI_Type_vector'-based distributed transpose. . . . . 130
4.12 Flowchart for the calculation of the buffer requirements for the main 'MPI_Type_vector'-based distributed transpose. . . . . 132
4.13 Flowchart for the communication phase of the main 'MPI_Type_vector'-based distributed transpose. . . . . 136

5.1 A plot of the distribution of 12000 points on a unit sphere. . . . . 148
5.2 A schematic of the scattering conventions for specification of the scattering directions using the two spherical coordinates angles. . . . . 150

6.1 A short-range comparison of the different algorithms that can be applied to calculate the matrix-vector product to allow the inclusion of the explicit calculation using the CBLAS zgemv function from Intel MKL. . . . . 167

6.2 A long-range comparison of the different algorithms that can be applied to calculate the matrix-vector product to illustrate the high end behaviour of the algorithm and to show that the new algorithm can be applied to systems long after the original algorithm has become impracticable. . . . . 168
6.3 OpenMP speedup achieved across 4, 8 & 16 threads, respectively, for the long-range case shown in Fig. 6.2. . . . . 169
6.4 A long-range comparison of the time taken to create the FFTW plan using FFTW_MEASURE for the different algorithms. . . . . 170
6.5 Approximate memory required to perform the matrix-vector product with double-complex accuracy. . . . . 171
6.6 OpenMP scalability and timing run for a spherical target with $N_d = 113,089,912$ occupied sites. (Scalability view) . . . . . 172
6.7 OpenMP scalability and timing run for a spherical target with $N_d = 113,089,912$ occupied sites. (Component view) . . . . . 173
6.8 Speedups associated with the OpenMP scalability and timing run for a spherical target with $N_d = 113,089,912$ occupied sites. . . . . 174
6.9 MPI scalability and timing run with associated speedups for a spherical target with $N_d = 113,089,912$ occupied sites. . . . . 177
6.10 A heuristic plot for the maximum number of dipoles that can be used within the standalone double-precision DFT matrix-vector kernel for a spherical target as a function of the number of processors (assuming 2 GB per node for data storage). . . . . 179
6.11 Simple in-place local transpose for a square system, i.e., simple flip across the diagonal. . . . . 180
6.12 A comparison of the custom non-square and basic square in-place local transpose algorithms. . . . . 181
6.13 A comparison of the custom in-place and simple out-of-place non-square local transpose algorithms. . . . . 181

6.14 A comparison of the serial, OpenMP-based shared-memory and MPI-based distributed-memory implementations. . . . . 183
6.15 The performance, on NEMO, of the full serial system as a function of the iterative scheme and, where applicable, the number of starting vectors. . . . . 185
6.16 The performance, using 4 threads on NEMO, of the full OpenMP-based shared-memory system as a function of the iterative scheme and, where applicable, the number of starting vectors. . . . . 186
6.17 A heuristic plot for the maximum number of dipoles that can be used within the full system as a function of the number of processors and the iterative scheme (assuming 2 GB per node for data storage and, where appropriate, three starting vectors). . . . . 189

D.1 The performance, using 4 processors on NEMO, of the full MPI-based distributed-memory system as a function of the distributed-transpose algorithm employed to support the DFT kernel, the iterative scheme, and where applicable, the number of starting vectors. . . . . 264
D.2 The performance, using 4 nodes on WALTON, of the full MPI-based distributed-memory system as a function of the distributed-transpose algorithm employed to support the DFT kernel, the iterative scheme, and where applicable, the number of starting vectors. . . . . 265
D.3 The performance, using 8 nodes on WALTON, of the full MPI-based distributed-memory system as a function of the distributed-transpose algorithm employed to support the DFT kernel, the iterative scheme, and where applicable, the number of starting vectors. . . . . 266
D.4 The performance, using 16 nodes on WALTON, of the full MPI-based distributed-memory system as a function of the distributed-transpose algorithm employed to support the DFT kernel, the iterative scheme, and where applicable, the number of starting vectors. . . . . 267

D.5 The performance, using 32 nodes on WALTON, of the full MPI-based distributed-memory system as a function of the distributed-transpose algorithm employed to support the DFT kernel, the iterative scheme, and where applicable, the number of starting vectors. . . . . 268

D.6 The performance, using 64 nodes on WALTON, of the full MPI-based distributed-memory system as a function of the distributed-transpose algorithm employed to support the DFT kernel, the iterative scheme, and where applicable, the number of starting vectors. . . . . 269

# List of Tables

1.1 Approximate size ranges, mean corpuscular volume (MPV), diameter of a sphere of equal volume, and diameter-to-wavelength ratio & size parameter at both ends of the visible spectrum for an erythrocyte. . . 22

1.2 Approximate number of dipoles and memory, using *single* precision, required in the discrete representation of an erythrocyte. . . . . 23

3.1 Signs to account for the non-symmetries in sign only for the T-to-c conversions of the six independent tensor components. . . . . 81

3.2 Complexity estimations for the original and new algorithms. . . . . 94

4.1 Memory transformation cycles for the non-square local forward $xy$-to-$xz$ transpose of the example depicted in Fig. 4.8b. . . . . 120

4.2 Logical flow of the complete non-square local forward $xy$-to-$xz$ transpose algorithm. . . . . 121

4.3 Calculation of the buffer requirements for the forward $xy$-to-$xz$ distributed transpose of the given example. . . . . 134

6.1 Comparison of the number of 1D transforms that are required in order to carry out the matrix-vector product for the original and new algorithms. . . . . 164

6.2 Comparison of the complex number storage for the kernel of the matrix-vector product for the original and new algorithms. . . . . 165

6.3 Comparison of the number of dipoles, $N_d$, in the representation of the spherical target and the associated memory usage for the chosen upper limits of the benchmarking range for the various algorithms. . . . . 166

6.4 Total memory required and associated maximum memory requirement per node for the performance and scalability benchmark of the main MPI-based distributed-memory implementation of the new algorithm. 176

6.5 The relevant system sizes for the distributed benchmarks of the MPI-based variant of the OpenDDA framework, including the number of sites in the computational domain, $N$, and the number of occupied sites in the representation of the spherical target, $N_d$. . . . . 187

6.6 Comparative benchmark run on FRANKLIN between DDSCAT 6.1 and the single-precision serial and OpenMP-based shared-memory variants of OpenDDA 0.41, using BiCGSTAB, for $K = J = P = 200$, which produced a computational domain with $N = 8 \times 10^6$ sites in total and a representation of the spherical target with $N_d = 4,188,896$ occupied sites. . . . . 190

6.7 Comparative benchmark run on FRANKLIN between ADDA 0.76 and the double-precision serial and OpenMP-based shared-memory variants of OpenDDA 0.41, using BiCGSTAB, $\mathrm{BICG}_{\mathrm{sym}}$ & $\mathrm{QMR}_{\mathrm{sym}}$, for $K = J = P = 200$, which produced a computational domain with $N = 8 \times 10^6$ sites in total and a representation of the spherical target with $N_d = 4,188,896$ occupied sites. . . . . 192

6.8 Comparative benchmark run on FRANKLIN between ADDA 0.76 and the double-precision serial and OpenMP-based shared-memory variants of OpenDDA 0.41, using $\mathrm{QMR}_{\mathrm{sym}}$, for $K = J = P = 400$, which produced a computational domain with $N = 64 \times 10^6$ sites in total and a representation of the spherical target with $N_d = 33,511,376$ occupied sites. . . . . 193

6.9 Comparative benchmark run on NEMO between ADDA 0.76 and the double-precision serial and OpenMP-based shared-memory variants of OpenDDA 0.41, using BiCGSTAB, $\text{BICG}_{\text{sym}}$ & $\text{QMR}_{\text{sym}}$, for $K = J = P = 200$, which produced a computational domain with $N = 8 \times 10^6$ sites in total and a representation of the spherical target with $N_d = 4,188,896$ occupied sites. . . . . 194

6.10 Comparative benchmark run on HOBBES between ADDA 0.76 and the double-precision serial and OpenMP-based shared-memory variants of OpenDDA 0.41, using BiCGSTAB, $\text{BICG}_{\text{sym}}$ & $\text{QMR}_{\text{sym}}$, for $K = J = P = 200$, which produced a computational domain with $N = 3,375,000$ sites in total and a representation of the spherical target with $N_d = 1,768,496$ occupied sites. . . . . 195

6.11 System sizes, associated size of the computational domain and the number of occupied sites in the computational domain for the MPI-based comparative benchmark with ADDA across on 4, 8, 16, 32 & 64 nodes on WALTON. . . . . 196

6.12 Comparative benchmark run on WALTON between ADDA 0.76 and the double-precision MPI-based distributed-memory variant of OpenDDA 0.41, using BiCGSTAB, $\text{BICG}_{\text{sym}}$ & $\text{QMR}_{\text{sym}}$, across 4, 8, 16, 32 & 64 nodes. . . . . 197

6.13 Results for the operational verification of the new OpenDDA framework. 199

# List of Code Excerpts


2.1 Algorithm to calculate the size of the required bit-mask. . . . . 36

2.2 Algorithm to calculate the zero-padding required to enable the use of the most efficient DFT algorithms. . . . . 46

2.3 C structure-based double-complex type definition. . . . . 47

4.1 Example demonstrating the relevant OpenMP library routines for setting the number of parallel threads for subsequent parallel regions, querying the number of active threads and acquiring a unique thread identifier. . . . . 97

4.2 Example showing the use of some of the OpenMP compiler directives. 99

4.3 Example of '**for**' loop parallelisation using 'cyclic-plane' data distribution. . . . . 103

4.4 Example of parallel data access using 'cyclic-plane' distribution. . . . 108

5.1 Example of the grouped structure-based storage of associated control data employed within the OpenDDA framework . . . . . 142

5.2 Example demonstrating the declaration of 'static' variables on the stack and 'dynamic' variables on the heap. . . . . 155

6.1 Simple in-place local transpose for a square system, i.e., simple flip across the diagonal. . . . . 180

*"In the beginning the Universe was created. This has made a lot of people very angry and been widely regarded as a bad move."*

**Douglas Adams**

*"There is a theory which states that if ever anybody discovers exactly what the Universe is for and why it is here, it will instantly disappear and be replaced by something even more bizarre and inexplicable. There is another theory which states that this has already happened."*

**Douglas Adams**

# 1 Introduction

## 1.1 General introduction

As is inferred by Moore's Law, named after the co-founder of Intel, Gordon Moore, who originally proposed the trend in 1965, compute power tends to double every eighteen months or so. However, for obvious reasons, the associated price cannot and does not follow this trend, tending, due to competitive pressures, to decrease from year to year.

Due to the continually improving economic viability of computational hardware, access to large-scale resources is evermore present and readily available in our daily lives. As an example, at the start of this project the author had limited access to just a single medium-scale shared-memory system, which incidentally has already gone the way of the Dodo. However, by the end of the project there was practically unlimited access to more computational resources than the author knew what to do with, including multi-terabyte distributed clusters with nigh on a thousand processors and large-scale shared-memory systems with anywhere between 96 GB and 256 GB of physical memory. Furthermore, even the author's office desk now has two separate multiprocessor powerhouses, each with 16 GB of physical memory.

With such abundant access to computational resources and with the availability of convenient and user-friendly parallel Application Programming Interfaces (APIs), such as OpenMP [1] for shared-memory parallelisation and MPICH1 [2], MPICH2 [3] & Open MPI [4] for distributed-memory parallelisation, it is imperative that computational bridges be built between these resources and any and all numerical schemes

whose applicability and true potential are restricted by associated computational overheads. The aim of this project was to build such a computational bridge for a numerical scheme for calculating the electromagnetic scattering and absorption by a target of arbitrary geometry, known as the Discrete Dipole Approximation (DDA). Within the DDA, a continuum target is represented as a 3D array of point dipoles, and thus, due to its volumetric nature, the accuracy of the scheme and the size of the systems that can be simulated is wholly dependent on the number of dipoles that can be employed in the discrete representation. As the dimensions of the dipole array are increased, the computational overheads become significant and to make these simulations viable, support for large-scale parallel compute environments must be incorporated in order to take proper advantage of the different architectures that exist in the current generation of computational resources and beyond.

At the beginning of this project, the only publicly availably DDA implementation was DDSCAT [5]. DDSCAT is a Fortran-based implementation developed by Bruce T. Draine from the Princeton University Observatory and Piotr J. Flatau from the Scripps Institution of Oceanography, University of California, San Diego. Although it is a credit to its developers and has been the cornerstone of DDA simulation for more than a decade, its applicability within the realm of the modern generation of high-performance computational environments is limited by its inherent serial nature. Any attempt to adapt DDSCAT to avail of the latest computational resources (especially distributed-memory environments) would necessitate a complete rewrite of the code including its statically allocated core. Due to the large-scale nature of the code, the unfamiliarity with the subtleties of the inner workings of DDSCAT and without the inherent knowledge of the code that an original developer would possess, this endeavour would require more reverse engineering, time and effort than to create a new code. Furthermore, the finished product would probably have ended up being suboptimal. By custom building a new framework, tailored support for the latest computational environments can be combined with a myriad of new and novel optimisations to provide maximal benefit.

To ensure that all components of the new framework integrate seamlessly, without the need for unnecessary and unwanted changes in data format to comply with the requisite structure of some external package, which would only serve to subvert any and all attempts to maximise the performance and streamline the structure of the framework, the design and development was performed completely from scratch. Due to the enormity of the proposed project and the fact that all design and development

was performed by a single developer, to get the framework finished within a realistic time frame, there was no time to stop and smell the roses and a strict '*eye-on-the-prize*' policy was implemented, whereby, some intermediate results had to be taken at face value without justification.

As it was deemed nonsensical to stray off the beaten track, the structure of the new framework is loosely based on that of DDSCAT. DDSCAT is widely used, and thus, provides a worthy and useful template for the new optimised framework. Rather than trying to reinvent the proverbial wheel, it makes sense to base any new implementation along similar lines. This maximises the appeal of the new framework to the current user base that would be accustomed to the 'DDSCAT' way of doing things and also serves to ease any possible future migration.

Unfortunately, an unavoidable side-effect of the computational nature of the project is that most of the thesis reads like a technical document.

## 1.2 Discrete Dipole Approximation

### 1.2.1 Basic Theory

From a microscopic point of view, matter is comprised of atoms, which in turn are comprised of elementary charges. Consider a dielectric material, where a constituent electrically neutral atom has a nucleus with a charge of $+Ze$ and an "electron cloud" with a charge of $-Ze$, as depicted in Fig. 1.1$a$. When an external electric field is applied, the centre of mass of the "electron cloud" moves downward, see Fig. 1.1$b$, so that the distribution of charge under the influence of the externally applied electric field can be represented by an electric dipole, see Fig. 1.1$c$. A nice schematic of the electric field and lines of equipotential associated with such a dipole is shown in Fig. 1.2.

This is all well and good, however, an atom is an extremely small entity and the representation of a material as a collection of such components would be impossible. The simulation of a target can be made tractable by representing an ensemble of atoms as a small dipole. A target can be subdivided into a number of identical elements, each of which contains many atoms but is sufficiently small to be represented as a dipole. By using these simple concepts, it is possible to represent a volume element within a dielectric by a small dipole and a bulk material as an ensemble of such dipoles.

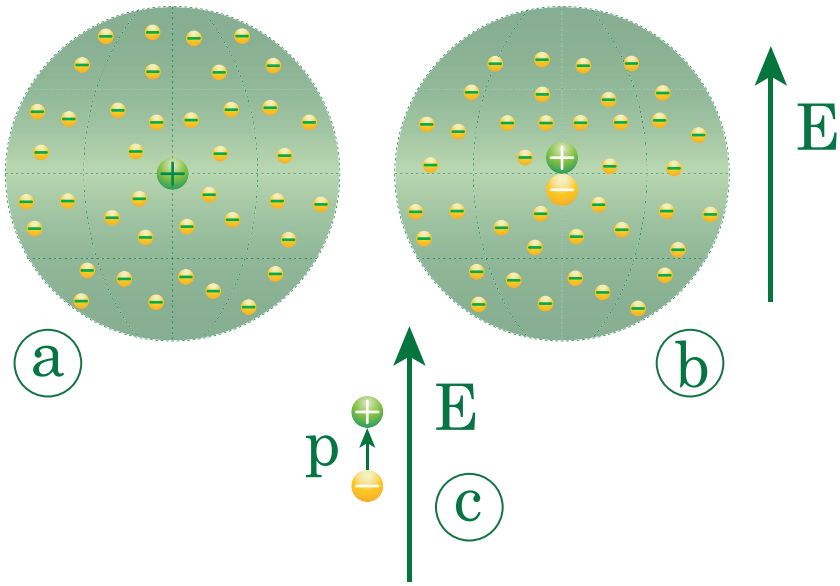



Figure 1.1: Atom in an external electric field.

The Discrete Dipole Approximation (DDA) [6, 7, 8, 9] is a numerical method for calculating the optical properties associated with a target of arbitrary geometry, whose dimensions are comparable to or moderately larger than the incident wavelength. The basic concept was first introduced in 1964 to study the optical properties of molecular aggregates [10, 11]. Subsequently, prompted by the polarisation of light by interstellar dust and the desire to calculate the approximate extinction, absorption and scattering cross sections for wavelength-sized dielectric grains of arbitrary shape, a comparable scheme was formulated [12]. Following on from this, seminal contributions and advancements were made [6, 13, 14, 15] in the development of the most widely used DDA implementation [5]. Put succinctly, the DDA is an extremely flexible and powerful numerical scheme which replaces a continuum scatterer by an ensemble of $N_d$ dipoles situated on a cubic lattice, see Fig. 1.3. Each of the $N_d$ dipoles is driven by the incident electric field and by contributions from the $N_d - 1$ other dipoles. Eq. 1.1 outlines the kernel of this scheme which is a very large and fully populated set of $3N_d$ complex-linear equations.

[4]Adapted from the Java 3D Vector Fields Applet v1.3, http://www.falstad.com/vector3de

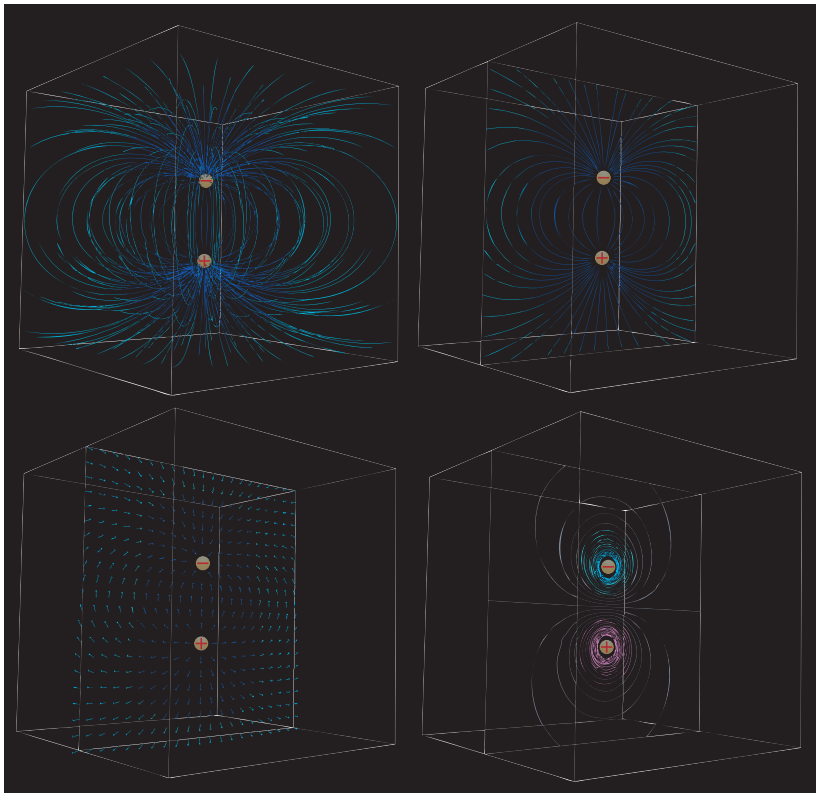


Figure 1.2: Schematic of the electric field and the lines of equipotential (panel on bottom right) of a dipole[4].

$$
\begin{aligned}
&\frac{\mathrm{k}^2}{\epsilon_0}\sum_{j=0}^{N_d-1}\mathbf{A}_{ij}\mathbf{P}_j = \mathbf{E}_{inc,i} \quad i = 0,\ldots,N_d-1 \\
&\mathbf{A}_{ii} = {\alpha_i}^{-1} \\
&\mathbf{A}_{ij} = \frac{e^{\imath \mathrm{k} r_{ij}}}{4\pi {r_{ij}}^3}\left\{\left[\frac{\imath \mathrm{k} r_{ij}-1}{\mathrm{k}^2 {r_{ij}}^2}\right]\left[3\mathbf{r}_{ij}\otimes\mathbf{r}_{ij} - {r_{ij}}^2\mathbf{I}\right] + \left[\mathbf{r}_{ij}\otimes\mathbf{r}_{ij} - {r_{ij}}^2\mathbf{I}\right]\right\}
\end{aligned}
\tag{1.1}
$$

Eq. 1.1 must be solved for the unknown polarisations, $\mathbf{P}_j$, from which the scattering properties can be calculated. Here, k is the wavenumber, $\epsilon_0$ is the permittivity of free space, $\alpha_i$ is the polarisability of dipole $i$, $\mathbf{E}_{inc,i}$ is the incident electric field at dipole $i$, $\mathbf{I}$ is the identity matrix, $\mathbf{r}_{ij} = \mathbf{r}_i - \mathbf{r}_j$, $r_{ij} = |\mathbf{r}_{ij}|$ and $\mathbf{r}_{ij}\otimes\mathbf{r}_{ij}$ is the dyadic product, see Eq. 1.2, which is a special case of the Kronecker product for vectors of equal order.

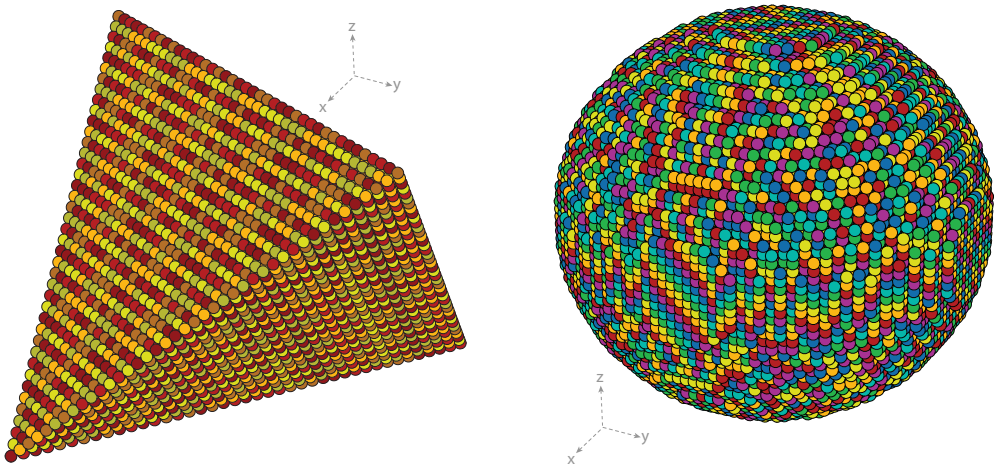



Figure 1.3: Dipole representation of a tetrahedral and a spherical scatterer, composed of $N_d = 23001$ and $N_d = 65117$ dipoles, respectively.

$$\mathbf{r}_{ij} \otimes \mathbf{r}_{ij} = \begin{pmatrix} {r_x}^2 & r_x r_y & r_x r_z \\ r_y r_x & {r_y}^2 & r_y r_z \\ r_z r_x & r_z r_y & {r_z}^2 \end{pmatrix} \tag{1.2}$$

The volumetric nature of the DDA is not only its fundamental theoretical strength but also happens to be its computational "Achilles Heel". It offers complete flexibility in regards to the description of both the spatial extent and composition of the target, which are specified on a 'dipole-by-dipole' basis, and the accuracy can be improved by simply increasing the number of dipoles in the representation of the target. However, due to the 3D nature of the computational domain, as the size of the system increases, the associated computational requirements skyrocket and for this reason the DDA is often dismissed due to the associated computational overheads.

To solve Eq. 1.1, due to the sheer size of the system, iterative $O(h[g]^2)$ methods, where $g = 3N_d$ and $h$ is the number of iterations required to reach convergence, are employed, which are much faster than direct $O(g^3)$ methods for large $N_d$ and $h \ll N_d$. The cardinal computational hurdle and key to the acceleration of these iterative methods, such as the conjugate gradient method, are the matrix-vector products required by these schemes. A major breakthrough in this respect was the realisation that by embedding the arbitrarily shaped target in a $N = K \times J \times P$ lattice and neglecting the $\mathbf{A}_{ii}$ terms, the matrix-vector products can be reformulated as

convolutions [13]. Here, $K$, $J$ & $P$ are the $\hat{\mathbf{x}}$, $\hat{\mathbf{y}}$ & $\hat{\mathbf{z}}$ dimensions of the smallest cuboid that completely encompasses the target. Once the convolutions have been performed it is then a trivial matter to include the contributions from the previously neglected diagonal terms. Consequently, Discrete Fourier Transform (DFT) techniques can be employed to calculate these matrix-vector products in $O(12N'\ln N')$ operations and storage of $O(9N')$, where $[N'] = [K' \times J' \times P'] = [2K-1] \times [2J-1] \times [2P-1]$, rather than $O(g^2)$ operations and storage that are required for explicit matrix-vector multiplication. Albeit a pivotal advancement in the solution of such systems, further optimisations are possible. The very modifications that are required in order to permit the use of these DFT techniques allow further innovation and it is these optimisations and their associated benefits that are the driving force behind many of the computational improvements that this project brings to the table.

Computationally, in terms of the description of the target, apart from the specification of the spatial and compositional properties of the target and the size of the computational domain, the user also provides a value for the so-called effective radius of the target '$a$', i.e., the radius of a sphere of equal volume. Note that in atmospheric physics, the aerodynamic radius, i.e., radius of a sphere of unit density with the same terminal velocity as the target, is often used instead of the effective radius as a characterisation parameter [16, 17]; this will be revisited in §1.2.2.2. Using these parameters and the value for the incident wavelength $\lambda$, other descriptive parameters can be derived that characterise the simulation and give a feel for the quality of the representation of the target.

$$
\begin{aligned}
size_parameter &= x \equiv \frac{2\pi a}{\lambda} \\
volume &= \frac{4}{3}\pi a^3 \equiv N_d(dipole_spacing)^3 \\
\Rightarrow dipole_spacing &= a\left(\frac{4\pi}{3N_d}\right)^{1/3} \\
\Rightarrow dipoles_per_wavelength &= \frac{\lambda}{dipole_spacing}
\end{aligned}
$$

Based on these parameters, the associated literature suggests some heuristic criteria that can be employed in order to achieve a sufficient/reasonable level of accuracy, including the use of $10 \rightarrow 20$ dipoles per wavelength [7] or employing enough dipoles in the representation of the target to ensure that $(|m|k \times dipole_spacing) < 0.5$ [5],

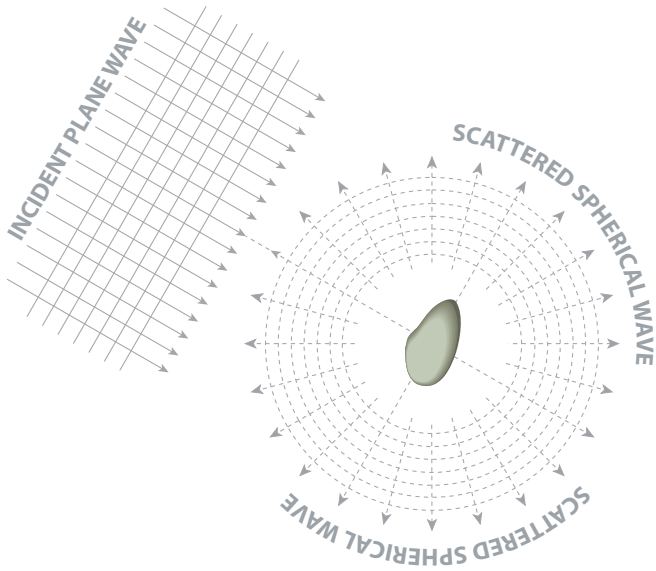



Figure 1.4: A schematic of the general scattering problem.

where $m$ is the complex refractive index of the target and $k$ is the wavenumber. The basic idea is to ensure that not only are there enough dipoles so that the structure of the target is well described, but also to keep the proverbial wool over the eyes of the incident wave by ensuring that there are enough dipoles in the representation that the incident wave sees a continuum and is oblivious to the whole discretisation scandal.

### 1.2.2 Applications

The DDA is, in principle, allowing for the consequent computational overheads, capable of treating targets of arbitrary geometry and optical properties. Due to the inherent flexibility associated with the description of the system to be simulated, the DDA can be, and has been, employed to calculate the optical properties associated with non-spherical, porous and/or inhomogeneous particles in many scientific fields. The following sections outline the typical problems and give a brief description of the various applications associated with such particles; for cosmic dust, see §1.2.2.1, for atmospheric particulates, see §1.2.2.2, for surface features, see §1.2.2.3, and for biological cells, see §1.2.2.4.

#### 1.2.2.1 Cosmic Dust

**Cosmic** dust was once solely an annoyance to astronomers, as it obscures objects they wish to observe. When infrared astronomy began, those so-called annoying dust particles were observed to be significant and vital components of astrophysical processes [18, 19]. Dust is everywhere in the Universe; it is a ubiquitous feature of the cosmos, impinging directly or indirectly on most fields of modern astronomy [20]. It occurs in a wide variety of astrophysical regions, ranging from the local environment of the Earth to distant galaxies [21, 22] and quasars [23, 24]; from meteorites originated in the asteroid belt, the most pristine solar system objects, frozen relics of a tumultuous past - comets [25, 26, 27, 28, 29, 30], circumlunar halos [31], and stratospherically collected interplanetary dust particles (IDPs) of either cometary or asteroidal origin, to external galaxies (both normal & active, nearby & distant) and circumnuclear tori around active galactic nuclei (AGN) [32, 33, 34, 35, 36]; from circumstellar envelopes around evolved stars [37] (cool red giants, asymptotic giant branch (AGB) stars [38, 39, 40]) and Wolf-Rayet (WR) stars [41, 42, 43, 44], planetary nebulae [45, 46], nova [47, 48] and supernova ejecta [49, 50, 51], and supernova remnants [52, 53, 54, 55] to interstellar clouds [56, 57] and star-forming regions [22, 58, 59, 60, 61, 62]; from the terrestrial zodiacal cloud [63, 64] to protoplanetary disks around young stellar objects [65, 66, 67, 68] and debris disks [69, 70] around main-sequence stars [20].

Cosmic dust consists of irregularly-shaped dust grains and aggregates of dust grains, with sizes ranging up to approximately 0.1 mm [18, 74]. The specific size, composition and morphology of the dust depends on the astronomical location, e.g., intergalactic, interstellar, interplanetary, circumplanetary, circumstellar, cometary, asteroidal or zodiacal dust [18]. The zodiacal cloud is a tenuous disk of both compact & highly porous, non-spherical, and inhomogeneous chondritic dust aggregates [75], with diameters between $1 \rightarrow 300\,\mu$m [76], with contributions from cometary, asteroidal, Edgeworth-Kuiper belt, and transient interstellar dust [18]. It is aligned along the plane of the ecliptic [77], orbiting within the inner part of the solar system out to about 5 AU [78], and is responsible for the zodiacal light; a faint, roughly triangular, whitish glow seen in the night sky which appears to extend up from the vicinity of the sun along the ecliptic [77].

Dust plays a fundamentally important role in the Universe; underpinning many of the physical processes that drive, shape and determine the nature of the Universe. It manifests (abstracted from [20]): (a) as an emitter, absorber, scatterer, and polariser

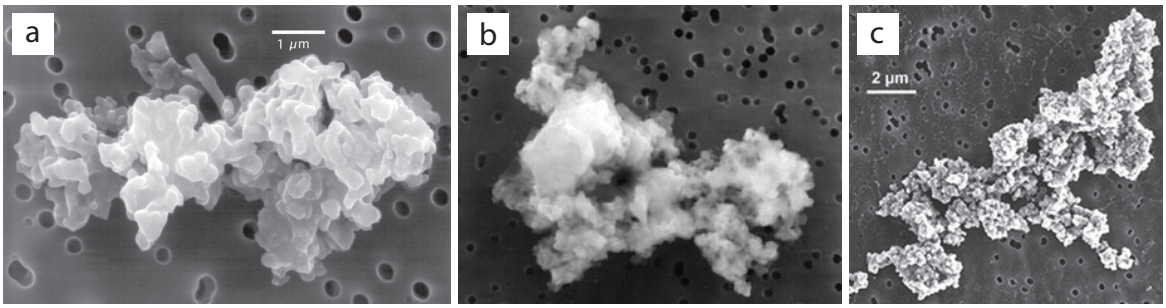



Figure 1.5: Scanning Electron Microscope (SEM) images of irregular porous chondritic interplanetary dust particles (IDPs) collected by a NASA U2 aircraft in the stratosphere. (a) 8 $\mu$m IDP, Elmar Jessberger, Institut für Planetologie, Münster, and Donald Brownlee, University of Washington [18] (b) 10 $\mu$m IDP, Donald Brownlee, University of Washington [71]. The composition and cluster-of-grapes morphology (porous aggregates built from many smaller $\sim 0.3\,\mu$m sized units) are consistent with a cometary origin [72] (c) 15 $\mu$m IDP, John Bradley, Institute for Geophysics and Planetary Physics, Lawrence Livermore National Laboratory [73].

of electromagnetic radiation, giving useful information about the dust grain sizes [18]; (b) as a revealer of heavily obscured objects, e.g., infrared sources, of which we might otherwise be unaware; infrared light can penetrate the cosmic dust clouds, allowing regions of star formation and the centres of galaxies to be probed. Most of the infrared radiation is blocked by the Earth's atmosphere and cannot be observed from the ground [18]. NASA's Spitzer Space Telescope is the largest infrared telescope ever launched into space. The Spitzer Space Telescope obtains images and spectra by detecting the infrared energy (heat) radiated by objects in space between wavelengths of $3 \rightarrow 180\,\mu$m (c) as a driver for the mass loss of evolved stars [79, 80]; (d) as a sink for heavy elements which, if otherwise in the gas phase, would profoundly affect the interstellar gas chemistry [81, 82, 83, 84]; (e) as an efficient catalyst for the formation of $H_2$ and other simple molecules, as well as complex organic molecules, and as a protector, shielding them from photodissociation by ultraviolet photons in the interstellar medium (ISM) [85]; (f) as an efficient agent for heating the interstellar gas by providing photoelectrons [86, 87]; (g) as an important coolant in dense regions by radiating infrared photons, which is particularly important for the process of star formation in dense clouds by removing the gravitational energy of collapsing clouds and allowing star formation to take place [88, 89, 90, 91]; (h) as an active participant in interstellar gas dynamics by communicating radiation pressure to the gas, and providing coupling of the magnetic field to the gas in regions of low fractional ionisation

[92, 93, 94]; (i) as a building block in the formation of stars and planetary bodies [95, 96, 97, 98]; although dust is only a minor contributors to the mass of galaxies, it determines the thermal, dynamical, and chemical state of the dense phases of the ISM, thereby controlling the star formation process. Furthermore, the coagulation of these particles in circumstellar disks is a first but decisive step towards the formation of planetesimals [99] (j) as a tool for diagnosing the physical conditions of the region where the dust is located, e.g., gas density, temperature, radiation intensity, electron density and magnetic field.

Of the order of millions of kilograms of cosmic dust descend upon the Earth each year [100, 101]. The dust particles are collected in the Earth's atmosphere using plate collectors under the wings of stratospheric-flying NASA airplanes, from surface deposits on the large Earth ice-masses (Antarctica & Arctic), and in deep-sea sediments. Dust detectors have also been incorporated into interplanetary spacecraft. In the past, dust detectors flew on the HEOS-2, Helios, Pioneer 10 & 11, Giotto, and Galileo space missions, on the Earth-orbiting LDEF, EURECA, and Gorid satellites, and some scientists have utilised the Voyager 1 & 2 spacecraft as giant Langmuir probes to directly sample the cosmic dust. Dust detectors are also part of the Ulysses, Cassini, Proba, Rosetta, Stardust, and the New Horizons spacecraft [18]. Analysis of these particles, larger meteorites and data from the space missions using probes, imagers, landers and collectors, has yielded a wealth of information about the particulate matter in the Solar System. However, it is not clear to what extent this local material resembles the interstellar dust that is scattered throughout the Galaxy. Apart from these local explorations, knowledge of the characteristics of cosmic dust is limited to deciphering the associated properties from the light that is emitted, absorbed, scattered and polarised by the dust [102].

Cosmic dust makes up approximately 1% of the mass of the ISM. Unfortunately, it scatters and absorbs strongly in the optical and ultraviolet, obscuring objects, e.g., ISM dust obscures the nucleus of the Galaxy at optical wavelengths and limits viewing to roughly a kiloparsec ($\sim 3 \times 10^{18}$ m $\equiv$ 326 LY) along the plane of the Galaxy. In addition, the distribution of dust is non-uniform and cannot be easily modelled or corrected for, except on a object-by-object basis [102], and to complicate matters even more, the dust exhibits differential or selective extinction. Not only does the dust make objects appear dimmer, leading to erroneous magnitude estimations, and hence distance estimations, but dust extinction is more pronounced at shorter wavelengths. Blue light is more readily scattered out of the line of sight than red

light is; so-called interstellar reddening, where objects viewed through dust appear somewhat redder than they should [96, 102]. This limits the ability to interpret the local and distant Universe in these observational windows, where the vast majority of the astronomical data has been obtained, and it is thus imperative that the effects of interstellar extinction be corrected for [20].

Cosmic dust grains can even ferry in organic molecules and might carry the precursors of life, seeding planetary systems with the prebiotic molecules needed for the synthesis of proteins and nucleic acids in living organisms [96, 103]. Analyses of cometary dust captured in the stratosphere together with data obtained from space missions to comets have revealed the presence of a great variety of organic molecules. Since substantial amounts of cometary dust were deposited on Earth, their organic content could have played a major role in prebiotic processes prior to the appearance of microorganisms [104, 105, 106, 107, 108].

Apart from the local knowledge of dust from the collection of IDPs and in-situ studies by space missions, e.g., the NASA Stardust mission which flew a spacecraft through the dust and gas cloud surrounding Comet Wild 2 on January 2004 [71], data regarding the physical properties of the particles is obtained from remote observations of the light scattered by the dust [109]. In order to be able to properly interpret the results of astronomical observations, and to able to correct for the effects of interstellar extinction, the ability to accurately characterise the physical properties of the particles involved is essential [110]. The DDA can treat arbitrary geometry and optical properties, however using serial-based kernels, its applicability is limited to particles with size parameters $\lesssim$ 10 to 15, a restriction that is mainly due to prohibitively large memory requirements and long computation times [111]. With cosmic dust ranging from sub-micrometre to millimetre scales, it is imperative that the limitations imposed by the overheads associated with the brute-force numerical schemes that are able to properly treat these large, non-spherical, and often highly porous, inhomogeneous and aggregated particles be addressed. The new OpenDDA framework boasts fully parallel kernels, for both shared and distributed environments, allowing multiple processors to collaborate on the solution. Furthermore, apart from the multiprocessing aspect and the memory reductions associated with *only* storing data associated with occupied sites, for the distributed computing environment, the new framework also provides a decrease in the memory required per node that is directly proportional to the number of available nodes.

#### 1.2.2.2 Atmospheric Particulates

**Atmospheric** particulates are minute particles of solid or liquid suspended in the atmosphere which can have both natural or anthropogenic sources. Naturally occurring particulate matter (PM) originates from such sources as dust storms, see Fig. 1.6 (a), wildfires (forest and grassland), see Fig. 1.6 (b), volcanoes, see Fig. 1.6 (c), living vegetation, and sea spray. For example, volcanic eruptions like Mount Pinatubo in 1991, see Fig. 1.6 (c), can lead to the formation of aerosol layers that are dispersed throughout the globe by winds and that stay in the stratosphere for years. These aerosol layers reflect sunlight, reducing the amount of energy reaching the lower atmosphere and the Earth's surface. The relative coolness of 1993 is thought to have been a response to the stratospheric aerosol layer that was produced by the Mount Pinatubo eruption [112]. Above and beyond these natural sources, human activities, such as industry, agriculture, mining, transportation, construction, habitations and deforestation, also generate significant amounts of aerosols, i.e., the burning of fossil fuels in internal combustion engines in vehicles, power plants, and various industrial processes and wind blown mineral dust from construction sites and other land areas where the water or vegetation has been removed. Averaged over the globe, it is estimated that anthropogenic aerosols currently account for approximately 10% of the total amount of aerosols in the atmosphere [113].

A fundamental concern in regards to the study of PM is the size of the individual particles. As particles are often non-spherical, there are many definitions of particle size. The most widely used definition in atmospheric physics is the aerodynamic diameter [116]. The aerodynamic behaviour of a particulate is a function of its size, shape and density. The aerodynamic diameter is defined as the diameter of a unit density ($1000\ ^{\mathrm{kg}}/_{\mathrm{m}^3}$) sphere having the same settling velocity as the particle under consideration [117], i.e., a particle with an aerodynamic diameter of 10 $\mu$m moves in a gas like a sphere of unit density with a diameter of 10 $\mu$m. Particulate matter diameters range from less than 10 nm to more than 100 $\mu$m [118]. These dimensions represent the continuum from a few molecules up to the size where particles can no longer be carried by a gas. Depending on the associated aerodynamic diameter, the various particle sizes can be subdivided into several distinct categories known as fractions [113]. The notation $\mathrm{PM}_{10}$ is used to denote the '*thoracic fraction*', i.e., particles $\leq$ 10 $\mu$m, $\mathrm{PM}_{2.5}$ is used to denote the '*respirable fraction*', i.e., particles $\leq$ 2.5 $\mu$m, and everything below 100 nm, down to the size of individual molecules is classified as ultrafine particles (UFP) [119].

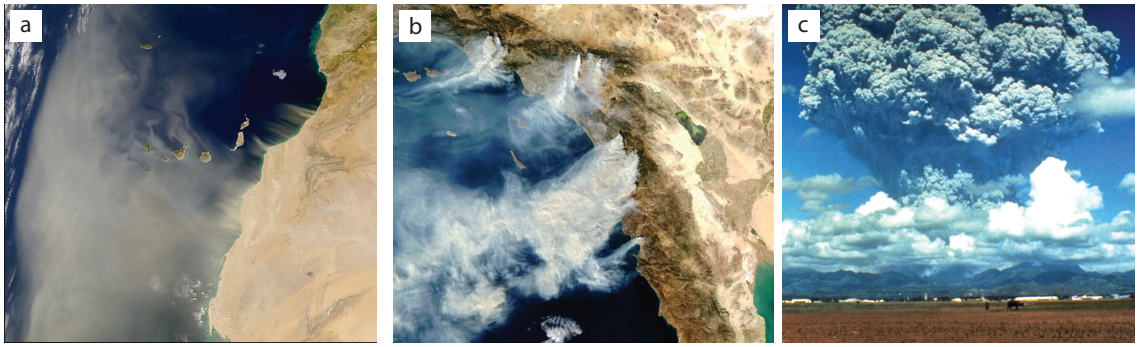



Figure 1.6: Natural sources of atmospheric particulates (a) [11$^{\text{th}}$ Feb 2001] Mineral dust from a massive dust storm over north-west Africa and the Sahara Desert; acquired by the Sea-viewing Wide Field-of-view Sensor (SeaWiFS) Project, NASA Goddard Space Flight Center [114] (b) [27$^{\text{th}}$ Oct 2003] Wildfires raging across southern California which burnt over 200,000 acres; acquired by NASA's Aqua satellite using the Moderate Resolution Imaging Spectroradiometer (MODIS), MODIS Rapid Response Team, NASA Goddard Space Flight Center [114] (c) [12$^{\text{th}}$ June 1991] An 18 km-high ash plume from the eruption of the Mount Pinatubo volcano, Philippines, viewed from Clark Air Base (about 20 km east of the volcano); acquired by Dave Harlow, U. S. Geological Survey [115].

Aerosol particulates originate from many disparate sources and consequently exhibit considerable compositional variation and size ranges. Furthermore, once formed, they may undergo subsequent modification through chemical reactions and processes such as diffusion, coagulation, sedimentation, evaporation and nucleation [120]. The chemical composition of the aerosol, which depends on its source, determines the overall refractive index and thus, directly affects how it interacts with radiation [113]. For example, wind-blown mineral dust tends to be made of mineral oxides and other material blown from the Earth's crust and is weakly absorbing [119]. Sea salt from sea spray, which does not absorb, consists mainly of sodium chloride with secondary constituents reflecting the composition of sea water, i.e., magnesium, sulphate, calcium and potassium [121]. In addition, sea spray aerosols may contain organic compounds, which influence their chemistry. Organic matter can be either primary or secondary, the latter part deriving from the oxidation of volatile organic compounds (VOCs); organic material in the atmosphere may either be biogenic or anthropogenic and influences the atmospheric radiation field by both scattering and absorption [113]. Black carbon (often called soot), a by-product of combustion processes, is highly absorbing (a heating influence) and is one of the most important absorbing aerosol species [120].

Secondary particulates derive from the oxidation of primary gases such as sulphur and nitrogen oxides, which may have either anthropogenic or natural biogenic origin, into sulphuric and nitric acid. In the presence of ammonia, secondary aerosols often take the form of ammonium salts, i.e., ammonium sulphate and ammonium nitrate; in the absence of ammonia, secondary compounds take an acidic form as sulphuric acid and nitric acid. Secondary sulphate and nitrate aerosols are strong scatterers as the presence of sulphate and nitrate causes the aerosols to increase to a size that scatters light effectively [113].

Aerosols, natural and anthropogenic, can affect the climate by changing the way radiation is transmitted through the atmosphere [113]. The climate of the Earth depends on the balance between the energy absorbed from incoming short-wavelength solar radiation and the outgoing long-wavelength radiation emitted to space [120]. The approximate balance between the solar and terrestrial energy fluxes defines the mean temperature of the system, i.e., the variable that is most widely perceived as characterising the climatic state [122]. The PM suspended in the atmosphere interacts both directly and indirectly with the Earth's radiation budget and climate, referred to as direct forcing and indirect forcing [112, 123].

In terms of incoming solar radiation, the direct effect, via albedo (the extent to which an object diffusely reflects light from the sun), whereby aerosols reflect sunlight back to space, is to cool the planet [112, 123]. The effect varies strongly geographically, with most cooling believed to be at and downwind of major industrial centres [113, 123]. However, the aerosol particles also scatter and absorb the solar radiation reflected from the Earth's surface and the outgoing terrestrial radiation. This results in the redistribution of the energy from the solar and terrestrial radiation in the atmosphere, causing atmospheric heating or cooling depending on the relevant microphysical and optical properties and on the spatial distribution of the particulates [120, 122].

The most important indirect influence of aerosol particles on the radiation balance is through their effect on clouds [122]. Aerosols in the lower atmosphere can alter the size of cloud particles, modifying the albedo and lifetime of the clouds and changing how the clouds reflect and absorb sunlight, thereby affecting the Earth's energy budget [113, 112]. Aerosols, acting as (additional) cloud condensation nuclei (CCN) and ice nuclei, strongly affect the formation, microstructure, lifetime and optical properties of water and ice clouds, and thus their radiative properties [120, 122, 123]. The concentration of CCN determines the cloud droplet concentration and the average

cloud droplet size [120]. The aerosols cause the number of cloud droplets to increase, spreading the cloud's liquid water over more, smaller, droplets. This makes the clouds more reflective, and longer lasting [112, 123]. These effects have been observed from weather satellites, whereby the pollution from the stacks of ships at sea has been seen to modify the low-lying clouds above them; leaving tracks through the cloud layers [114, 112]. Absorption by aerosol particles is always a warming influence, both at solar and terrestrial wavelengths. Backscattering of incoming solar radiation produces a cooling effect but backscattering of reflected radiation from the Earth's surface is a warming influence. The presence of clouds increases the reflectance of the atmosphere and is thus a cooling influence, however, increasing cloud layer also reduces radiation loss to space and is a warming influence [122, 124].

Aerosols also can act as sites for chemical reactions to take place. The most significant of these reactions are those that lead to the destruction of stratospheric ozone. During winter in the polar regions, aerosols grow to form polar stratospheric clouds. The large surface areas of these cloud particles provide sites for chemical reactions to take place. These reactions lead to the formation of large amounts of reactive chlorine and, ultimately, to the destruction of ozone in the stratosphere. For example, evidence now exists that shows similar changes in stratospheric ozone concentrations occur after major volcanic eruptions, like Mount Pinatubo in 1991, see Fig. 1.6 (c), where tons of volcanic aerosols are blown into the atmosphere [112].

Heavy concentration of atmospheric particulates could also lead to the hypothetical global climate condition that is predicted to be a possible outcome of a large-scale nuclear war, known as nuclear winter [125]. It is thought that severely cold weather could be caused by detonating large numbers of nuclear weapons, especially over flammable targets such as cities, where large amounts of smoke and soot would be injected into the Earth's stratosphere [126]. Similar after-effects are also thought to be possible from large-scale asteroid impacts or super-volcanic eruptions [127]. Supporting this hypothesis of severe temperature decrease from large amounts of particulates being released into the atmosphere is the burning of 526 Kuwaiti oil wells during the Persian Gulf War. This released vast amounts of PM in a geographically limited area, and directly underneath the smoke plume, constrained model calculations suggested that daytime temperature may have dropped by $\sim 10\,^{\circ}\mathrm{C}$ within $\sim 200$ km of the source [128].

In terms of the time that aerosols spend suspended in the atmosphere, it is believed that much of the removal of atmospheric aerosols occurs in the vicinity of large

weather systems and high altitude jet streams, where the stratosphere and the lower atmosphere become intertwined and exchange air with each other [112]. In general, the smaller and lighter a particle is, the longer it will stay in the air. Larger particles ($\gtrsim$ 10 $\mu$m) tend to settle to the ground by gravity in a matter of hours whereas accumulation particles ($0.1 \rightarrow 1.0\,\mu$m) can stay in the atmosphere for up to a week or more. Particles $\geq 1.0\,\mu$m are mostly removed by precipitation [113], while particles $\leq 0.1\,\mu$m are mostly removed by diffusional processes [122].

A major concern in regards to PM is the associated health effects; increasing the morbidity and mortality rate in urban dwellers [129]. The effects of inhaling PM has been widely studied in humans and animals and include altered lung function such as asthma, lung cancer, cardiovascular issues, and premature death. The size of the particle is a main determinant of where in the respiratory tract the particle will come to rest when inhaled. Larger particles are generally filtered in the nose and throat and do not cause problems, but $PM_{10}$ can settle in the bronchi and lungs and cause health problems. Similarly, $PM_{2.5}$ tends to penetrate into the gas-exchange regions of the lung, and ultrafine particles may pass through the lungs to affect other organs, including the brain [113, 121]. Particulates emitted into the atmosphere from modern diesel engines can take the form of individual particles or chain aggregates and carry highly carcinogenic and mutagenic components like benzopyrenes [130] adsorbed on their surface. Exposures have been linked with acute short-term symptoms such as headache, dizziness, light-headedness, nausea, coughing, difficult or laboured breathing, tightness of chest, and irritation of the eyes and nose and throat. Long-term exposures can lead to serious chronic health problems such as cardiovascular disease, cardiopulmonary disease, and cancer [113, 131].

The potential importance of atmospheric haze, dust and smoke to the climatic balance of our planet has been recognised for a long time [122]. However, too many uncertainties still exist to estimate accurately to what extent these effects might compensate for global warming by greenhouse gases such as carbon dioxide [129]. The warming effect of the greenhouse gases is expected to take place everywhere, but the cooling effect of the pollution aerosols will be somewhat regionally dependent, near and downwind of industrial areas [123]. In order to properly ascertain the effect of atmospheric particulates on the climatic balance, it is imperative that the fundamental interactions between these particulates and the incoming and outgoing radiation be understood. In many cases, this requires the ability to be able to model non-spherical, inhomogeneous, aggregated, and even highly absorbing particulates. The

DDA is capable of treating such particles explicitly, however, its associated computational overheads make simulations and parameter space searches laborious and often temporarily prohibitive. In order to make DDA's inherent flexibility more accessible, optimisations of the serial-based kernels and new highly-optimised parallel kernels are essential. To this end, the new OpenDDA framework, not only employs a host of novel serial-based optimisations, but also incorporates both OpenMP-based shared-memory & MPI-based distributed-memory parallelisation to efficiently administer and allocate the available computational resources and to enable multiple processors to collaborate on a single simulation.

#### 1.2.2.3 Surface Features

The efficient and accurate detection and classification of surface features is of ever-increasing importance in the fields of integrated circuit fabrication, digital storage media manufacturing, and for research on contamination of optical components. In the semiconductor industry, current state-of-the-art photolithographic techniques employ deep ultraviolet light with wavelengths of 248 nm & 193 nm [132], and as of early 2008, billion-transistor processors have become commercially available [133]. As fabrication techniques improve, the design of very-large-scale integration (VLSI) devices becomes smaller and denser. This escalates the challenges associated with the inspection of the silicon wafers and continued progress depends heavily on the evolution of wafer scanning technologies. Since circuits with smaller feature sizes are susceptible to electrical faults induced by smaller defects, defect detection technology must become increasingly more sensitive [134].

The goal of inspection is to prevent using wafers with "killer defects", i.e., those defects large enough to block, or cut, a conduction line in a device or interfere with the performance of an active component. Estimating the average feature diameter via characterisation of silicon wafer haze, i.e., light scattering from scratches, crystal defects, particulate contaminants, and surface roughness including pits and mounds, is very difficult because different features all scatter differently. The scattered light changes in intensity, direction, and polarisation as a function of feature characteristics. For a perfectly smooth and flat surface, the light reflects specularly. A contaminant deposited on the mirror-like surface disturbs the specular light, and this disturbance varies with the size, shape, composition, and orientation of the contaminant and with the surface properties; both composition and profile [136]. The background signal is associated with micro-roughness, while excursions from the background are

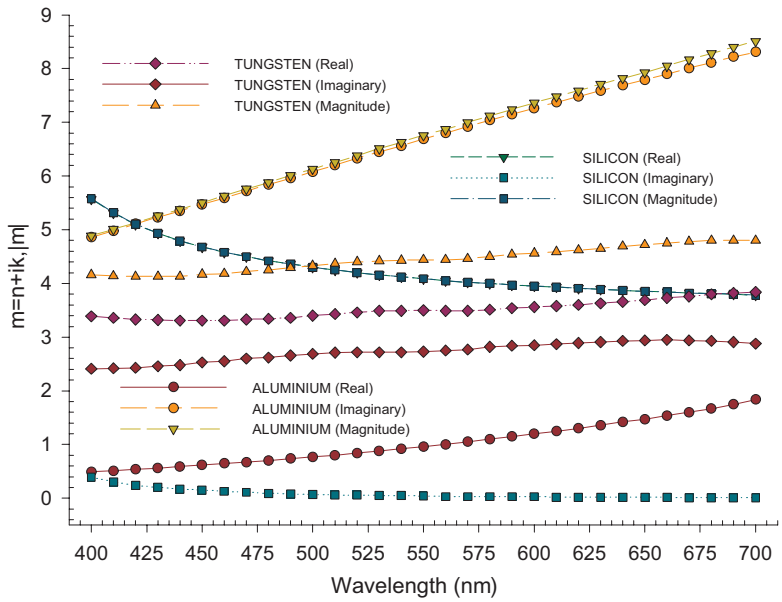



Figure 1.7: The real component, n, imaginary component, k, and magnitude, |m|, of the refractive index for aluminium (Al), silicon (Si), and tungsten (W) in the visible spectrum. Note that the data for this plot was abstracted from the 'FreeSnell - Thin-Film Optical Simulator' software package [135].

associated with localised scattering sources, such as particles or sub-surface defects [137]. To make the problem a little more difficult, each year, as line widths (the size of a printed feature printed, measured at a specific height above the substrate) are reduced, and the surface features that are considered "killer defects" get smaller, the list of critical surface features becomes larger [138]. Inspection equipment identifies many defects by registering physically detectable anomalies on the wafer surface. In-situ sensors detect potential sources of electrical faults in real time and provide important information regarding defect sourcing and generation mechanisms [134].

In order to differentiate between surface topography, defects and contaminants, and to accurately identify the size and composition of the contaminants, so that their source can be identified, it is necessary to be able to correlate the scattering from silicon wafers with properties such as surface micro-roughness, particulate contamination, and subsurface defects. The development of efficient detection systems with the ability to characterise, rather than simply detect, a defect or a contaminant particle on a surface requires accurate scattering models for ideal and non-ideal particles, and appropriate interpretation of the results from those models [139, 140].

Although the DDA was originally developed to model particles in free space, the

method can be modified to model features on surfaces by incorporating the interaction between the dipoles and the surface via the Sommerfeld integrals [139, 141, 142]. A primary concern for the modelling of surface features is the associated refractive index. In semiconductor fabrication, common circuit features and particulate contaminants are composed of materials such as silicon, tungsten and aluminium [143]. These all have refractive index magnitudes greater than 3.78 across the visible spectrum, see Fig. 1.7, with magnitudes exceeding 8.5 for aluminium at the longer wavelengths due to the high absorption coefficient. Note that the data for this plot was abstracted from the 'FreeSnell - Thin-Film Optical Simulator' software package [135]. In terms of the computational modelling of such systems, the number of dipoles that need to be employed to ensure an accurate representation of a particular target is proportional to $|\mathrm{m}|^3$ [144], where $|\mathrm{m}|$ is the absolute magnitude of the complex refractive index. This means that at an incident wavelength of $0.5\,\mu$m, for example, a target composed of silicon, $|\mathrm{m}| \approx 4.3$, would require 35 times as many dipoles as a target that is composed of ice, $|\mathrm{m}| \approx 1.31$. A target composed of tungsten, $|\mathrm{m}| \approx 4.33$, would require 36 times as many dipoles, and a target composed of aluminium, $|\mathrm{m}| \approx 6.13$, would require 102 times as many dipoles. This complication, combined with the relative size of the possible surface features, with size parameters, $x \approx 5$, produces computational requirements that are substantial and leads to significantly increased computational times [145].

#### 1.2.2.4 Biological Cells

**Biological** cells encompass a diverse range of form and composition. Their shape may be nearly spherical, discoidal or even highly irregular dendritic structures. They may also be either prokaryotic or eukaryotic, i.e., prokaryotic cells, such as bacteria, cyanobacteria and archaea, are anucleate, lacking a distinct, membrane-bound nucleus or organelles, whereas eukaryotic cells are nucleate and contain organelles [146]. An organelle is a structure or part that is enclosed within its own membrane inside a cell and has a particular function. Examples include the nucleus, the lysosome, the Golgi apparatus, the vacuole, the endoplasmic reticulum, the chloroplast and the mitochondrion [147]. Moreover, for eukaryotic cells, the nucleus itself may be non-spherical and the number of individual organelles of each type present in a cell may vary depending upon the function of that cell. Further still, cells of a particular type may often exhibit disparity is size, shape and composition,

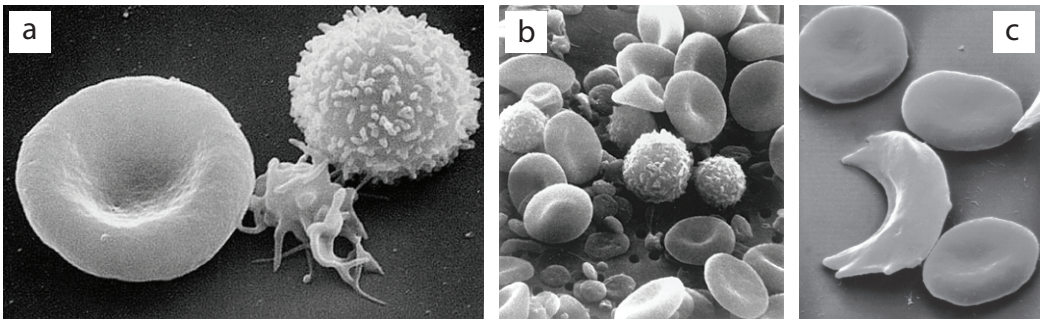



Figure 1.8: (a) From left to right: Scanning Electron Microscope (SEM) image of a human red blood cell (*erythrocyte*), a platelet (*thrombocyte*), and a white blood cell (*leukocyte*) [148] (b) SEM image of normal circulating human blood showing the inherent cell diversity [149] (c) SEM image of red blood cells that have been distorted by the genetic disorder known as Sickle-cell disease [150].

and the specific characteristics may depend on the age of the cells, leading to large morphological variability [146].

Due to the dynamic nature of biological cells, pathological effects usually manifest as changes in the cells morphology [146]. A knowledge of the theoretical light scattering behaviour of cells could enable advanced analysis, providing noninvasive, real-time in-vivo assessment of tissue pathology. For example, dysplasia, the precursor of neoplastic processes, manifests as an abnormality in cell maturation within a tissue, generally consisting of an expansion of immature cells, with a corresponding decrease in the number and location of mature cells [151]. More than 85% of all cancers originate in the epithelia lining the internal surfaces of human body. The majority of such lesions are readily treatable if diagnosed at an early stage, however, conventionally the detection of dysplasia requires the removal of tissue via biopsy which can miss dysplastic changes. Light scattering can be used in-vivo to detect subtle morphological changes in the epithelia lining the internal surfaces of the body [152]. The diagnosis is based on either heuristic models that predict changes in the scattering spectrum corresponding to morphological changes at the cellular and subcellular level, or more quantitative models used to determine the sizes of nuclei in epithelial tissue [153].

Light scattering by single cells is a well established and non-invasive method for study biological cells [154]. For example, in haematology, experimental techniques that employ the scattering from individual cells such as ektacytometry [155, 156, 146] and scanning flow cytometry [157, 158, 159, 160] are in routine use in biomedical and

| Cell Type | Diameter ($\mu$m) | MPV ($\mu m^3$) | Diameter (EVS) (D) ($\mu$m) | $D/\lambda$ ($\lambda$=0.4$\mu$m) | $D/\lambda$ ($\lambda$=0.7$\mu$m) | $x=\pi D/\lambda$ ($\lambda$=0.4$\mu$m) | $x=\pi D/\lambda$ ($\lambda$=0.7$\mu$m) |
|---|---|---|---|---|---|---|---|
| Erythrocyte [148] | 6 → 8 | 80 → 100 | 5.4 → 5.8 | 13.5 → 14.5 | 7.7 → 8.3 | 42.4 → 45.6 | 24.2 → 26.1 |

Table 1.1: Approximate size ranges, mean corpuscular volume (MPV) [163], diameter of a sphere of equal volume, and diameter-to-wavelength ratio & size parameter at both ends of the visible spectrum for an erythrocyte.

clinical environments [146].

Human blood consists of many different cell types, each with its own specialised function. Fig. 1.8 (a) shows a Scanning Electron Microscope (SEM) image of, from left to right, a human red blood cell (*erythrocyte*), a platelet (*thrombocyte*), and a white blood cell (*leukocyte*) [161], and Fig. 1.8 (b) is a SEM image of normal circulating human blood showing the inherent cell diversity [149]. White blood cells, or *leukocytes*, are cells of the immune system responsible for destroying and removing old or aberrant cells and cellular debris, as well as attacking pathogens and foreign substances. They range in size from *lymphocytes*, with diameters of 7 → 8 $\mu$m, to *macrophages*, with diameters of 14 → 17 $\mu$m [149]. Platelets, or *thrombocytes*, with diameters of just 2 → 4 $\mu$m, are cells that are involved in the cellular mechanisms of primary hemostasis leading to the formation of blood clots [162]. Red blood cells, or *erythrocytes*, with diameters of 6 → 8 $\mu$m, are cells that are responsible for the transport of oxygen from the lungs to body tissues.

In mammals mature erythrocytes are anucleate. They have nuclei during early phases of development, but extrude them as they mature in order to provide more space for haemoglobin; the iron-containing oxygen-transport metalloprotein in the red blood cells [164]. Mammalian erythrocytes also lose their other organelles, and as a result, use none of the oxygen they transport. They are biconcave disks, and this shape, combined with the loss of the nucleus and the organelles, optimises the cell for the exchange of oxygen with its surroundings [148]. Pathologically, in-vivo light scattering measurements could enable the detection of abnormal mutated erythrocytes caused by disease [160]. For example, such morphological mutations occur from the hereditary blood disorder known as Sickle-cell disease. Healthy erythrocytes are rounded and flexible so as to fit through tiny capillaries [148], however, in the case of Sickle-cell disease, an abnormal type of haemoglobin causes normal rounded and flexible erythrocytes to become rigid, curved and pointed (or sickle shaped) [165], see Fig. 1.8 (c). This inhibits their ability to pass easily through the narrow blood vessels

| | **20** dipoles/$\lambda$ | | | | **30** dipoles/$\lambda$ | | | | **50** dipoles/$\lambda$ | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | ($\lambda$=**0.4**$\mu$**m**) | | ($\lambda$=**0.7**$\mu$**m**) | | ($\lambda$=**0.4**$\mu$**m**) | | ($\lambda$=**0.7**$\mu$**m**) | | ($\lambda$=**0.4**$\mu$**m**) | | ($\lambda$=**0.7**$\mu$**m**) | |
| | **A** | **B** | **A** | **B** | **A** | **B** | **A** | **B** | **A** | **B** | **A** | **B** |
| Dipoles $\left(\times 10^6\right)$ | $\approx$17.8 | $\ll$14.0 | $\approx$3.3 | $\ll$2.6 | $\approx$60.2 | $\ll$47.3 | $\approx$11.3 | $\ll$8.8 | $\approx$278.9 | $\ll$219.0 | $\approx$52.1 | $\ll$40.9 |
| Memory (GB) | $\approx$7.2 | $\ll$4.1 | $\approx$1.3 | $\ll$0.8 | $\approx$24.3 | $\ll$13.9 | $\approx$4.5 | $\ll$2.6 | $\approx$112.1 | $\ll$64.3 | $\approx$21.1 | $\ll$12.0 |

Table 1.2: Approximate number of dipoles and memory, using *single* precision, required in the discrete representation of an erythrocyte with a diameter of 7.65 $\mu$m and a maximum thickness of 2.44 $\mu$m; values which have been used in the computational modelling of erythrocytes [166].

in the body, causing blockages that lead to poor blood flow. As a result, the organs in the body receive a limited supply of the oxygen, resulting in permanent damage and sudden, prolonged bouts of overwhelming pain, and the disorder can even cause complications such as kidney damage, increased risk of infection, thrombosis, anaemia, and stroke.

The modelling of human erythrocytes is by no means a trivial concern [154]. They are neither much smaller than the wavelength, as required by the Rayleigh Approximation, ($x \ll \lambda$), nor are they much larger than the wavelength, as required by the Geometrical Optics Approximation, ($x \gg \lambda$) [167]. Furthermore, their complex shape negates treating the erythrocytes as spheres of equal volume, which would allow analytical solution via Mie theory [168], but which would fail to take their inherent anisotropy into account [169]. Table 1.1 lists the average diameter, mean corpuscular volume (MPV) [163], diameter of a sphere of equal volume, and diameter-to-wavelength ratio & size parameter at both ends of the visible spectrum for an erythrocyte. The DDA is capable of treating particles of arbitrary geometry, however, the size of the erythrocytes imposes computational complications. Table 1.2 gives the approximate number of dipoles and memory, using single precision, required in the discrete representation of an averaged sized erythrocyte with a diameter of 7.65 $\mu$m and a maximum thickness of 2.44 $\mu$m; values which have been used in the computational modelling of erythrocytes [166], for three different accuracies, 20, 30, & 50 $^{dipoles}/\lambda$. Case '**A**' refers to a computational framework which must store the entire computational domain, e.g., DDSCAT, whereas case '**B**' refers to a computational framework that *only* stores data corresponding to *occupied* lattice sites, e.g., OpenDDA. Note that the values quoted for case '**B**' are an overestimated upper limit derived as those for a cylinder of diameter of 7.65 $\mu$m and a thickness of 2.44 $\mu$m. Ob-

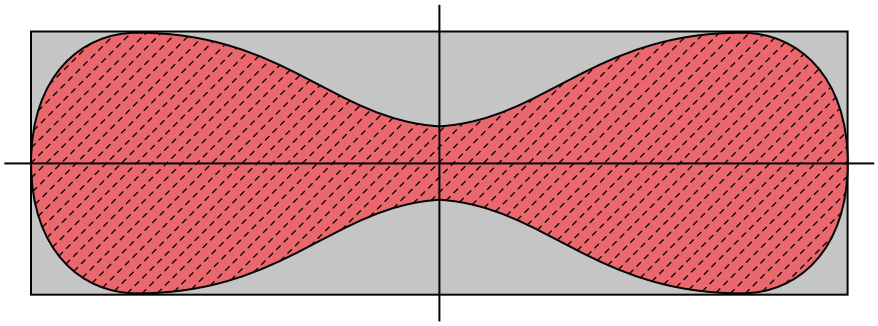


Figure 1.9: Disparity between the cross section of a cylinder and that of an erythrocyte.

viously, the actual values for case '**B**' will be significantly lower due to the disparity between a cylinder and a discoidal erythrocyte, see Fig. 1.9.

Although the dipolar requirements and associated computational overheads are beyond the limits of serial kernels, new computational frameworks, such as OpenDDA are extending the applicability of the numerical scheme to be able to treat such scientific problems.

## 1.3 Reference frames & target rotation

**Within** the OpenDDA framework, as with DDSCAT [5], there are two distinct reference frames, the laboratory frame and the target frame, where the coordinate axes are specified by the unit vectors $\hat{\mathbf{x}}_{\mathbf{L}}$, $\hat{\mathbf{y}}_{\mathbf{L}}$ & $\hat{\mathbf{z}}_{\mathbf{L}} = \hat{\mathbf{x}}_{\mathbf{L}} \times \hat{\mathbf{y}}_{\mathbf{L}}$ and $\hat{\mathbf{x}}_{\mathbf{T}}$, $\hat{\mathbf{y}}_{\mathbf{T}}$ & $\hat{\mathbf{z}}_{\mathbf{T}} = \hat{\mathbf{x}}_{\mathbf{T}} \times \hat{\mathbf{y}}_{\mathbf{T}}$, respectively. All user-defined parameters are given with respect to the laboratory frame and the target frame, which is fixed with respect to the target, for convenience, is often chosen such that, if present, $\hat{\mathbf{z}}_{\mathbf{T}}$ coincides with an axis of symmetry within the target.

For the scattering problem, as conceptualisation in Fig. 1.4, the layout of the OpenDDA framework is conventionalised such that the direction of propagation of the incident wave $\hat{\mathbf{n}}_{inc}$ coincides with the $+\hat{\mathbf{z}}$ direction in the laboratory frame. The orientation of the target frame with respect to the laboratory frame is specified via the three Euler angles, $(\phi, \theta, \psi)$. These angles constitute the '*ZYZ-moving angles of rotation*', i.e., an initial rotation $\phi$ about the $+\hat{\mathbf{z}}$ axis, followed by a rotation $\theta$ about the new $+\hat{\mathbf{y}}$ axis in the frame created by the $\phi$ rotation, and finally, a rotation $\psi$

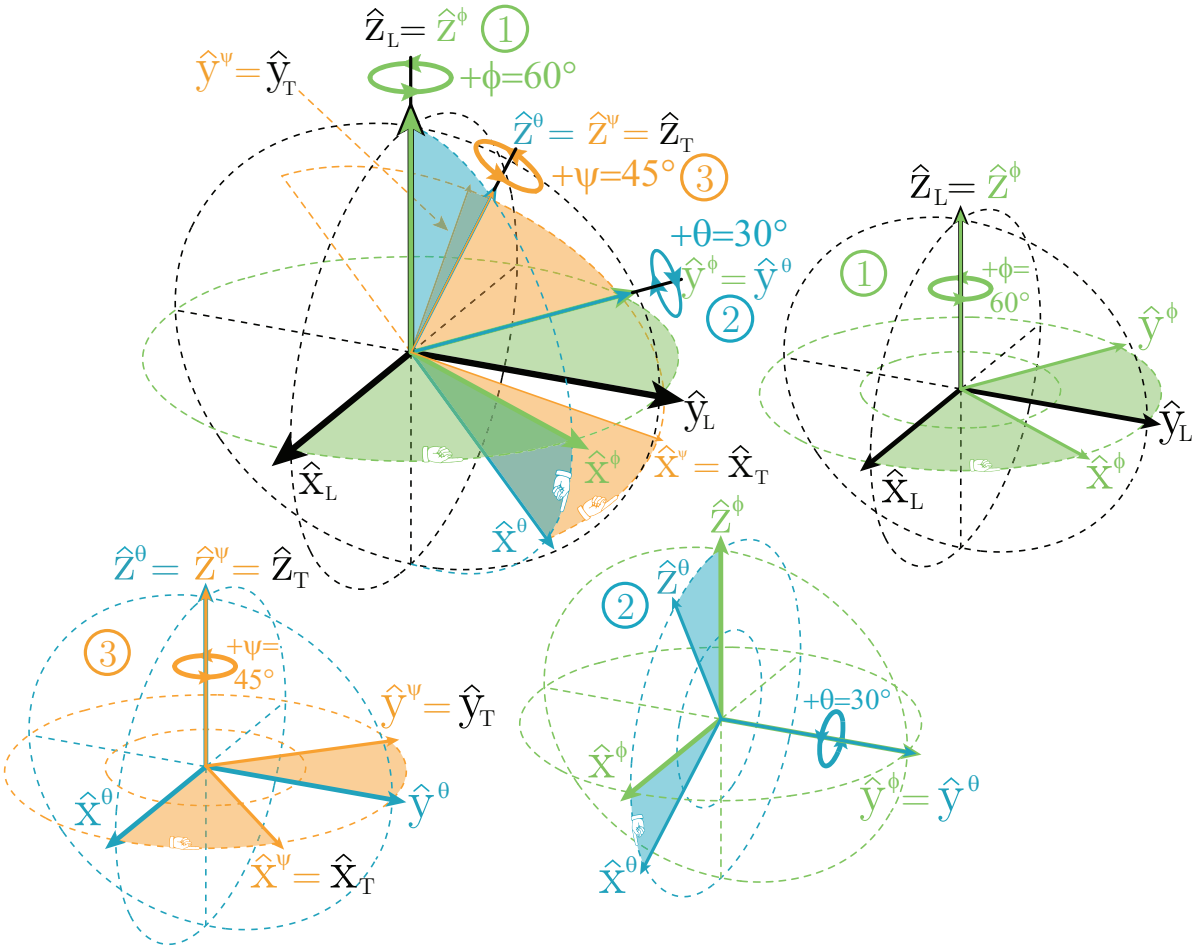



Figure 1.10: Laboratory frame→target frame transformation via the three Euler angles $(\phi, \theta, \psi)$, $\phi \in [0, 2\pi)$ about $\hat{\mathbf{z}}_\mathbf{L} = \mathbf{z}^\phi$, $\theta \in [0, \pi]$ about $\hat{\mathbf{y}}^\theta = \hat{\mathbf{y}}^\phi$, $\psi \in [0, 2\pi)$ about $\hat{\mathbf{z}}^\psi = \hat{\mathbf{z}}^\theta = \hat{\mathbf{z}}_\mathbf{T}$.

about the new $+\hat{\mathbf{z}}$ axis in the frame created by the $\theta$ rotation. The transformations between these two reference frames are given by the following rotation sequences.

- Laboratory frame→target frame $(\phi, \theta, \psi)$
  - $\phi \in [0, 2\pi)$, $\theta \in [0, \pi]$, $\psi \in [0, 2\pi)$
  - **(1)** $\phi$ about $\hat{\mathbf{z}}_\mathbf{L} = \hat{\mathbf{z}}^\phi$, see Fig. 1.10 ①
  - **(2)** $\theta$ about $\hat{\mathbf{y}}^\theta = \hat{\mathbf{y}}^\phi$, see Fig. 1.10 ②
  - **(3)** $\psi$ about $\hat{\mathbf{z}}^\psi = \hat{\mathbf{z}}^\theta = \hat{\mathbf{z}}_\mathbf{T}$, see Fig. 1.10 ③

- Target frame→laboratory frame $(-\psi, -\theta, -\phi)$
  - $-\psi \in [0, -2\pi)$, $-\theta \in [0, -\pi]$, $-\phi \in [0, -2\pi)$
  - **(1)** $-\psi$ about $\hat{\mathbf{z}}_\mathbf{T} = \hat{\mathbf{z}}^{-\psi}$, see Fig. 1.11 ①
  - **(2)** $-\theta$ about $\hat{\mathbf{y}}^{-\theta} = \hat{\mathbf{y}}^{-\psi}$, see Fig. 1.11 ②
  - **(3)** $-\phi$ about $\hat{\mathbf{z}}^{-\phi} = \hat{\mathbf{z}}^{-\theta} = \hat{\mathbf{z}}_\mathbf{L}$, see Fig. 1.11 ③

The Euler angles $\phi$ & $\psi$ are azimuthal angles, i.e., a clockwise angle $[0, 2\pi)$ in the $xy$-plane about the $+\hat{\mathbf{z}}$ axis looking in the $+\hat{\mathbf{z}}$ direction, and define longitudes,

whereas the Euler angle $\theta$ is a polar or zenith angle, i.e., the clockwise angle $[0, \pi]$ in the $xz$-plane about the $+\hat{\mathbf{y}}$ axis looking in the $+\hat{\mathbf{y}}$ direction, and defines latitudes.

Mathematically, the rotation matrices that describe these transformations as a whole, are obtained by multiplying the individual rotation matrices associated with each constituent rotation. The individual rotation matrices for each of the three rotations, governed by the three Euler angles, can be deduced from visual inspection of the rotation process. For instance, consider the laboratory frame → target frame transformation depicted in Fig. 1.10. The following mathematical snippet is all that is required to derive the three individual rotation matrices.

$\phi$ about $\mathbf{z_L}$, see Fig. 1.10 ①

$$\cos(\phi) = \frac{\mathbf{x_L}}{\mathbf{x}^\phi}, \sin(\phi) = \frac{\mathbf{y_L}}{\mathbf{x}^\phi}$$
$$\mathbf{x}^\phi \cos^2(\phi) = \mathbf{x_L} \cos(\phi)$$
$$\mathbf{x}^\phi \sin^2(\phi) = \mathbf{y_L} \sin(\phi)$$
$$\mathbf{x}^\phi = \mathbf{x_L} \cos(\phi) + \mathbf{y_L} \sin(\phi)$$
$$\mathbf{y}^\phi = -\mathbf{x_L} \sin(\phi) + \mathbf{y_L} \cos(\phi)$$
$$\mathbf{z}^\phi = \mathbf{z_L}$$

$$R(\phi) = \begin{pmatrix} \mathbf{x}^\phi \cdot \mathbf{x_L} & \mathbf{x}^\phi \cdot \mathbf{y_L} & \mathbf{x}^\phi \cdot \mathbf{z_L} \\ \mathbf{y}^\phi \cdot \mathbf{x_L} & \mathbf{y}^\phi \cdot \mathbf{y_L} & \mathbf{y}^\phi \cdot \mathbf{z_L} \\ \mathbf{z}^\phi \cdot \mathbf{x_L} & \mathbf{z}^\phi \cdot \mathbf{y_L} & \mathbf{z}^\phi \cdot \mathbf{z_L} \end{pmatrix} = \begin{pmatrix} \cos(\phi) & \sin(\phi) & 0 \\ -\sin(\phi) & \cos(\phi) & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

$\theta$ about $\mathbf{y}^\phi$, see Fig. 1.10 ②

$$\cos(\theta) = \frac{\mathbf{z}^\phi}{\mathbf{z}^\theta}, \sin(\theta) = \frac{\mathbf{x}^\phi}{\mathbf{z}^\theta}$$
$$\mathbf{z}^\theta \cos^2(\theta) = \mathbf{z}^\phi \cos(\theta)$$
$$\mathbf{z}^\theta \sin^2(\theta) = \mathbf{x}^\phi \sin(\theta)$$
$$\mathbf{z}^\theta = \mathbf{z}^\phi \cos(\theta) + \mathbf{x}^\phi \sin(\theta)$$
$$\mathbf{x}^\theta = -\mathbf{z}^\phi \sin(\theta) + \mathbf{x}^\phi \cos(\theta)$$
$$\mathbf{y}^\theta = \mathbf{y}^\phi$$

$$R(\theta) = \begin{pmatrix} \mathbf{x}^\theta \cdot \mathbf{x}^\phi & \mathbf{x}^\theta \cdot \mathbf{y}^\phi & \mathbf{x}^\theta \cdot \mathbf{z}^\phi \\ \mathbf{y}^\theta \cdot \mathbf{x}^\phi & \mathbf{y}^\theta \cdot \mathbf{y}^\phi & \mathbf{y}^\theta \cdot \mathbf{z}^\phi \\ \mathbf{z}^\theta \cdot \mathbf{x}^\phi & \mathbf{z}^\theta \cdot \mathbf{y}^\phi & \mathbf{z}^\theta \cdot \mathbf{z}^\phi \end{pmatrix} = \begin{pmatrix} \cos(\theta) & 0 & -\sin(\theta) \\ 0 & 1 & 0 \\ \sin(\theta) & 0 & \cos(\theta) \end{pmatrix}$$

$\psi$ about $\mathbf{z}^\theta$, see Fig. 1.10 ③

$$\cos(\psi) = \frac{\mathbf{x}^\theta}{\mathbf{x}^\psi}, \sin(\psi) = \frac{\mathbf{y}^\theta}{\mathbf{x}^\psi}$$
$$\mathbf{x}^\psi \cos^2(\psi) = \mathbf{x}^\theta \cos(\psi)$$
$$\mathbf{x}^\psi \sin^2(\psi) = \mathbf{y}^\theta \sin(\psi)$$
$$\mathbf{x}^\psi = \mathbf{x}^\theta \cos(\psi) + \mathbf{y}^\theta \sin(\psi) = \mathbf{x_T}$$
$$\mathbf{y}^\psi = -\mathbf{x}^\theta \sin(\psi) + \mathbf{y}^\theta \cos(\psi) = \mathbf{y_T}$$
$$\mathbf{z}^\psi = \mathbf{z}^\theta = \mathbf{z_T}$$

$$R(\psi) = \begin{pmatrix} \mathbf{x}^\psi \cdot \mathbf{x}^\theta & \mathbf{x}^\psi \cdot \mathbf{y}^\theta & \mathbf{x}^\psi \cdot \mathbf{z}^\theta \\ \mathbf{y}^\psi \cdot \mathbf{x}^\theta & \mathbf{y}^\psi \cdot \mathbf{y}^\theta & \mathbf{y}^\psi \cdot \mathbf{z}^\theta \\ \mathbf{z}^\psi \cdot \mathbf{x}^\theta & \mathbf{z}^\psi \cdot \mathbf{y}^\theta & \mathbf{z}^\psi \cdot \mathbf{z}^\theta \end{pmatrix} = \begin{pmatrix} \cos(\psi) & \sin(\psi) & 0 \\ -\sin(\psi) & \cos(\psi) & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

Although the order of the physical rotations was $\phi$ about $\hat{\mathbf{z}}_\mathbf{L}$, see Fig. 1.10 ①, $\theta$ about $\hat{\mathbf{y}}^\phi$, see Fig. 1.10 ②, and $\psi$ about $\hat{\mathbf{z}}^\theta$, see Fig. 1.10 ③, mathematically, the order is reversed, $R = R(\psi)R(\theta)R(\phi)$, as in mathematical terms, it is $R(\phi)$ that must be applied first, then $R(\theta)$, and finally $R(\psi)$.

$$R = R(\psi)R(\theta)R(\phi) = \begin{pmatrix} \cos(\phi)\cos(\theta)\cos(\psi) - \sin(\phi)\sin(\psi) & \sin(\phi)\cos(\theta)\cos(\psi) + \cos(\phi)\sin(\psi) & -\sin(\theta)\cos(\psi) \\ -\cos(\phi)\cos(\theta)\sin(\psi) - \sin(\phi)\cos(\psi) & -\sin(\phi)\cos(\theta)\sin(\psi) + \cos(\phi)\cos(\psi) & \sin(\theta)\sin(\psi) \\ \cos(\phi)\sin(\theta) & \sin(\phi)\sin(\theta) & \cos(\theta) \end{pmatrix}$$

Fortunately, the same procedure is not required to obtain the corresponding rotation matrix for the target frame → laboratory frame transformation, see Fig. 1.11. The realisation that the only differences between the two transformations is that the

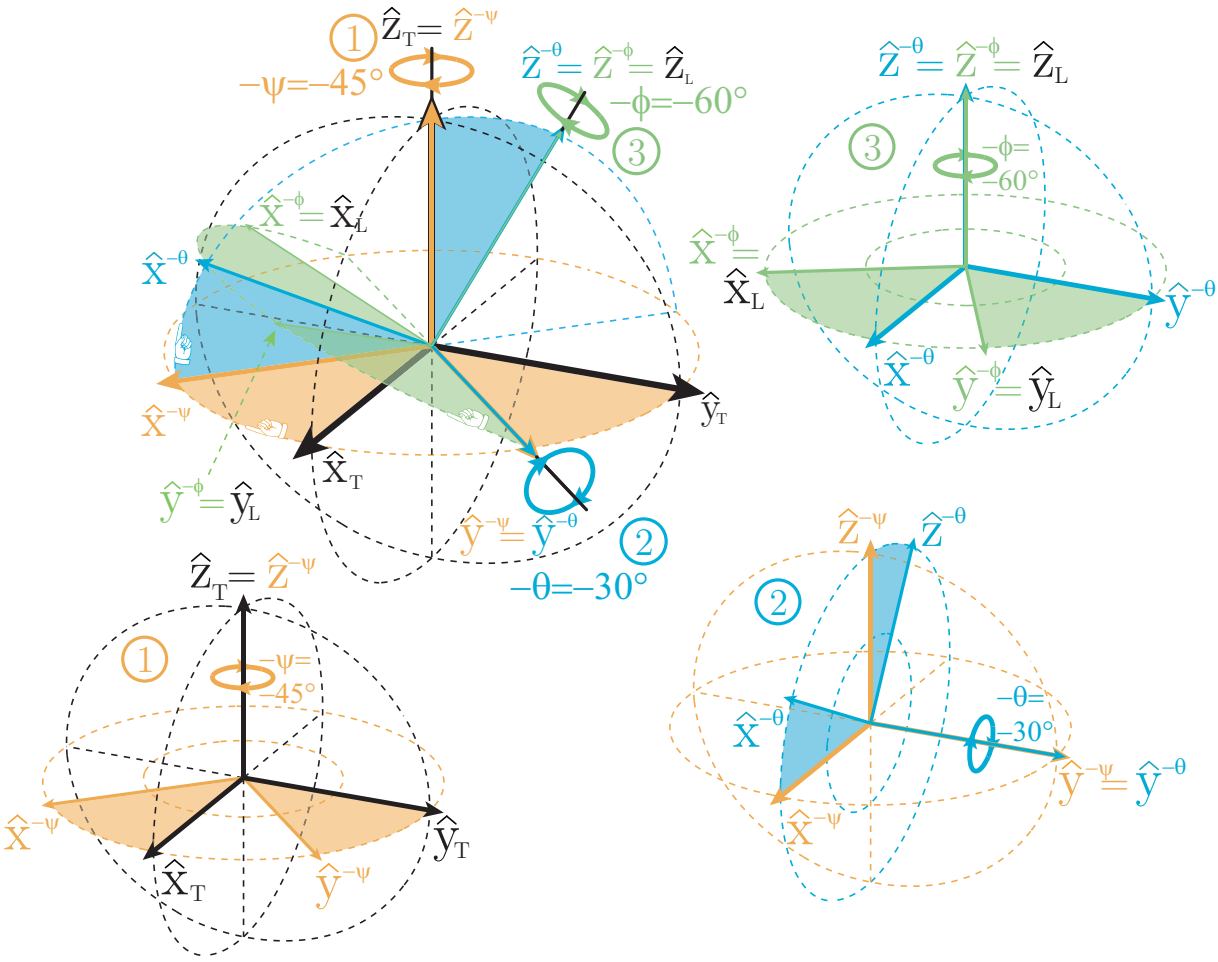



Figure 1.11: Target frame → laboratory frame transformation via the three Euler angles $(-\psi, -\theta, -\phi)$, $-\psi \in [0, -2\pi)$ about $\hat{\mathbf{z}}_\mathbf{T} = \hat{\mathbf{z}}^{-\psi}$, $-\theta \in [0, -\pi]$ about $\hat{\mathbf{y}}^{-\theta} = \hat{\mathbf{y}}^{-\psi}$, $-\phi \in [0, -2\pi)$ about $\hat{\mathbf{z}}^{-\phi} = \hat{\mathbf{z}}^{-\theta} = \hat{\mathbf{z}}_\mathbf{L}$.

signs flip and $\phi$ & $\psi$ swap places, i.e., $\phi \to -\psi$, $\theta \to -\theta$ & $\psi \to -\phi$, means that all that is required is to swap $\phi$ & $\psi$ and flip the sign of any terms containing an odd number of *sines* in the expression for $R$ above, the result of which, turns out to be $R^T$. Thus, the expressions for the coordinate transformations take on the following form.

$$\begin{pmatrix} \mathbf{x_{T(x)}} & \mathbf{x_{T(y)}} & \mathbf{x_{T(z)}} \\ \mathbf{y_{T(x)}} & \mathbf{y_{T(x)}} & \mathbf{y_{T(x)}} \\ \mathbf{z_{T(x)}} & \mathbf{z_{T(x)}} & \mathbf{z_{T(x)}} \end{pmatrix} = R \begin{pmatrix} \mathbf{x_{L(x)}} & \mathbf{x_{L(y)}} & \mathbf{x_{L(z)}} \\ \mathbf{y_{L(x)}} & \mathbf{y_{L(x)}} & \mathbf{y_{L(x)}} \\ \mathbf{z_{L(x)}} & \mathbf{z_{L(x)}} & \mathbf{z_{L(x)}} \end{pmatrix}, \quad \begin{pmatrix} \mathbf{x_{L(x)}} & \mathbf{x_{L(y)}} & \mathbf{x_{L(z)}} \\ \mathbf{y_{L(x)}} & \mathbf{y_{L(x)}} & \mathbf{y_{L(x)}} \\ \mathbf{z_{L(x)}} & \mathbf{z_{L(x)}} & \mathbf{z_{L(x)}} \end{pmatrix} = R^T \begin{pmatrix} \mathbf{x_{T(x)}} & \mathbf{x_{T(y)}} & \mathbf{x_{T(z)}} \\ \mathbf{y_{T(x)}} & \mathbf{y_{T(x)}} & \mathbf{y_{T(x)}} \\ \mathbf{z_{T(x)}} & \mathbf{z_{T(x)}} & \mathbf{z_{T(x)}} \end{pmatrix}$$

As a final remark, it is worth noting that for computational efficiency, as with DDSCAT [5], the OpenDDA framework does not actually rotate the target explicitly. Instead, the computationally lightweight equivalent is performed by leaving the target be and performing the equivalent rotation of the user-defined vectors that are associated with the laboratory frame but in the opposite direction, i.e., a positive rotation of a set of coordinate axes is equivalent to a negative rotation of any associated vectors.

## 1.4 Computational motivation

The computational requirements for systems based on 3D computational domains are extremely sensitive to the system size. Even a small increase in the dimensions of the domain significantly increases the computational overheads, and thus, there are still many numerical schemes whose true potential is encumbered by the severe and often prohibitive burdens imposed by the computational scale required to produce meaningful results. In these cases, it is crucial that every possible algorithmic or computational enhancement be exploited in order to augment their applicability.

For such systems, the available processing power is not as crucial a factor as the amount of physical memory that is available, which in the author's humble opinion, is the truly definitive limiting factor. Processing power is, for all intents and purposes, a virtual resource and is not limited in the same way that physical memory is. In terms of computation, as the job size increases, the runtime increases and excluding the imposition of 'walltimes' by job scheduling software on 'batch queue' systems, (which can be circumvented using an appropriate checkpointing scheme), larger jobs will just take longer to run. Although they may be temporarily prohibitive, which is in itself a subjective consideration, they are still computationally viable, whereas with memory, there is a distinct physical upper-bound on the job size governed by the amount of physical memory that exists.

During the development phase of the new framework, although every attempt was made to maximise the performance of each component, memory reduction was also high on the agenda and a concerted effort was made to reduce the associated memory requirements.

## 1.5 Thesis organisation

The thesis is organised as follows. Chapter 2 discusses the design and development of the custom computationally efficient and portable infrastructure of the new OpenDDA framework. It begins with a brief description in §2.2 of the bit-fielding optimisations that have been employed to streamline the storage of integer data associated with both the spatial and compositional specification of the target. The chapter also discusses the data interpolation & extrapolation routine that has been incorporated into the new framework, which depending on the number of known abscissas and the relative position of the point of interest, will employ a variety of different methods, ranging from basic linear interpolation & extrapolation to so-called improved Akima interpolation. §2.4 deals with the provision of the requisite DFT functionality that underpins the computational efficiency of the framework and §2.5 discusses the provision of portable and lightweight complex functionality, which is not native in the ANSI C89 standard. Finally, §2.6 introduces the variety of custom-built iterative solution methods that have been incorporated in the OpenDDA framework to integrate seamlessly with the custom highly-optimised DFT matrix-vector product kernel.

Chapter 3 deals with the design and development of this DFT kernel, which efficiently computes the matrix-vector products that are required by these iterative methods. §3.2 introduces the fundamental Toeplitz & circulant matrix theory on which the standard algorithm and possible optimisations are predicated, and §3.3.1, §3.3.2 & §3.3.3 show how DFTs can be used to efficiently compute matrix-vector products based on these structures for increasing levels of complexity. §3.4 briefly describes the structure of the DDA problem and §3.5 takes the theory from §3.3.3 and steps through the standard solution algorithm as applied to this problem. §3.6 is devoted to the various improvements that are possible due to the very modifications that are required in order to permit the use of DFTs in the standard algorithm.

Chapter 4 describes the extension of this theory to the parallel computing environment. §4.2 discusses the design and development of the custom OpenMP-based shared-memory implementation, including the treatment of thread safety issues. §4.3 discusses the design, development and added complexities associated with the MPI-based distributed-memory implementation of the new framework, including the custom parallel memory allocation and domain decomposition schemes in §4.3.2 & §4.3.1,

and the design considerations & creation of the custom distributed-transpose algorithms that support the distributed DFT kernel in §4.3.3→4.3.7.

Chapter 5 discusses the actual code itself, describing the general structure, versions and variants in §5.2. §5.3 describes the program control and the associated input parameters. §5.4 explains the methods for accessing the performance of the framework and for ascertaining the approximate memory requirements. Finally §5.5 steps through the prerequisites, the compilation and subsequent execution of the various serial, OpenMP-based shared-memory and MPI-based distributed-memory implementations of the OpenDDA framework.

Chapter 6 provides a whole host of serial, shared-memory & distributed-memory benchmarks for the new framework and its constituent components. The chapter begins with a quick description of the various systems that were used during the benchmarking procedure. §6.2 looks at the performance of just the new custom DFT kernel on its own, with §6.2.1 dealing with the performance of the serial implementation and §6.2.2 looking at the performance of both the parallel implementations. §6.3 reports on the performance of the serial, shared-memory & distributed-memory implementations as a whole and, to put the performance in context, §6.4 presents performance and memory usage comparisons against two other DDA implementations. The chapter finishes up with a quick verification of the operation of the new OpenDDA framework via comparison with the output of a pre-existing DDA implementation in §6.5.

The thesis closes with Chapter 7, which presents concluding remarks in §7.1 and a short discussion on possible future work in §7.2.

*"The goal of all inanimate objects is to resist man and ultimately defeat him."*

**Russell Baker**

*"To err is human, but to really foul things up requires a computer."*

**Farmers Almanac**

# 2

# Computational Infrastructure

## 2.1 Introduction

As the name suggests, this chapter deals with the creation of an efficient and portable computational infrastructure for the OpenDDA framework. It includes descriptions of the various optimisations that have been implemented and also deals with a number of miscellaneous issues concerning the provision of certain custom-built functionality, and when not possible and/or sensible, external functionality.

§2.2 introduces the notion of bit-fielding integer data, a technique used to optimise integer storage by utilising the bits within unsigned integers to store pertinent data rather than the actual datatype itself. The theory is used to optimise the storage of data pertaining to the spatial description of a target via Boolean bit-fielding and is also used to optimise the storage of non-Boolean data sets relating to the composition of the target. Although the reduction in memory provided by these optimisations has broad ramifications across all variants of the OpenDDA framework, including all three precisions, single, double & long double, and all architecture specific variants, serial, shared-memory & distributed-memory, the benefits are most pronounced for the distributed-memory environment. Bit-fielding paves the way for an efficient distributed-memory framework by allowing the spatial and compositional description of the target to be known globally without a prohibitive overhead. In this environment each autonomous node has only a limited amount of local memory and in order to permit large-scale simulations to be efficiently parallelised across the

active nodes, it is imperative that the spatial and compositional descriptions of the target are as lightweight as possible. Although these are the two main uses of bit-fielding within the OpenDDA framework, Boolean bit-fielding also finds application in the creation of the custom-built local and distributed-transpose algorithms that support the distributed-memory implementation of OpenDDA.

§2.3 describes the interpolation & extrapolation of user provided tables using a custom-built implementation of the improved Akima method [170, 171]. §2.4 discusses the provision of the requisite DFT functionality that underpins the 3D DFTs required by the optimised matrix-vector products that power the iterative methods in §2.6. §2.5 briefly mentions the custom-built complex number and associated mathematical functionality that was created to circumvent any portability issues associated with the fact that there is no native support for complex numbers in the ANSI C standard.

Finally, §2.6 discusses the custom-built iterative methods that have been incorporated to solve Eq. 1.1, including the basic underlying theory, the computational implementation and optimisation of the various algorithmic variants, preconditioning, pseudorandom number generation and vector orthonormalisation.

## 2.2 Bit-fielding data

In many applications, it is often the case that the range provided by the standard integer datatypes far exceeds the maximum range associated with the task at hand. For example, the standard 32-bit unsigned integer has a range of $0 \ldots 2^{32} - 1 = 0 \ldots 4294967295$. There are countless instances where only a fraction of this range is required, and as a result, most of the 32 bits are superfluous and constitute a serious waste of resources. Bit-fielding is a way to overcome this issue by storing data using the bits within the integers rather than datatypes themselves. In many circumstances, the full range can be specified with just a few bits, or for the limiting case, if the data consists of Boolean 0/1 flags, only single bits are required, i.e., on or off, there or not there, finished or still to do, etc. The entries can be set, tested and extracted by taking the appropriately bit-shifted bit-mask and applying the relevant bitwise operator, LOGICAL OR for setting and LOGICAL AND for testing/extracting, i.e., '|' and '&' in C notation. The simplicity and speed of bitwise operations makes bit-fielding an extremely efficient method of storing non-negative integer based data.

Habitually, bit-fielding has been limited to the compression of Boolean flags, however, this need not be the case. Boolean bit-fielding is just a special case where the bit-mask is a single bit and there is no reason why wider bit-masks cannot be employed to extend the scope beyond Boolean data to the bit-fielding of broader data sets. In terms of the DDA, there are several places where bit-fielding is employed to streamline the storage of both Boolean and non-Boolean data sets. The Boolean implementation is used to compress the occupation flags which indicate whether or not a lattice site is occupied by a dipole or not and is also used within the kernel of the MPI version to support the local and distributed transposes, this will be discussed later in §4.3.5. On the other hand, the extension of the method to non-Boolean data sets is employed to compress the data associated with the compositional description of the target.

### 2.2.1 Bit-fielding of the target occupation data

**Within** the DDA, the spatial extent of the target can be specified by storing the 3D coordinates of the computational domain and, for each of the constituent sites, storing a Boolean flag to indicate whether or not the site is occupied by a dipole. This method requires four integers per lattice site, three for the $(i, j, k)$ coordinate and one occupation flag.

When dealing with large systems, the storage associated with this integer data can become appreciable and even detrimental to the overall functionality of the application. For instance, consider a large system with $K = J = P = 500$. Regardless of the shape of the specific target within this computational domain, the domain still consists of $N = K \times J \times P = 125 \times 10^6$ individual sites. With four integers per site and four bytes per integer, this requires $(4 \times N) \times 4\,\mathrm{B} = 2 \times 10^9\,\mathrm{B} \approx 1.863\,\mathrm{GB}$ just to store the data associated with the physical layout of the target. It is conceded that such a system would only be viable in a large parallel environment, and that, when employing large-scale shared-memory machines with orders of magnitude more physical memory than is currently available in desktop workstations, integer storage of this magnitude may be acceptable. However, it is hardly ideal, and in any case, the main driving force behind the possible optimisation of these storage requirements is the creation of a versatile and efficient distributed implementation of the DDA. For a shared-memory system, the storage of this data does not pose a significant problem as with a single address space that is accessible by all processors, only one

instance of the data is required. However, for a distributed-memory system, each autonomous node must have a copy of the data and with just its own local memory to work with, typically a few gigabytes, the storage requirements for just the target layout are comparable to the maximum possible local allocation per node, negating the possibility of a useful distributed implementation. Thus, although any and all optimisations of the storage requirements for the target layout have significant ramifications for all serial and parallel versions of the algorithm, the optimisations were originally designed with the distributed parallel environment in mind.

The first obvious optimisation is to apply the Boolean bit-fielding method to compress the $N$ Boolean occupation flags associated with the computational domain. There are 32 bits per integer, and thus, $ctrl = (int)\left(\frac{N-1}{32}+1\right)$ integers are required to store the $N$ Boolean flags, $index = 0 \ldots N-1$. The '$-1$' in the formula simply ensures that the formula does not overestimate the requirement when $N$ is a multiple of 32. The flags are stored within an array of $ctrl$ separate host integers, called *occupied*. In order to be able to set or test a Boolean flag that corresponds to a particular lattice site, $index$, one must know in which of these integers the relevant bit is and indeed, which bit is it in that integer. These can be extracted from the value of $index$ via integer truncation and the *modulo* function. The Boolean flag corresponding to $index$ is located in integer $(int)(\frac{index}{32})$ and is bit $(index\%32)$ within that integer. If this flag is set then the corresponding lattice site is occupied and is vacant otherwise. To mark a lattice site as occupied, a bit-mask of '1' is shifted up by $(index\%32)$ and bitwise OR is used to set the bit in integer $(int)(\frac{index}{32})$, i.e.,

$$occupied\left[(int)\left(\frac{index}{32}\right)\right] | = (1 \ll (index\%32)),$$

where, in C notation, '$\ll$' is the left or up bit-shift operator. In a similar fashion, to check if a specific site is occupied or not, bitwise AND is used to extract the Boolean flag from the appropriate integer, i.e.,

$$occupied\left[(int)\left(\frac{index}{32}\right)\right] \&(1 \ll (index\%32)).$$

Above and beyond the optimisation of the storage of the Boolean occupation flags, further reductions can be achieved by the realisation that the $(i, j, k)$ coordinates of the computational domain do not have to be stored at all. These coordinates are based on a computational grid of known dimensions, $(K, J, P)$, and as such, if required, are readily available via several mechanisms. Firstly, for efficiency and

performance purposes, all data arrays within the new code are stored in 1D format and pertinent data is accessed using 3D loop structures that step through the 1D arrays, i.e., $index = kM+jK+i = 0\ldots N-1$, $M = K\times J$, $i = 0\ldots K-1$, $j = 0\ldots J-1$, $k = 0\ldots P-1$. This means that the $(i, j, k)$ coordinates are dynamically available within the execution loops themselves as the loop control variables. Secondly, even if this was not the case, for scenarios where 1D control loop are preferential and the $(i, j, k)$ coordinates are still required, the coordinates can be extracted directly from the array index itself.

$$\begin{aligned} index &= kM + jK + i = 0\ldots N-1 \\ k &= (int)\left(\frac{index}{M}\right), j = (int)\left(\frac{index - kM}{K}\right), i = index - kM - jK \\ k &= 0\ldots P-1,\ j = 0\ldots J-1,\ i = 0\ldots K-1 \end{aligned}$$

Combining both the bit-fielding optimisation of the Boolean occupation flags and the memory savings provided by not storing the $(i, j, k)$ coordinates of the computational grid yields a vast improvement in the memory required to describe the physical layout of the target. For the $K = J = P = 500$, $N = K\times J\times P = 125\times 10^6$ system discussed earlier, the memory requirements are reduced from $(4\times N)\times 4\text{B} = 2\times 10^9\,\text{B} \approx 1.863$ GB to just,

$$ctrl \times 4\,\text{B} = (int)\left(\frac{N-1}{32} + 1\right) \times 4\,\text{B} \approx 15.625 \times 10^6\ \text{B} \approx 14.9\,\text{MB},$$

which is a factor of approximately 128. Not only does this provide a significant reduction in terms of the serial implementations but, more importantly, makes a distributed MPI-based implementation much more viable.

### 2.2.2 Dynamic bit-fielding of the target composition data

**Another** integer-based data requirement is the data associated with the physical composition on the target. The DDA's theoretical strength, which is also its computational weakness due to imposed computational overheads, is its volumetric nature. Its ability to treat arbitrarily shaped particles with complex internal structures derives from the fact that each constituent site in the computational domain has separate integer flags describing its properties. The site can be occupied by a dipole or left vacant simply by setting a single Boolean flag, and if a dipole is present, the material properties of the target are specified,

on a dipole-by-dipole basis, by attributing complex refractive index values to each individual dipole. This is done by associating each of the different complex refractive index values with a distinct integer tag, and for each dipole, storing three such integer tags to define the associated complex refractive index in the $\hat{\mathbf{x}}$, $\hat{\mathbf{y}}$ & $\hat{\mathbf{z}}$ directions.

As an example, consider the $K = J = P = 500$, $N = K \times J \times P = 125 \times 10^6$ system mentioned in the previous section. If this system has more than one constituent material, the storage associated with just the integer flags to indicate each dipole's specific material properties amalgamates to a memory requirement of $(3 \times N) \times 4\text{B} = 1.5 \times 10^9\text{B} \approx 1.397\text{GB}$. This is of the order of the original memory requirement for the target occupation data and like that, in terms of the creation of a feasible distributed implementation, is prohibitively large.

However, it is not all doom and gloom. As was mentioned in the introduction to the bit-fielding section, by employing a wider bit-mask, the Boolean bit-fielding method can be extended to treat broader integer-based data sets. To store the $\hat{\mathbf{x}}$, $\hat{\mathbf{y}}$ & $\hat{\mathbf{z}}$ material data, separate bit-fielded arrays are constructed for the three separate Cartesian directions. The first task to extend the method to non-Boolean values is to calculate the size of the bit-mask required. To help with the documentation of the method, consider a scenario where the aforementioned $K = J = P = 500$ system has four distinct complex refractive indices associated with it, i.e., four disparate materials denoted by the associated integer tags, $0, 1, 2$ & $3$, with the actual specifics of the complex refractive index values being irrelevant. In binary, 2 bits are all that is required to encompass this decimal range, $0_{dec} = 00_{bin}, 1_{dec} = 01_{bin}, 2_{dec} = 10_{bin}$ & $3_{dec} = 11_{bin}$, and the required bit-mask, as given by Excerpt 2.1, is $3_{dec} = 11_{bin}$.

Code Excerpt 2.1: Algorithm to calculate the size of the required bit-mask.

```
/* Calculate the size of the bit-mask required */
mask_size=1;
while((1<<mask_size)<number_of_materials){
    mask_size++;
}
mask=(1<<mask_size)-1; /* Actual bit-mask */
```

The next task is to find out how many entries can be stored per integer. With 32 bits per integer and $mask_size$ bits per bit-mask, the number of masks per integer is given by $masks_per_integer = (int)(\frac{32}{mask_size})$. An important note is that the value of $mask_size$ need not be a factor of 32, i.e., if there were eight materials, $[0_{dec}, 1_{dec}, \ldots, 7_{dec}] = [000_{bin}, 000_{bin}, \ldots, 111_{bin}]$, the corresponding bit-mask would be $7_{dec} = 111_{bin}$, with $mask_size = 3$, and 10 entries could be stored per integer, with 2 bits leftover. For a given direction, $\hat{\mathbf{x}}$, $\hat{\mathbf{y}}$ or $\hat{\mathbf{z}}$, the number of integers required to store the $N$, $index = 0 \ldots N-1$, material integer flags is given by a similar expres-

sion to that for the Boolean flags, except instead of 32 flags per integer, there are $masks_per_integer$ entries per integer, and thus, $ctrl = (int)\left(\frac{N-1}{masks_per_integer} + 1\right)$. Of course, if there are only two constituent materials, the extended formalism simplifies to the prescribed basic Boolean case.

The entries are stored within three separate arrays, each containing $ctrl$ host integers, called $composition[vc]$, $vc = 0 \ldots 2$, i.e., one for each of the three Cartesian directions. To set and extract these data entries, as with the Boolean case, which integer the data is in and which data entry it is, must be known. As with the Boolean case, for a particular lattice site, $index$, these values can be extracted from the value of the $index$ using integer truncation and the $modulo$ function. The data entry corresponding to $index$ is located in integer $(int)\left(\frac{index}{masks_per_integer}\right)$ and is data entry $shift = (index\%masks_per_integer) \times mask_size$ within that integer. To set the $\hat{\mathbf{x}}$, $\hat{\mathbf{y}}$ & $\hat{\mathbf{z}}$ data entries for the lattice site corresponding to $index$, the values $value_x, value_y$ & $value_z$ are shifted up by the value of $shift$ and bitwise OR is used to set the data entry in integer $(int)\left(\frac{index}{masks_per_integer}\right)$, i.e.,

$$
\begin{aligned}
composition[0]\left[(int)\left(\frac{index}{masks_per_integer}\right)\right] \mid &= (value_x \ll shift), \\
composition[1]\left[(int)\left(\frac{index}{masks_per_integer}\right)\right] \mid &= (value_y \ll shift), \\
composition[2]\left[(int)\left(\frac{index}{masks_per_integer}\right)\right] \mid &= (value_z \ll shift).
\end{aligned}
$$

In a similar fashion, to extract the data entries for the lattice site corresponding to $index$, the host integer itself, is shifted down by the value of $shift$ and bitwise AND is used to extract the $value_x, value_y$ & $value_z$ values, i.e.,

$$
\begin{aligned}
value_x &= \left(composition[0]\left[(int)\left(\frac{index}{masks_per_integer}\right)\right] \gg shift\right) \&mask, \\
value_y &= \left(composition[1]\left[(int)\left(\frac{index}{masks_per_integer}\right)\right] \gg shift\right) \&mask, \\
value_z &= \left(composition[2]\left[(int)\left(\frac{index}{masks_per_integer}\right)\right] \gg shift\right) \&mask,
\end{aligned}
$$

where, in C notation, '$\gg$' is the right or down bit-shift operator. This non-Boolean bit-fielding of the material integer data reduces the memory required for the integer arrays by a factor equal to number of data entries that can be packed into an integer. For the illustrative example, i.e., a $K = J = P = 500$ system with four associated ma-

terial properties, using the non-Boolean bit-fielding, only 2 bits are required in order to store all possible material possibilities, and as a result, the number of data entries per integer has been increased from 1 to 16. Please note that although only four materials were used in this example, bit-fielding does not impose any restrictions on the number of distinct materials and there is no limitation on the number that can be employed within the OpenDDA framework. This reduces the memory requirements by a factor of 16, from $(3 \times N) \times 4\,\mathrm{B} = 1.5 \times 10^9\,\mathrm{B} \approx 1.397\,\mathrm{GB}$ to,

$$3 \times ctrl \times 4\,\mathrm{B} = 3 \times (int) \left( \frac{N-1}{masks_per_integer} + 1 \right) \times 4\,\mathrm{B} \approx 93.75 \times 10^6\,\mathrm{B} \approx 89.41\,\mathrm{MB}.$$

Coupled with the optimisations of the target occupation data from the previous section, for the given example, the overall memory requirements, for the integer-based data pertaining to the physical description of the target, has been reduced from approximately 3.26 GB to just over 0.1 GB.

## 2.3 Data interpolation & extrapolation

For many applications, it is often the case that only a limited and frequently inadequate data set exists, or is available, and in these scenarios, in lieu of a comprehensive set, efficient interpolation & extrapolation of this data set can provide a viable substitute. Interpolation involves the estimation of intermediate values from a discrete set of known abscissas, $x_0 \ldots x_{n-1}$, often referred to as nodes, and their corresponding ordinates, $y_0 \ldots y_{n-1}$, where, $x_i < x_{i+1}$, $i = 0 \ldots n-2$. If the point of interest happens to lie outside the known range, then the process is termed extrapolation.

For maximum flexibility, the user interface of OpenDDA allows for the provision of several user-defined input files, see §5.3. As part of the target specification, for each constituent material, the user can explicitly set the relevant complex refractive index value for the particular incident wavelength in the input parameter file '*control.input*'. Alternatively, for any of the materials, the user can choose to supply a file containing a table of complex refractive index versus wavelength ($\mu$m). Apart from the necessary syntax, the only restriction for this data is that the wavelength values must be positive and monotonically increasing. This is all well and good, however, the complication now arises that it is conceivable that an entry corresponding to the particular wavelength of interest might not appear in the supplied table explicitly.

To remedy this problem OpenDDA includes both interpolation and extrapolation functionality. Please note that, although extrapolation is possible, the results of extrapolation are often conjectural at best, and it should be used only when prudent and with extreme caution. The algorithm employed to implement the interpolation or extrapolation is wholly dependent on the number of entries in the given data set and on whether or not the point of interest happens to lie within the interpolation regime, i.e., within the interval delimited by $x_0$ and $x_{n-1}$.

For $n < 5$ and $x$ within the interval, the algorithm employs low degree polynomial interpolation. Polynomial interpolation can be succinctly described using the so-called Lagrange polynomials [172], $L_k^{(n)}(x)$, and essentially boils down to the fact that through any $n+1$ points, there is a unique polynomial of at most $n^{th}$ degree, i.e., there is a unique line through any two points, a unique quadratic through any three points, a unique cubic through any four points, etc. For $n = 2, 3$ & $4$, the interpolation polynomials are linear, quadratic and cubic, respectively.

$$y = \sum_{k=0}^{n} y_k L_k^{(n)}(x)\,,\ L_k^{(n)}(x) = \prod_{\substack{j=0\\ j\neq k}}^{n} \left(\frac{x - x_j}{x_k - x_j}\right) [172]$$

$$a = \left(\frac{x-x_0}{x_1-x_0}\right), b = \left(\frac{x-x_1}{x_2-x_1}\right), c = \left(\frac{x-x_2}{x_0-x_2}\right), d = \left(\frac{x-x_3}{x_1-x_3}\right), e = \left(\frac{x-x_3}{x_0-x_3}\right), f = \left(\frac{x-x_3}{x_2-x_3}\right)$$

$$\mathbf{n=1}, \textsc{linear},\ [(x_0, y_0), (x_1, y_1)], y = y_0(1-a) + y_1 a \tag{2.1}$$

$$\mathbf{n=2}, \textsc{quadratic},\ [(x_0, y_0), (x_1, y_1), (x_2, y_2)], y = y_0(1-a)c + y_1(1-b)a + y_2(1-c)b \tag{2.2}$$

$$\mathbf{n=3}, \textsc{cubic},\ [(x_0, y_0), (x_1, y_1), (x_2, y_2), (x_3, y_3)],$$
$$y = y_0(1-a)ce + y_1(1-b)ad + y_2(1-c)bf + y_3(1-d)(1-e)(1-f) \tag{2.3}$$

Polynomial interpolation is not without its disadvantages. First of all, the addition, removal or modification of a point can have significant impact on the overall form of the interpolant. Furthermore, for high degree polynomials, polynomial interpolation is susceptible to Runge's phenomenon [173], whereby equidistant nodes cause the interpolant to exhibit large spurious oscillations in the form of divergence of the interpolant near the ends of the interval, that worsens as the number of nodes is increased. This effect can be overcome by concentrating the nodes towards the ends of the interval using the Chebyshev distribution [172], where the $n$ Chebyshev nodes are given by $x_i = cos\left(\frac{2i-1}{2n}\pi\right)$, $i = 1 \ldots n$. However, tabulated data, be it sampled, empirical or whatever, is predominantly equispaced and such control over the abscissas is not possible.

If $x$ happens to lie outside the interval, the algorithm employs linear extrapolation,

i.e., creates a tangent line at the end of the known data. For $n=2$, this is identical to the interpolation case, see Eq. 2.1, while for higher values of $n$, the algorithm employs slightly more involved estimations of the slope. The following gives expressions for the first derivatives of a quadratic [170] and cubic [171] polynomial, fitted to the points $p_i, p_j, p_k$ and $p_i, p_j, p_k, p_l$, respectively, at point $p_i$, where $i$ must be the first index and the order of the remaining indices can be arbitrary.

$$
\begin{aligned}
& y = y_i + m\,(x - x_i) \\
& \mathbf{n = 3},\ [(x_i, y_i)\,, (x_j, y_j)\,, (x_k, y_k)]\,,\ m(i,j,k) = \frac{f_1 + f_2}{f_3 + f_4} \\
& f_1 = (y_j - y_i)(x_i - x_k)^2,\ f_2 = (y_i - y_k)(x_j - x_i)^2 \\
& f_3 = (x_j - x_i)(x_i - x_k)^2,\ f_4 = (x_i - x_k)(x_j - x_i)^2 \\
& \mathbf{n \geq 4},\ [(x_i, y_i)\,, (x_j, y_j)\,, (x_k, y_k)\,, (x_l, y_l)]\,,\ m(i,j,k,l) = \frac{f_1 + f_2 + f_3}{f_4} \\
& f_1 = (y_j - y_i)(x_k - x_i)^2 (x_l - x_i)^2 (x_l - x_k) \\
& f_2 = (y_k - y_i)(x_l - x_i)^2 (x_j - x_i)^2 (x_j - x_l) \\
& f_3 = (y_l - y_i)(x_j - x_i)^2 (x_k - x_i)^2 (x_k - x_j) \\
& f_4 = (x_j - x_i)(x_k - x_i)(x_l - x_i)(x_k - x_j)(x_l - x_k)(x_l - x_j)
\end{aligned}
\tag{2.4}
$$

For $n > 4$ and $x$ within the interval, the algorithm employs what is known as piecewise interpolation. Piecewise interpolation overcomes the problems associated with high-degree polynomial interpolation by stitching together a series of low-degree polynomials. This is done by matching the first and second derivatives of the end-points of adjacent curves, yielding a smooth curve called a spline. The specific method used is known as improved Akima univariate interpolation, named after the developer, Hiroshi Akima, who first introduced the concept in 1969 [174, 170], and then 21 years later, improved upon the original theory [171].

The local nature of the method ensures that, as with manual curve fitting, the shape of the interpolant is only due to points that are confined to a small region, and hence, produces a smooth natural looking curve that resembles a hand drawn curve. The algorithm locates the subinterval that includes the point of interest using a binary search. Once the interval $[x_i, x_{i+1}]$ has been identified, for a given $x$, the algorithm calculates the corresponding ordinate $y$ using a cubic polynomial.

$$
\begin{aligned}
& y = a_0 + a_1\,(x - x_i) + a_2\,(x - x_i)^2 + a_3\,(x - x_i)^3\,,\ a_0 = y_i,\ a_1 = g_i \\
& a_2 = \frac{-2\,(g_i - m_i) - (g_{i+1} - m_i)}{(x_{i+1} - x_i)},\ a_3 = \frac{(g_i - m_i) + (g_{i+1} - m_i)}{(x_{i+1} - x_i)^2},\ m_i = \frac{(y_{i+1} - y_i)}{(x_{i+1} - x_i)}
\end{aligned}
\tag{2.5}
$$

Here, $m_i$ is the slope of the line connecting the endpoints of the interval and $g_i$ & $g_{i+1}$ are estimates for the first derivative of the interpolant at these endpoints. The key to the success of this method is the estimation of these derivatives. For each endpoint, $p_i = (x_i, y_i)$, the algorithm uses the seven points, $p_{i-3}, p_{i-2}, p_{i-1}, p_i, p_{i+1}, p_{i+2}, p_{i+3}$, to construct four primary estimates, $u_j$, $j = 0 \ldots 3$, for the first derivative at point $p_i$, of a cubic polynomial fitted to the four consecutive data sets that contain $p_i$, using Eq. 2.4.

$$\begin{aligned}
[p_{i-3}, p_{i-2}, p_{i-1}, p_i] \rightarrow \quad & u_0 = m\,(i, i-3, i-2, i-1) \\
[p_{i-2}, p_{i-1}, p_i, p_{i+1}] \rightarrow \quad & u_1 = m\,(i, i+1, i-2, i-1) \\
[p_{i-1}, p_i, p_{i+1}, p_{i+2}] \rightarrow \quad & u_2 = m\,(i, i+1, i+2, i-1) \\
[p_i, p_{i+1}, p_{i+2}, p_{i+3}] \rightarrow \quad & u_3 = m\,(i, i+1, i+2, i+3)
\end{aligned}$$

Ideally, to allow the computation of all four primary estimates, all seven data points must lie within the given data set. When this is not the case, i.e., when any of the four consecutive points for a particular primary estimate falls outside the data range, all four primary estimates cannot be calculated and the method only uses the available primary estimates. Once the primary estimates have been calculated for the endpoint in question, the algorithm computes the final estimate of the first derivative as the weighted mean of these primary estimates.

$$g_i = \frac{u_0 w_0 + u_1 w_1 + u_2 w_2 + u_3 w_3}{w_0 + w_1 + w_2 + w_3}$$

$$\begin{aligned}
w\,(i, j, k, l) &= [d\,(i, j, k, l)\; v\,(i, j, k, l)]^{-1} \\
w_0 &= w\,(i, i-3, i-2, i-1) \\
w_1 &= w\,(i, i+1, i-2, i-1) \\
w_2 &= w\,(i, i+1, i+2, i-1) \\
w_3 &= w\,(i, i+1, i+2, i+3)
\end{aligned}$$

The weight for each primary estimate takes account of the nature of the four consecutive points on which the primary estimate is based. Each set contains the endpoint in question, $p_i$, and three other points, and the resultant primary estimate is assigned a weight which depends on how far the three points are from the $p_i$, given by the distance factor, $d\,(i, j, k, l)$, and how far they are from collinearity with $p_i$, given by the volatility factor, $v\,(i, j, k, l)$.

$$\begin{aligned}
d\,(i, j, k, l) &= (x_j - x_i)^2 + (x_k - x_i)^2 + (x_l - x_i)^2 \\
v\,(i, j, k, l) &= \sum_{a=i,j,k,l} [y_a - (b_0 + b_1 x_a)]^2,\; b_0 = \frac{s_{x^2} s_y - s_x s_{xy}}{4 s_{xx} - {s_x}^2},\; b_1 = \frac{4 s_{xy} - s_x s_y}{4 s_{xx} - {s_x}^2} \\
s_x &= \sum_{a=i,j,k,l} x_a,\; s_y = \sum_{a=i,j,k,l} y_a,\; s_{xy} = \sum_{a=i,j,k,l} x_a y_a,\; s_{xx} = \sum_{a=i,j,k,l} {x_a}^2
\end{aligned}$$

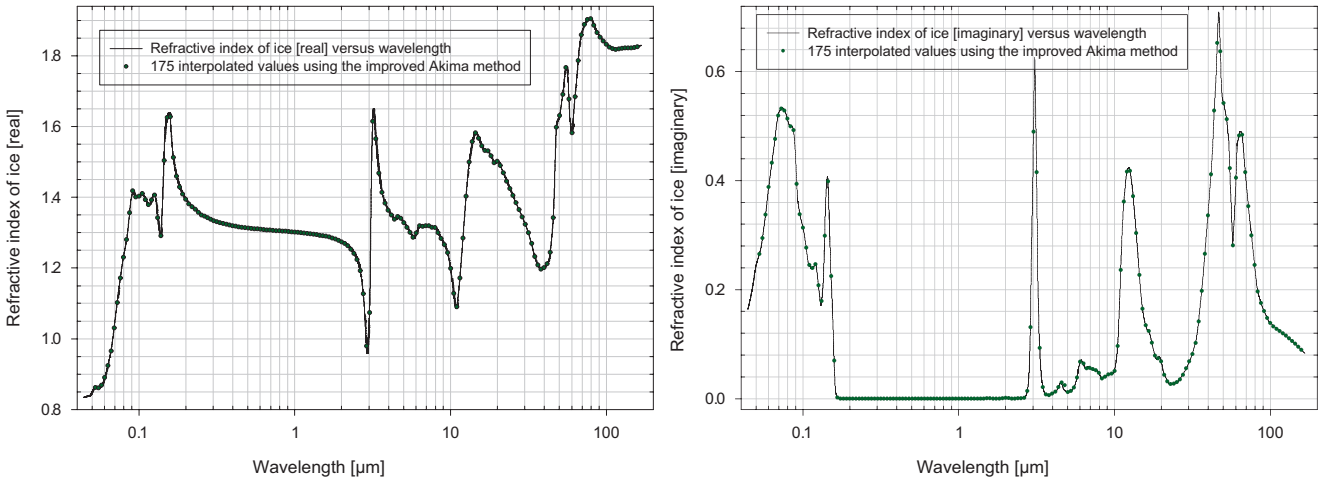



Figure 2.1: Refractive index versus wavelength ($\mu$m) from 45 nm to 167 $\mu$m showing 175 interpolated points spaced evenly in $\log_{10}$.

The further the three points are from $p_i$ and from collinearity with $p_i$, the smaller the associated weight is. The distance factor is defined as the sum of the squares of the distances from $p_i$ to the other three points and the volatility factor is defined as the sum of the squares of the deviations of the four ordinate values from the least squares fit. For the distance factor, the index $i$ must be first, however, the order of the remaining indices can be arbitrary, whereas for the volatility factor, the order of all four indices can be arbitrary. With the primary estimates, $g_i$, in hand, for a given $x$, the value the corresponding ordinate value can be calculated using Eq. 2.5.

The custom implementation, written in C, is an adaptation/port of the author's original 1991 FORTRAN version[5] [175]. Fig. 2.1 shows a purely qualitative verification of the output, in which, the algorithm was employed to interpolate 175 refractive index values, equispaced in $\log_{10}$ between the wavelength values of 45 nm and 167 $\mu$m, for ice. The data was acquired from the REFICE routine, originally developed by Warren in 1984 [176], and subsequently improved by Warren, Gao and Wiscombe in 1995[6].

[5]Association for Computing Machinery Digital Library, http://doi.acm.org/10.1145/114697.116808 & Netlib, http://www.netlib.org/toms/697

[6]Unpublished, source code available in the PyARTS package, a python package developed to compliment the Atmospheric Radiative Transfer Simulator - ARTS. Available from the Satellite Atmospheric Science Group, http://www.sat.ltu.se/arts/tools

## 2.4 DFT functionality

The computational implementation of DFTs can be broken down into several distinct operations. The first involves the creation of a plan, in which, the specific details of the proposed DFT are used to find the most efficient way to compute the transform within the current environment. These details include the dimensions and direction of the transform, the input and output buffers, and the level of optimisation to apply to the plan creation process. In the event that only a few transforms are required, the optimisation level allows for a limited initialisation to be employed to minimise the associated overheads, resulting in a plan that is probably suboptimal. On the other hand, if many transforms are required and the initialisation time is considered irrelevant, a more rigorous analysis can be applied to find the most efficient way of computing the transform. Furthermore, depending on the complexity of the transform and the library being used to provide the DFT functionality, the planning information can also include how many of these transforms are to be computed, the distance between successive transforms and their input and output strides. Once a plan has been constructed, the execution of the transform as dictated by the given plan, can be repeated as many times as is required. Finally, after all requisite transforms have been computed, the associated plan can be destroyed/freed.

There are several highly-optimised libraries available that provide the prescribed functionality and can be used to perform the DFTs required by the matrix-vector kernel. The three potential candidates that were considered for the OpenDDA framework were, the '*Fastest Fourier Transform in the West*', FFTW-2.1.5[7] [177], FFTW-3.1.2[7] [178], and the Intel Math Kernel Library[8] (Intel MKL) Fourier transform interface (DFTI) [179]. The reason for the consideration of two different versions of FFTW is that the FFTW Application Programming Interface (API) was completely re-engineered between versions 2 & 3 and the interface for FFTW 3 is fundamentally different from that of FFTW 2. In order to identify the library that was best suited to provide the requisite DFT functionality, a list of their associated advantages and disadvantages, in relation to the task at hand, was compiled and is shown here.

- **FFTW-2.1.5**
  - Does not support in-place functionality. ✗

[7]Fastest Fourier Transform in the West (FFTW), http://www.fftw.org
[8]Intel Math Kernel Library (MKL), http://developer.intel.com/software/products/mkl

  - Does not use the same plan for both directions. ✘
  - Plans are not associated with specific arrays. ✔
  - Freely available under the terms of the GNU General Public License. ✔
  - Portable. ✔
  - Comparably lightweight DFT library written in C. ✔

- **FFTW-3.1.2**
  - Does support in place functionality. ✔
  - Does not use the same plan for both directions. ✘
  - Plans are associated with specific arrays. ✘
  - Freely available under the terms of the GNU General Public License. ✔
  - Portable. ✔
  - Comparably lightweight DFT library written in C. ✔

- **Intel MKL Fourier transform interface (DFTI)**
  - Does support in place functionality. ✔
  - Does use the same plan for both directions. ✔
  - Plans are not associated with specific arrays. ✔
  - Free for non-commercial use, however, academic use does not qualify for a non-commercial license. ✘
  - Not as portable as the FFTW libraries. ✘
  - Comparably large multifunctional library. ✘

After careful deliberation, based not only on the factors above, but also on the premise that the finished code should be as efficient, accessible, open and portable as possible, both the Intel MKL and FFTW-2.1.5 were ruled out. For the Intel MKL, although it has the advantage that the same plan can be used for both transform directions, it was deemed nonsensical to have the situation where, due to licensing issues, the operability of OpenDDA was wholly reliant on the limited availability of a proprietary library. However, although not being used explicitly, if available, the Intel MKL DFTI routines can still be accessed via FFTW wrappers that have been incorporated into the Intel MKL interface. These wrappers allow FFTW-based programs to use the Intel MKL Fourier transforms without having to alter the program source code. FFTW-2.1.5 was also disregarded as it does not support in-place operation and its use would constitute an unnecessary waste of precious resources.

For FFTW-3.1.2, the fact that plans are associated with specific arrays is not a cause for concern. First of all, once a plan has been created, as long as the parameters are the same, creating new plans for different arrays is cheap. Furthermore, if need be, an existing plan can be applied to alternate arrays of identical size using the so-called *guru* interface, i.e., instead of using '*void fftw_execute(const fftw_plan plan)*' to apply a plan to the arrays for which it was created, it can be applied to different input and output arrays using the *guru* execution function, '*void fftw_execute_dft(const fftw_plan p, fftw_complex *in, fftw_complex *out)*'.

An extremely powerful feature of FFTW-3.1.2 is the provision of an advanced interface, which extends the functionality beyond that of the basic interface. It allows the creation of plans to treat multiple arrays simultaneously and can be used to transform subsets of larger multidimensional arrays by specifying the number of transforms to compute, '*how many*', the associated stride, '*stride*', and the distance between the successive arrays, '*distance*'. For instance, consider a $[P \times J \times K]$ 3D array, stored as a 1D array of length $N = [P \times J \times K]$ for computational efficiency. By specifying the appropriate multiplicities, strides and distances, it is possible to only transform subsections of the array in each direction, i.e., by setting, '*how many*'$= J$, '*stride*' $= 1$ & '*distance*' $= K$, just the first $xy$-plane can be transformed in the $\hat{\mathbf{x}}$ direction, by setting, '*how many*' $= K$, '*stride*' $= K$ & '*distance*' $= 1$, the same plane can be transformed in the $\hat{\mathbf{y}}$ direction, and by setting, '*how many*' $= K$, '*stride*' $= M = J \times K$ & '*distance*' $= 1$, just the first $xz$-plane can be transformed in the $\hat{\mathbf{z}}$ direction.

To provide a method to tailor the initialisation overheads to suit the application, FFTW provides several levels of optimisation that can be applied to the plan creation process. The two that are predominantly used are FFTW_ESTIMATE and FFTW_MEASURE. The former quickly builds a reasonable plan, without computation, that is probably sub-optimal, whereas the latter instructs FFTW to run and measure the execution time of several DFTs in order to find the best way to compute the transform [178]. It is important to note that, if using FFTW_MEASURE, the plans must be created before the arrays are initialised, as FFTW_MEASURE overwrites the input and output arrays during the planning procedure.

FFTW is most efficient for system sizes which satisfy $2^a 3^b 5^c 7^d 11^e 13^f$, where $a, b, c, d$ are arbitrary and $(e + f \leq 1)$ [178], and in order to ensure that the best possible performance can be achieved, it is often necessary to zero-pad the arrays out to a sympathetic dimension. To facilitate this functionality, the new code calculates

the minimum amount of zero-padding that is needed in order to satisfy the requisite prime factorisation. The algorithm, see Excerpt 2.2, begins with the original system size and tests the dimension by successive factorisation using the pertinent primes and steadily increments the required zero-padding until a value is reached that satisfies the prescribed criterion.

Code Excerpt 2.2: Algorithm to calculate the zero-padding required to enable the use of the most efficient DFT algorithms.

```
/* orig: Original system size
   zpad: Zero-padding */
int i,orig,size;

size=orig;zpad=0; /* Begin with no zero-padding */
while(size!=1){ /* While not factored */
   size=orig+zpad; /* Original size + zero-padding */
   while((size&1)==0){size=size>>1;} /* While even, factor 2, bit-shift divide */
   if(size==1){break;} /* Break if factored */
   for(i=3;i<=7;i+=2){ /* i=3,5,7 */
      while(size%i==0){size/=i;} /* Factor i */
      if(size==1){break;} /* Break if factored */
   }
   if(size==1){break;} /* Break if factored */
   if(size%11==0){size/=11;} /* Factor 11 */
   else if(size%13==0){size/=13;} /* Factor 13 (Only 1 of 11 or 13 allowed ) */
   if(size==1){break;} /* Break if factored */
   else{
     zpad++; /* If not factored, increment the zero-padding and try again */
   }
}
```

## 2.5 Complex functionality in C

ANSI C, as defined by the ISO C89 standard, does not contain native support for complex numbers and such support was not introduced to C until the ISO C99 standard. The standard defines a single, double & long double complex type and also defines a number of complex mathematical functions. Unfortunately, apart from the standard itself, documentation for this functionality is scarce. This is probably due, in part, to the fact that this complex support does not seem to be widely used and to the fact that not all C compilers are compliant with the ISO C99 standard.

The requisite complex support is available from external libraries such as the GNU Scientific library[9] (GSL) [180]. However, during the development process, a primary aim was to minimise the reliance on such external dependencies. The inclusion of such a facility, just to provide complex functionality, was considered excessive. Due to this reluctance to out-source the complex support, the ambiguity of C99 support and the lack of native complex support within the ANSI C89 standard, to circumvent any

[9]GNU Scientific library (GSL), http://www.gnu.org/software/gsl

possible complex support and associated portability issues, it was decided to take the bull by the horns and custom build the necessary complex functionality. Although this may seem like a rather severe course of action, the creation of more than adequate complex support requires a great deal less effort than one might think. The first step in the development was to actually define the new complex types. For simplicity, these were defined as basic structures, i.e., for the double-complex datatype, the definition takes the form listed in Excerpt 2.3.

Code Excerpt 2.3: C structure-based double-complex type definition.

```
#ifndef _DCOMPLEX_ /* If not defined already */
#define _DCOMPLEX_
typedef struct{
    double dat[2]; /* real part=dat[0], imaginary part=dat[1] */
} dcomplex;
#endif
```

Similar definitions exist for both single-complex and long double-complex types. The use of these simple structure-based definitions allows the components of a particular complex number to be accessed directly, without the use of specialised functions, leading to compact, efficient, readable, intuitive and maintainable code.

Apart from the actual complex type definitions themselves, it turns out that just the following ten complex mathematical functions, ranging from elementary algebraic operations to the vector norm, are all that are needed to provide all the requisite complex functionality.

$z_0 = a + bi,\ z_1 = c + di$

1. **complex conjugate:** ${z_0}^* = a + (-1)\,bi$
2. **absolute magnitude:** $|z_0| = \sqrt{a^2 + b^2}$
3. **scale:** $(w)\,(z_0) = (w)\,(a) + (w)\,(b)\,i$
4. **& 5. addition & subtraction:** $z_0 \pm z_1 = (a \pm c) + (b \pm d)i$
6. **multiplication:** $(z_0)\,(z_1) = (ac - bd) + (ad - bc)i$
7. **division:** $\dfrac{z_0}{z_1} = \left(\dfrac{z_0}{z_1}\right)\left(\dfrac{{z_1}^*}{{z_1}^*}\right) = \dfrac{(ac + bd) - i(ad - bc)}{|z_1|^2}$
8. **exponential:** $e^{z_0} = e^a\,[\cos b + (\sin b)\,i]$
9. **square root**[181]**:** $\sqrt{z_0} = \sqrt{\dfrac{|z_0| + a}{2}} + \left(\dfrac{b}{|b|}\right)\sqrt{\dfrac{|z_0| - a}{2}}$
10. **vector 2-norm:** $\mathbf{x} = x_0, x_1 \ldots x_{n-1},\ ||x||_2 = \sqrt{\sum_{i=0}^{n-1} |x_i|^2} \equiv \sqrt{x^T x}$

## 2.6 Iterative methods

**Iterative** methods involve starting with an initial guess $x_0$ for the solution to $Ax = b$ and iteratively improving upon the initial guess, $x_1, x_2, \ldots$, until some predefined criterion have been satisfied. Usually, the iterative process ceases when either the associated error in $Ax_i = b$ is below a convergence threshold or when an upper limit on the number of iterations has been reached; whichever comes first.

### 2.6.1 Conjugate Gradient basics

**The** basic Conjugate Gradient (CG) Method [182, 183, 184, 185] is a method for solving the linear system $Ax = b$, where the matrix $A$ is of order $n$, symmetric, $A^T = A$, and positive-definite, $x^T Ax > 0$, $\forall x \neq 0$, $b$ is a known vector of length $n$, and $x$ is the unknown requisite solution, also of length $n$. CG is based on the minimisation of the associated quadratic form $f(x)$ [185].

$$f(x) = \frac{1}{2}x^T Ax - b^T x + c, f'(x) = \frac{1}{2}A^T x + \frac{1}{2}Ax - b = Ax - b, \iff A^T = A \quad (2.6)$$

Here $f'(x)$ is the gradient, i.e., the direction of greatest increase of $f(x)$. Since $A$ is positive-definite, $f(x)$ can be minimised by setting $f'(x) = 0 \Rightarrow Ax = b$, and thus, the solution to $Ax = b$ is a critical point of $f(x)$ and minimising $f(x)$ is equivalent to solving $Ax = b$. The solution process begins with an initial guess $x_0$ and is iteratively improved upon until either, the residual $r_i = b - Ax_i$ is below a predefined convergence threshold or a certain upper limit on the number of iterations has been reached. The residual can either be thought of as an indication of how far the current approximate solution $x_i$ is from solving the system, or from comparison with Eq. 2.6, $-f'(x_i) = b - Ax_i = r_i$, and thus, the residual can alternatively be thought of as the direction of steepest descent [185].

In order to get to grips with the unintuitive world of conjugate gradients, a few underlying prerequisites need to be established to set the scene. The starting point is the so-called method of steepest descent [183]. The basic premise is to start from the initial guess $x_0$ and for each iteration $i > 0$, move in the direction of local steepest descent $r_i$ by an amount $\alpha_i$ such that $f(x_{i+1})$ is minimised along that line, $x_{i+1} = x_i + \alpha_i r_i$, where $r_i$ & $\alpha_i$ are known as the search direction and step length, respectively. $f(x_{i+1})$ is minimised when the directional derivative, of which the partial derivative is just a special case, is equal to zero, $\frac{d}{d\alpha_i} f(x_{i+1}) = f'(x_{i+1})^T \frac{d}{d\alpha_i}(x_{i+1}) = f'(x_{i+1})^T r_i =$

$r_{i+1}^T r_i = 0$ [185]. By direct substitution, this requirement for orthogonality provides the requisite expression for $\alpha_i$, allowing the method of steepest descent to take shape.

$$\begin{aligned} r_{i+1}^T r_i &= 0 = (b - Ax_{i+1})^T r_i = [b - A(x_i + \alpha_i r_i)]^T r_i \\ &= (b - Ax_i)^T r_i - \alpha_i (Ar_i)^T r_i = r_i^T r_i - \alpha_i (Ar_i)^T r_i \Rightarrow \alpha_i = \frac{r_i^T r_i}{r_i^T A r_i} \end{aligned}$$

**Method of Steepest Descent**

$$r_i = b - Ax_i, r_0 = b - Ax_0$$

$$\begin{aligned} \alpha_i &= \frac{r_i^T r_i}{r_i^T A r_i} \\ x_{i+1} &= x_i + \alpha_i r_i \\ r_{i+1} &= r_i - \alpha_i A r_i \qquad (Eq.\ 2.7) \times (-A) + b \end{aligned} \tag{2.7}$$

The quadratic form is minimised when the step length is such that the movement is to where the gradient is orthogonal to the search line, and thus, the new direction is orthogonal to the direction just traversed. The problem with this method is that there is no track record; no reliance on other previous directions in the search history. As a result, convergence can be slow as the method often takes multiple steps in the same direction, zigzagging back and forth in response to the local gradient, which does not point toward the global minimum, toward the eventual solution [173, 186, 185, 183], a bit like a 'downhill' skier.

The solution to this problem is to choose search directions $p_i$, $x_{i+1} = x_i + \alpha_i p_i$ that do not necessarily correspond to the residuals (directions of steepest descent), $r_i$, but do take account of all previous search directions. An important factor in the choice of these new search directions is the so-called error term, $e_i = x_i - x$, which is an expression for how far the current approximation is from the actual solution [185], i.e., unlike $r_i$, which is the error in $Ax = b$, $e_i$ is the error in $x$. One option that would incorporate information about previous directions and would eliminate multiple steps in any direction would be to choose $p_i$ to be a set of conjugate directions, i.e., a set of $n$ mutually orthogonal search directions. By taking a single step in each direction with a step length $\alpha_i$ that moves $x_i$ to a point $x_{i+1}$ and results in $p_i$ being orthogonal to the associated error $e_{i+1}$, another step in the search direction $p_i$ would never be required, and after $n$ such steps, the system would be solved [185]. As before, by direct

substitution, this requirement for orthogonality provides the requisite expression for $\alpha_i$.

$$p_i^T e_{i+1} = 0 = p_i^T (x_{i+1} - x) = p_i^T (x_i + \alpha_i p_i - x) = p_i^T (e_i + \alpha_i p_i) \Rightarrow \alpha_i = -\frac{p_i^T e_i}{p_i^T p_i} \quad (2.8)$$

The problem with Eq. 2.8 is that the calculation $e_i$ requires the knowledge of the actual solution $x$, and if this were known, the system would be solved and iterative methods would be irrelevant. However, the definition of $e_i$ does imply that $r_i = b - Ax_i = b - A(e_i + x) = b - Ae_i - Ax = -Ae_i$. Thus, by requiring that $p_i$ and $e_{i+1}$ be $A$-orthogonal, $p_i^T A e_{i+1} = 0$, rather than orthogonal, the unknown error term in Eq. 2.8 can be eliminated.

$$\begin{aligned} p_i^T A e_{i+1} = 0 &= p_i^T A (x_{i+1} - x) = p_i^T A (x_i + \alpha_i p_i - x) \\ &= p_i^T A (e_i + \alpha_i p_i) \Rightarrow \alpha_i = -\frac{p_i^T A e_i}{p_i^T A p_i} = \frac{p_i^T r_i}{p_i^T A p_i} \end{aligned} \quad (2.9)$$

To obtain a set of $n$ $A$-orthogonal search directions, the Gram-Schmidt algorithm [183], see §2.6.4 for more details, can be used to generate the requisite directions from a chosen set of $n$ linearly independent vectors, $u_0 \ldots u_{n-1}$ [185], i.e., no constituent vector can be expressed as a linear combination of the others.

$$\begin{aligned} i = 0 &: p_0 = u_0 \\ i > 0 &: p_i = u_i + \sum_{j=0}^{i-1} \beta_{ij} p_j \end{aligned} \quad (2.10)$$

The coefficients are chosen so that the current search direction $p_i$ is $A$-orthogonal to all previous search directions, $p_j$, $j < i$. This is done by taking each of the predecessors in turn, finding the component of $u_i$ that is not $A$-orthogonal to it and removing it to ensure the $A$-orthogonality of $p_i$ to $p_j$, $j < i$. The criterion for $A$-orthogonality, $p_i^T A p_j = 0$, $i \neq j$, can be used to identify the requisite expression for the coefficients by multiplying Eq. 2.10 by $Ap_k$ and using the fact that only the '$j = k$' term will survive in the resulting summation.

$$p_i^T A p_k = u_i^T A p_k + \sum_{j=0}^{i-1} \beta_{ij} p_j^T A p_k = 0 = u_i^T A p_k + \beta_{ik} p_k^T A p_k, \; i \neq k$$

$$\Rightarrow \beta_{ik} = -\frac{u_i^T A p_k}{p_k^T A p_k} \tag{2.11}$$

This formalism has some convenient properties. First of all, since the search directions $p_i$ are, by definition, $A$-orthogonal, $p_i^T A p_j = 0,\ i \neq j$, and the error term is evermore $A$-orthogonal to all previous search directions, $p_i^T A e_{i+1} = 0$, the fact that $r_i = -Ae_i$ implies that the residual is evermore orthogonal to all previous search directions [185].

$$p_i^T r_j = 0,\ i \neq j \tag{2.12}$$

Furthermore, by multiplying Eq. 2.10 by $r_k$, Eq. 2.12 implies that both $u_i^T r_k = 0,\ i \neq k$ and $p_i^T r_i = u_i^T r_i,\ k = i, i \neq j$.

$$p_i^T r_k = u_i^T r_k + \sum_{j=0}^{i-1} \beta_{ij} p_j^T r_k \quad \Rightarrow u_i^T r_k = 0,\ i \neq k \tag{2.13}$$

$$p_i^T r_i = u_i^T r_i + \sum_{j=0}^{i-1} \beta_{ij} p_j^T r_i \quad \Rightarrow p_i^T r_i = u_i^T r_i,\ k = i, j \neq i \tag{2.14}$$

Unfortunately, the implementation of the prescribed theory has one serious flaw. For a general choice of the initial vectors, $u_i$, $i = 0 \ldots n-1$, the search direction $p_i$ in the Gram-Schmidt algorithm is dependent on all previous search directions, $p_j$, $j < i$, and thus, all previous search directions must be recorded in order to calculate $p_i$.

The Conjugate Gradient Method circumvents this issue by choosing to base the $A$-orthogonal search directions $p_i$ on the residuals $r_i$, i.e., $u_i = r_i$. Each residual is orthogonal to all previous residuals, $r_i^T r_j = 0,\ i \neq j$, see Eq. 2.13. Furthermore, each new residual $r_{i+1}$ is just a linear combination of the previous residual $r_i$ and $Ap_i$, creating what is a known as a Krylov subspace, a subspace generated by repeatedly multiplying a vector by a matrix [185].

$$\begin{aligned} x_{i+1} &= x_i + \alpha_i p_i,\ e_i = x_i - x \\ \Rightarrow e_{i+1} &= e_i + \alpha_i p_i \\ \Rightarrow r_{i+1} &= -Ae_{i+1} = -A\left(e_i + \alpha_i p_i\right) = -Ae_i + \alpha_i A p_i = r_i + \alpha_i A p_i \end{aligned} \tag{2.15}$$

The choice of $u_i = r_i$ has the effect of trivialising the Gram-Schmidt procedure as combining the following three facts means that the residual $r_{i+1}$ is automatically

$A$-orthogonal to all previous search directions except $p_i$ [185].

1. The search directions are $A$-orthogonal, $p_i^T A p_j$, $i \neq j$.
2. Each residual is orthogonal to all previous search directions, $p_i^T r_j$, $i < j$, see Eq. 2.12.
3. The residual $r_{i+1}$ is just a linear combination of the previous residual $r_i$ and $Ap_i$, see Eq. 2.15.

The wonderful upshot is that the Gram-Schmidt algorithm is reduced from having to worry about $A$-orthogonality to all previous search directions to just the previous search direction, $p_i = r_i + \beta_i p_{i-1}$, i.e., only the $k = i - 1$ term in the summation in Eq. 2.10. The final step for the Conjugate Gradient Method is to obtain an expression for $\beta_i$ by setting $k = i - 1$ in the expression for the Gram-Schmidt coefficients in Eq. 2.11 and using $r_i \times$ Eq. 2.15 to simplify the resultant [185].

$$
\begin{aligned}
\beta_i &= -\frac{r_i^T A p_{i-1}}{p_{i-1}^T A p_{i-1}}, \; k = i-1 \\
r_i \times \text{Eq. 2.15} &= r_i^T r_{k+1} = r_i^T r_k - \alpha_k r_i^T A p_k \Rightarrow r_i^T r_i = r_i^T r_{i-1} - \alpha_{i-1} r_i^T A p_{i-1}, \; k = i-1 \\
\Rightarrow r_i^T A p_{i-1} &= -\frac{1}{\alpha_{i-1}} r_i^T r_i \qquad \text{[by Eq. 2.13]} \\
\Rightarrow \beta_i &= \frac{r_i^T r_i}{p_{i-1}^T r_{i-1}} \qquad (2.16)
\end{aligned}
$$

**Conjugate Gradient Method**

$$
\begin{aligned}
r_i &= b - Ax_i, r = r_0 = b - Ax_0 \\
\hline
i = 0 \; \{p_0 &= r_0\} \\
i > 0 \; \Bigg\{\beta_i &= \frac{r_i^T r_i}{r_{i-1}^T r_{i-1}} \qquad \text{[by Eqs. 2.16 \& 2.14]} \\
p_i &= r_i + \beta_i p_{i-1} \Bigg\} \\
\alpha_i &= \frac{r_i^T r_i}{p_i^T A p_i} \qquad \text{[by Eqs. 2.9 \& 2.14]} \\
x_{i+1} &= x_i + \alpha_i p_i \\
r_{i+1} &= r_i - \alpha_i A p_i
\end{aligned}
$$

### 2.6.2 Algorithmic variants & custom-built implementations

**OpenDDA** incorporates a wide variety of iterative methods, ranging from the basic CG method to much more elaborate and involved algorithms. To date, no definitive "best" algorithm has been identified. DDSCAT[10] [5] recommends the stabilised version of the BiConjugate-Gradients (BiCGSTAB) [187], whereas ADDA[11] [188], which also includes the BiCGSTAB algorithm, seems to favour the quasi-minimal residual method for symmetric systems ($\text{QMR}_{sym}$) [189]. To permit further investigation, allow meaningful comparison with alternate DDA implementations and to maximise the flexibility and the usefulness of the end product, OpenDDA includes the following iterative methods:

- Conjugate-Gradients (CG) [182, 183, 184, 185], see §C.1.
- Conjugate-Gradients Squared (CGS) [190, 184], see §C.2.
- BiConjugate-Gradients (BiCG) [191, 184], see §C.3.
- BiConjugate-Gradients for symmetric systems ($\text{BiCG}_{sym}$) [189], see §C.4.
- Stabilised version of the BiConjugate-Gradients (BiCGSTAB) [187], see §C.5.
- Restarted, stabilised version of the BiConjugate-Gradients (RBiCGSTAB) [192], see §C.6.
- Quasi-minimal residual with coupled two-term recurrences (QMR) [193, 194, 195], see §C.7.
- Quasi-minimal residual for symmetric systems ($\text{QMR}_{sym}$) [189], see §C.8.
- Transpose-free quasi-minimal residual (TFQMR) [196], see §C.9.
- A variant of the stabilised version of the BiConjugate-Gradients (BiCGSTAB) based on multiple Lanczos starting vectors ($\text{ML(n)BiCGSTAB}_{orig}$) [197], see §C.10.
- A variant of the stabilised version of the BiConjugate-Gradients (BiCGSTAB) based on multiple Lanczos starting vectors (ML(n)BiCGSTAB) [198] Appendix. Algorithm 2.2′ ML(n)BiCGSTAB with preconditioning, see §C.11.
- A variant of the stabilised version of the BiConjugate-Gradients (BiCGSTAB) based on multiple Lanczos starting vectors ($\text{ML(n)BiCGSTAB}_{ss}$) [198] Appendix. Algorithm 2.3′ ML(n)BiCGSTAB with preconditioning (storage-saving), see §C.12.

[10] DDSCAT, http://www.astro.princeton.edu/ draine/DDSCAT.html
[11] ADDA, http://www.science.uva.nl/research/scs/Software/adda

The pseudocode for each algorithm, along with a brief description of the constituent vectors and variables, and where relevant, a quick note on the computational implementation of the algorithm, are presented in Appendix C. Each has been custom-built to integrate seamlessly with the highly-optimised DFT matrix-vector kernel, see §3 & §4. Each exists in all precisions, single, double and long double, and apart from the serial implementation, architecture specific versions have been constructed to take advantage of both shared-memory and distributed-memory environments. Apart from the actual operational parallelisation itself and the necessary decomposition of data across the constituent nodes in the distributed-memory implementation, the only major difference between the serial and parallel versions is the so-called reduction processes. These are required to implement the data reduction for the scalar products. In the parallel environments, the active processors collaborate on a particular task, and thus, for the scalar product, each processor will only treat a specific subset and the individual components must be combined to obtain the actual result. For the shared-memory environment, the OpenMP API, see §4.2, §4.2.1 & §B.1 for more details, includes such a reduction routine. However, unfortunately, the interface does not support user-defined datatypes like the custom structure-based complex type which is defined in OpenDDA, see §2.5. To overcome this problem, for the scalar products, the shared-memory implementation calculates and reduces the real and imaginary components separately. For the distributed-memory environment, the scalar products, which act as synchronisation points, have the added complication that the individual components are computed on autonomous nodes and are physically located in separate address spaces. The reduction process must not only combine the components from all processes but must also distribute the result back to all processes. The MPI API, see §4.3 for more details, does incorporate this functionality. However, to implement this routine, a user-defined MPI datatype must be defined to represent the complex type as two contiguous values, and furthermore, a custom reduction operation function must be passed to the reduction routine to enable the addition of this custom MPI datatype. In terms of the associated performance of both the shared-memory and distributed-memory implementations of the reduction routines, no adverse performance issues have been observed in relation to the necessary custom alterations to support the reduction of complex numbers.

Please note that, apart from the brief interjection in the previous section to describe the fundamentals upon which CG methods are based, no attempt will be made to decipher the intricacies of the more involved schemes. Not only are the associated

complexities well beyond the scope and time limits of the project, but due to the author's limited exposure to hardcore linear algebra, any mathematical digression into this unfamiliar territory would be token at best and would not add to this discourse. The various methods are both well-proven and accepted, and as such, have been incorporated without the need for any theoretical justification. As far as verification goes, the fact that all the custom-built implementations, whose performance will be discussed in §6, converge to the same result is considered as ample verification of their correct operation. Furthermore, since there is nothing novel about the actual iterative schemes themselves, the discussion will focus solely on the custom implementations from a purely computational perspective. For detailed descriptions and dissections of the methods, readers are referred to either the associated references, see above, or to the erudite works of Saad [199] and/or Barrett et al. [184].

In terms of computational efficiency, a key ingredient is that the custom-built implementations only store information corresponding to occupied lattice sites, i.e., this boils down to the elimination of zero storage within all iterative solution vectors. For the majority of cases, this provides a significant reduction in the memory required to store these vectors, as the target being simulated does not fully utilise the computational domain, generally having fewer occupied sites towards the exterior of the domain. As a quick example, consider a spherical target based on an $18\times18\times18$ computational grid. Although the grid contains 5832 lattice sites, for a spherical target, only 3112 correspond to occupied lattice sites, and dumping the unoccupied data, which is identically zero, equates to a 46.64% decrease in the memory required for the iterative vectors.

For the serial case, the computational implementation of this optimisation is straightforward as it does not matter how many occupied sites are in each plane and the vector lengths are simply reduced from $3N$ to $3N_d$, where $N$ is the number of elements in the computational domain and $N_d$ is the number of elements in the actual target. However, for both parallel implementations of the OpenDDA framework, as will be discussed in §4, the delegation of work to processors is done on a 'plane-by-plane' basis and it is crucial that the starting point for each plane can be identified in the vectors. To facilitate this optimisation in the parallel environments, during the target creation process, a control array is created to keep track of the populations, and thus, enable each active process to attend to its specific subset. For the shared-memory implementation this requires a single global population array, whereas for the distributed-memory case, each node has a separate population array

specific to only the planes that the node has locally.

The only other requirement for the elimination of the zero-storage is the creation of data extension and compression algorithms. A data extension algorithm is needed to extend/embed a target vector out/into a larger zero-padded array to facilitate the DFT matrix-vector product of this vector with the tensor components, and subsequently, a data compression algorithm is needed to compress the resultant back into the computationally efficient form with just the entries corresponding to occupied sites. It is worth noting that although the data extension is performed out-of-place, from the original target vector to the larger zero-padded array, the data compression is performed in-place in the matrix-vector product result buffer. This is important as for the shared-memory implementation, all active processors are working within a single address space, and thus, the data compression algorithm has a maximum scalability of three in the shared-memory environment. This is because there are three vector components and to prevent pertinent data loss by multiple processes working within the same array, the algorithm is limited to one processor per component. This is not an issue for concern as the data compression constitutes only a negligible fraction of the DFT kernel's runtime and the limited scalability of this section has very little effect on the scalability as a whole. This will be seen explicitly in §6. Both the serial and distributed-memory implementations, by definition, are free from such concerns.

### 2.6.3 Preconditioning

For a linear system $Ax = b$, preconditioning is a technique aimed at improving the convergence by solving an equivalent system $M^{-1}Ax = M^{-1}b$, where the condition of $M^{-1}A$ is more favourable than that of the original matrix $A$ [184]. The condition of a particular system is quantified by the condition number $\kappa(A)$, which is a measure of the spread of the spectrum of $A$, denoted $\sigma(A)$, i.e., the set of all eigenvalues of the matrix $A$, or in other words, the solutions to the characteristic equation $det\,(A - \lambda\mathbf{I}) = 0$, where $det$ refers to the determinant and $\mathbf{I}$ is the identity matrix.

$$\kappa(A) = \rho(A)\rho(A^{-1}) = \frac{\lambda_{max}}{\lambda_{min}}$$

Here, $\rho(A)$ is the so-called spectral radius of the matrix A and is simply the magnitude of the dominant eigenvalue, i.e., the eigenvalue of largest magnitude, $\rho(A) = \lambda_{max} = max\left(|\lambda|\right), \lambda \in \sigma(A)$. For $A^{-1}$, $Ax = \lambda x$ implies that $A^{-1}x = \lambda^{-1}x$, i.e., the eigenvalues of $A^{-1}$ are the reciprocal of the eigenvalues of $A$, and thus, the spectral radius of $A^{-1}$ is $(1/\lambda_{min})$. The spectral radius is in itself an important consideration for iterative methods, as, if the magnitude of an eigenvalue is greater than unity, since $Ax = \lambda x$, repeated application of $A$ will result in divergence to infinity [185].

For singular matrices, which are matrices with no inverse, i.e., the determinant, which is the product of the eigenvalues is zero, meaning that at least one of the eigenvalues is zero, $\kappa(A) = \infty$, and for the identity matrix $\mathbf{I}$, $\kappa(A) = 1$. Systems with a small condition number are said to be '*well-conditioned*', whereas for large condition numbers, the descriptor '*ill-conditioned*' is used. Although, for singular matrices, $det(A) = 0$, the relation $det(A) \approx 0$ is not an appropriate measure of condition. There is little correlation between the determinant of the coefficient matrix and the condition of $Ax = b$ [183]. On the other hand, the condition number gives a good indication of the closeness to singularity. Apart from this measure, the condition number may also be considered as a relative error magnification factor; a sort of quantification of the sensitivity of the linear system. If $\kappa(A)$ is large, then the convergence of the '*ill-conditioned*' system may be hampered or even prevented by the loss of accuracy due to the amplification of perturbations due to rounding errors. For such systems, only slight changes to the problem can produce large changes in the solution. Preconditioning attempts to reduce the condition number and/or produce a more clustered eigenvalue structure, as in general, the convergence is worst when the eigenvalues are spread out, and hopefully, as a result, improve the convergence of the associated linear system.

The chosen preconditioner should be closely related to the original matrix, should be easily invertible and should be as lightweight as possible so that the additional resources required to initialise and apply the preconditioning do not negate the associated benefits. For OpenDDA, only the simplest possible preconditioner, that of Point Jacobi preconditioning [184], i.e., division by the diagonal, has been implemented, where for computational efficiency, the divisions are performed as multiplications using the reciprocals. The choice of preconditioner is constrained by the fact that due to the size of the simulations, the inherent symmetries that exist within the interaction matrix and the use of convolutional theory to exploit these symmetries, only the first column and diagonal of the matrix is stored. This means that

the chosen preconditioner must rely solely on just the diagonal, the first column or a combination of both.

Circulant preconditioners [200, 201, 202] where considered briefly as they possess several ideal properties. Firstly, as will be discussed in detail in the next chapter in relation to the optimisation of the matrix-vector products that power the iterative methods, the first column of a circulant matrix is all that is needed to completely define the matrix. Secondly, for circulant matrices, inversion, matrix-vector multiplication and matrix-matrix multiplication can be performed extremely efficiently using DFTs and the knowledge of just the first column of the matrix alone. Unfortunately, the applicability of the circulant preconditioners is extremely limited. The systems upon which the preconditioners are based must have constant diagonals. This means that, for all systems, except the very specific case of systems based on a homogeneous rectangular target, the main diagonal of what is a diagonally dominant system cannot be taken into account.

### 2.6.4 Pseudorandom number generation & vector orthonormalisation

Apart from the use of the FFTW package [178], which is used to implement the DFTs that are required by the matrix-vector kernel, the only other instance where an external library is used to provide supplementary functionality is for the generation of pseudorandom numbers. SFMT19937, which stands for SIMD (Single Instruction Multiple Data) oriented Mersenne Twister [203], is used to generate the pseudorandom numbers. SIMD is a set of special operations supported by some processors to allow a single operation to be applied to several numbers simultaneously. This particular generator is written in C, has a period of $2^{19937} - 1$, and is a new variant of the Mersenne Twister[12] algorithm [204], which the authors claim, has a better equidistribution property and is roughly twice as fast as the original Mersenne Twister algorithm.

Although, at present, pseudorandom number generation is not used extensively within the new code, an efficient pseudorandom number generator is always a valuable asset. Such a resource could find application in the creation of custom targets, the specification of random target orientations, etc. At the moment, the only instance

[12]Mersenne Twister, http://www.math.sci.hiroshima-u.ac.jp/∼m-mat/MT/emt.html

where pseudorandom number generation is used is for the creation of the $n$ orthonormal initial vectors, $q_0 \dots q_{n-1}$, required by the three variants of the ML(n)BiCGSTAB algorithm, see §2.6.2. SFMT19937 is initialised by an array of $n$ unsigned integers which are generated using the *lrand48* function; seeded using the clock, via the *srand48* function. Both *srand48* and *lrand48* are defined as part of the standard library. The requisite orthonormal vectors, $q_0 \dots q_{n-1}$, are produced by creating $n$ pseudorandom vectors, $a_0 \dots a_{n-1}$, and subsequently, orthonormalising the result using the Modified Gram-Schmidt (MGS) algorithm [183]. Rather than thinking about this in terms of single vectors of length $m$, by treating the individual vectors $a_i$ as the columns of an $m \times n$ matrix $A$, MGS can be considered in term of the QR decomposition of the matrix $A$, i.e., $A = QR$, where the columns of $Q$, $q_0 \dots q_{n-1}$, are orthonormal, the matrix $Q$ has the same dimension as $A$ and the matrix $R$ is $n \times n$ and upper triangular.

For the purposes of the task at hand, the values of the matrix $R$, which give the relationship between the columns of $A$ and $Q$, i.e., the original vectors $a_i$ and new orthonormal vectors $q_i$, $i = 0 \dots n-1$, are superfluous and can be disregarded. Furthermore, once the orthonormal vectors have been calculated, the original vectors are no longer required, and thus, for computational efficiency, the same storage is used for both $A$ & $Q$, i.e., the original vectors are overwritten by the new orthonormal vectors during the orthonormalisation procedure. With these alterations to MGS the algorithm for the orthonormalisation of the $n$ pseudorandom starting vectors becomes:

**Modified Gram-Schmidt Orthonormalisation**

**T**heoretically

$$i = 0: \quad q_0 = \frac{a_0}{||a_0||_2}$$

$$i > 0: \quad q_i = a_i - \sum_{j=0}^{i-1} \left(q_j^T a_i\right) q_j, \; q_i = \frac{q_i}{||q_i||_2}$$

**C**omputationally

**for** $s = 0$ **to** $n-1$

$\quad q_s = \dfrac{a_s}{||a_s||_2}$

$\quad$ **for** $t = s+1$ **to** $n-1$

$\quad\quad a_t = a_t - \left[q_s^T a_t\right] q_s$

$\quad$ **end for t**

**end for s**

Apart from the normalisation aspect, the algorithm boils down to taking each of the previous vectors in turn, finding the component of the unconverted vector that is not orthogonal to it and removing it to ensure orthogonality of the resultant. Before the orthonormalisation can begin, the first task is to set the $n$ original pseudorandom vectors, $a_0 \dots a_{n-1}$, using SFMT. Since orthogonality is a property of a set of vectors, not of an individual vector, for the first vector, $q_0$, only normalisation is important, $q_0 = a_0/||a_0||_2$. For each newly constructed orthonormal vector, $q_s$, the algorithm computes the component of each $a_t$ that is parallel to $q_s$, given by $\left[q_s^T a_t\right] q_s$, and removes it from $a_t$ to ensure orthogonality with $q_s$. To see why this works, consider the second vector, $q_1$, which must be constructed so that it is both orthogonal to $q_0$ and normalised. This is done by defining $q_1 = a_1 - f q_0$ and choosing $f$ such that $q_1$ is orthogonal to $q_0$, i.e., $q_0^T q_1 = 0 = q_0^T (a_1 - f q_0) = q_0^T a_1 - f q_0^T q_0$, and since, by definition, $q_0^T q_0 = 1$, $f = q_0^T a_1$, which gives, $q_1 = a_1 - \left(q_0^T a_1\right) q_0$. Thus, by removing $\left[\left(q_0^T a_1\right) q_0\right]$ from $a_1$, $q_1$ becomes orthogonal to $q_0$ and all that remains to complete the orthonormalisation is the normalisation of the result by $||q_1||_2$. In a similar manner, the orthogonalisation of the rest of the remaining vectors with $q_0$ is given by removal of $\left[\left(q_0^T a_t\right) q_0\right]$ from $a_t$, $t = 2 \dots n-1$. For each new orthonormal vector, $q_s$, this is repeated, whereby $\left[\left(q_s^T a_t\right) q_s\right]$ is removed from the succeeding unfinished vectors to ensure that each of them will be orthogonal to its predecessors, e.g.,

$$\begin{aligned}
q_2 &= a_2 - \left(q_0^T a_2\right) q_0 - \left(q_1^T a_2\right) q_1,\ q_2 = \frac{q_2}{||q_2||_2}, \\
q_3 &= a_3 - \left(q_0^T a_3\right) q_0 - \left(q_1^T a_3\right) q_1 - \left(q_2^T a_3\right) q_2,\ q_3 = \frac{q_3}{||q_3||}, \\
&\vdots \\
q_{n-1} &= a_{n-1} - \sum_{i=0}^{n-2} \left(q_i^T a_{n-1}\right) q_i,\ q_{n-1} = \frac{q_{n-1}}{||q_{n-1}||_2}.
\end{aligned} \tag{2.17}$$

For instance, for Eq. 2.17, the second term, $-\left(q_0^T a_3\right) q_0$, ensures that $q_3$ is orthogonal to $q_0$, the third term, $-\left(q_1^T a_3\right) q_1$, ensures that $q_3$ is orthogonal to $q_1$, and the last term, $-\left(q_2^T a_3\right) q_2$, ensures that $q_3$ is orthogonal to $q_2$. The final result is a set of $n$ orthonormal vectors whose properties can be verified by checking that $||q_s||_2 = 1$ and $q_s^T q_t = \begin{cases} 0, & \text{if } s \neq t; \\ 1, & \text{if } s = t. \end{cases}$, $s = 0 \dots n-1, t = 0 \dots n-1$.

*"In theory, there is no difference between theory and practice; In practice, there is."*

**Chuck Reid**

*"An expert is a person who has made all the mistakes that can be made in a very narrow field."*

**Niels Bohr**

# 3 DFT kernel - Serial implementation

## 3.1 Introduction

The following few sections are concerned with the fundamental theory which underpins DFT applicability and upon which all algorithmic optimisations are predicated. Although complex-symmetric level-three Toeplitz structure is of cardinal interest, many scientific applications are based on simpler Toeplitz systems, and thus, in order to maximise the usefulness of this work and help with the level-three theory, a quick discussion of the more rudimentary systems has been included, which can be viewed as concise treatments of the lower complexity systems or as mathematical stepping stones towards the full level-three Toeplitz system. The basic matrix structures, that of Toeplitz and circulant, along with their relationship to one another is introduced, and subsequently, the connection between circulant structure and discrete-cyclic convolution is developed for the most basic level-one Toeplitz system. The level-one theory is extended to level-two and level-three systems and the level-three theory is in turn used to describe the standard solution algorithm for level-three systems. Finally, the various improvements to the standard algorithm, which are possible due to the very modifications that were required in order to permit the use of DFTs, are detailed.

## 3.2 Toeplitz and circulant theory

The underlying kernel which serves as the initial construct for DFT applicability is known as Toeplitz structure [183]. A Toeplitz matrix of order $[n]$, denoted ${}^{[n]}\mathrm{T}$, is simply a matrix with constant diagonals, which can be expressed mathematically as ${}^{[n]}\mathrm{T}_{ij} = t_{j-i}$, $i, j = 0, \ldots, n-1$.

$$
{}^{[n]}\mathrm{T} = \begin{pmatrix} t_0 & t_1 & t_2 & \cdots & t_{n-2} & t_{n-1} \\ t_{-1} & t_0 & t_1 & \ddots & t_{n-3} & t_{n-2} \\ t_{-2} & t_{-1} & t_0 & \ddots & t_{n-4} & t_{n-3} \\ \vdots & \ddots & \ddots & \ddots & \ddots & \vdots \\ t_{-(n-2)} & \ddots & \ddots & \ddots & \ddots & t_1 \\ t_{-(n-1)} & t_{-(n-2)} & t_{-(n-3)} & \cdots & t_{-1} & t_0 \end{pmatrix} \tag{3.1}
$$

Likewise, a symmetric Toeplitz matrix of order $[n]$, denoted ${}^{[n]}\mathrm{Ts}$, has the form ${}^{[n]}\mathrm{Ts}_{ij} = t_{|j-i|}$, $i, j = 0, \ldots, n-1$.

$$
{}^{[n]}\mathrm{Ts} = \begin{pmatrix} t_0 & t_1 & t_2 & \cdots & t_{n-2} & t_{n-1} \\ t_1 & t_0 & t_1 & \ddots & t_{n-3} & t_{n-2} \\ t_2 & t_1 & t_0 & \ddots & t_{n-4} & t_{n-3} \\ \vdots & \ddots & \ddots & \ddots & \ddots & \vdots \\ t_{n-2} & \ddots & \ddots & \ddots & \ddots & t_1 \\ t_{n-1} & t_{n-2} & t_{n-3} & \cdots & t_1 & t_0 \end{pmatrix} \tag{3.2}
$$

In order to permit the use of DFTs in the multiplication of such Toeplitz matrices with arbitrary vectors, the Toeplitz matrix must be converted to a circulant matrix, a fact that will be shown explicitly in the next section. Circulant structure [205] is a special case of Toeplitz structure where the last element of each row is the same as first element of the next row. Explicitly, a circulant matrix of order $[n]$, denoted ${}^{[n]}\mathrm{C}$, has the form ${}^{[n]}\mathrm{C}_{ij} = c_{(j-i)\mathrm{mod}n}$, $i, j = 0, \ldots, n-1$.

$$
\begin{aligned}
{}^{[n]}\mathrm{C} &= circ\left(c_0, c_{n-1}, c_{n-2}, \ldots, c_1\right) && (3.3) \\
&= c_0\left[{}^{[n]}\psi^0\right] + c_1\left[{}^{[n]}\psi^1\right] + c_2\left[{}^{[n]}\psi^2\right] + \cdots + c_{n-1}\left[{}^{[n]}\psi^{n-1}\right] = \sum_{i=0}^{n-1} c_i\left[{}^{[n]}\psi^i\right] && (3.4) \\
&= \sum_{i=0}^{n-1} c_{(n-i)modn}\left[{}^{[n]}\pi^i\right] = \sum_{i=0}^{n-1} cr_i\left[{}^{[n]}\pi^i\right] = \sum_{i=0}^{n-1} cr_{(n-i)modn}\left[{}^{[n]}\psi^i\right]
\end{aligned}
$$

Here, ${}^{[n]}\psi$ and ${}^{[n]}\pi$, which are the fundamental building blocks for circulant structure, see Appendix A.1, are known as the downward-shift and forward-shift permutation matrices of order $[n]$, respectively. ${}^{[n]}\psi = [u_1|u_2|u_3|\ldots|u_{n-1}|u_0] = {}^{[n]}\pi^{\mathrm{T}} = [u_{n-1}|u_0|u_1|\ldots|u_{n-2}]^{\mathrm{T}}$, where $u_j$ is the $j^{th}$ column of the identity matrix ${}^{[n]}\mathbf{I}$. $c_i$ and $cr_i$, $i = 0\ldots n-1$, are the $i^{th}$ elements of the first column and first row of ${}^{[n]}\mathrm{C}$, respectively. Substituting for ${}^{[n]}\psi$ into Eq. 3.4, ${}^{[n]}\mathrm{C}$ becomes:

$$
\begin{aligned}
{}^{[n]}\mathrm{C} = \sum_{i=0}^{n-1} c_i \left[{}^{[n]}\psi^i\right] &= \sum_{i=0}^{n-1} c_i [u_1|u_2|\ldots|u_{n-1}|u_0]^i \\
&= \begin{pmatrix} c_0 & c_{n-1} & c_{n-2} & \cdots & c_2 & c_1 \\ c_1 & c_0 & c_{n-1} & \ddots & c_3 & c_2 \\ c_2 & c_1 & c_0 & \ddots & c_4 & c_3 \\ \vdots & \ddots & \ddots & \ddots & \ddots & \vdots \\ c_{n-2} & \ddots & \ddots & \ddots & \ddots & c_{n-1} \\ c_{n-1} & c_{n-2} & c_{n-3} & \cdots & c_1 & c_0 \end{pmatrix}
\end{aligned} \tag{3.5}
$$

Although this circulant structure may seem like only a negligible departure from that of Toeplitz structure, the benefits afforded by this humble transformation will be shown to be incontrovertible. For $n = 4$, the aforementioned forward and downward shift permutation matrices, ${}^{[n]}\pi$ and ${}^{[n]}\psi$, the circulant matrix ${}^{[n]}\mathrm{C}$ and the first column of the circulant matrix, ${}^{[n]}\mathrm{c}$, become:

$$
{}^{[4]}\psi = {}^{[4]}\pi^{\mathrm{T}} = \begin{pmatrix} 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix} \quad {}^{[4]}\mathrm{C} = \begin{pmatrix} c_0 & c_3 & c_2 & c_1 \\ c_1 & c_0 & c_3 & c_2 \\ c_2 & c_1 & c_0 & c_3 \\ c_3 & c_2 & c_1 & c_0 \end{pmatrix} \quad {}^{[4]}\mathrm{c} = \begin{pmatrix} c_0 \\ c_1 \\ c_2 \\ c_3 \end{pmatrix}
$$

A Toeplitz matrix, as defined in Eq. 3.1, is completely defined by its first row and first column, while a circulant matrix, as defined in Eqs. 3.3→3.5, is completely defined by its first column alone. Thus, to produce a circulant matrix, the first row and first column of the Toeplitz matrix must be combined into a single column, ${}^{[2n-1]}\mathrm{c}$, which will completely define the new circulant matrix. To convert a given Toeplitz matrix into a circulant matrix, the first column of the Toeplitz matrix, $t_{col} = (t_0\ t_{-1}\ \ldots\ t_{-(n-2)}\ t_{-(n-1)})^{\mathrm{T}}$, is taken and the entries of the first row, excluding the initial entry, in reverse order, $t_{row} = (t_{n-1}\ t_{n-2}\ \ldots\ t_2\ t_1)^{\mathrm{T}}$, are appended to $t_{col}$.

$$^{[2n-1]}\mathrm{C} = \begin{pmatrix} t_{col} \\ t_{row} \end{pmatrix} = (t_0 \;\; t_{-1} \;\; \ldots \;\; t_{-(n-2)} \;\; t_{-(n-1)} \;\; t_{n-1} \;\; t_{n-2} \;\; \ldots \;\; t_2 \;\; t_1)^{\mathrm{T}}$$

The conversion transforms the Toeplitz matrix, $^{[n]}\mathrm{T}$, of order $[n]$, into a circulant matrix, $^{[2n-1]}\mathrm{C}$, of order $[2n-1]$. For the symmetric case, the first column of the $^{[n]}\mathrm{Ts}$ contains all the required matrix entries to perform the conversion. In an analogous manner to the generic Toeplitz case, the first column of the symmetric Toeplitz matrix, $ts_{col} = (t_0 \; t_1 \; \ldots \; t_{n-2} \; t_{n-1})^{\mathrm{T}}$, is taken, but this time, instead of appending entries from the first row, the first column itself, excluding *both* the extreme entries, is mirrored across the last entry, appending $ts_{col'} = (t_{n-2} \; t_{n-3} \; \ldots \; t_2 \; t_1)^{\mathrm{T}}$ to $ts_{col}$.

$$^{[2n-2]}\mathrm{CS} = \begin{pmatrix} ts_{col} \\ ts_{col'} \end{pmatrix} = (t_0 \;\; t_1 \;\; \ldots \;\; t_{n-2} \;\; t_{n-1} \;\; t_{n-2} \;\; \ldots \;\; t_2 \;\; t_1)^{\mathrm{T}}$$

In this symmetric case, the conversion transforms the symmetric Toeplitz matrix, $^{[n]}\mathrm{Ts}$, of order $[n]$, into a symmetric circulant matrix, $^{[2n-2]}\mathrm{Cs}$, of order $[2n-2]$. A caveat on Toeplitz-to-circulant (T-to-c) conversion is that, if required, zeroes can be interjected between $t_{col}$ and $t_{row}$ *or* $ts_{col}$ and $ts_{col'}$ to increase the size of the circulant matrix to a size that permits the use of fast DFT algorithms, i.e., for FFTW [178], a dimension which satisfies $2^a 3^b 5^c 7^d 11^e 13^f$, where $a, b, c, d$ are arbitrary and $(e+f \leq 1)$.

## 3.3 Circulant diagonalisation

The following three subsections deal with the diagonalisation of a circulant matrix by the corresponding Fourier matrix. §3.3.1 introduces the concept for basic level-one circulant matrices and §3.3.2 & §3.3.3 extends the idea to circulant matrices of level-two and level-three, respectively.

### 3.3.1 Level-one diagonalisation

Although the T-to-c conversion may seem to have only served to exacerbate the computational requirements, the benefits afforded by this transformation become apparent by examining the eigen decomposition of a circulant matrix. In order to fully assimilate the significance of this eigen decomposition it is prudent to first introduce the Fourier matrix of order $[n]$, $^{[n]}\mathrm{F}$,

$$
{}^{[n]}\mathrm{F} = \frac{1}{\sqrt{n}}\omega^{pq} = \frac{1}{\sqrt{n}}\begin{pmatrix} 1 & 1 & 1 & 1 & \cdots & 1 \\ 1 & \omega & \omega^2 & \omega^3 & \ddots & \omega^{n-1} \\ 1 & \omega^2 & \omega^4 & \omega^6 & \ddots & \omega^{2(n-1)} \\ 1 & \omega^3 & \omega^6 & \omega^9 & \ddots & \omega^{3(n-1)} \\ \vdots & \ddots & \ddots & \ddots & \ddots & \vdots \\ 1 & \omega^{n-1} & \omega^{2(n-1)} & \omega^{3(n-1)} & \cdots & \omega^{(n-1)(n-1)} \end{pmatrix}, \tag{3.6}
$$

where $p, q = 0, \ldots, n-1$ and $\omega = e^{-\frac{2\pi\imath}{n}}$. Using the fact that the downward-shift permutation matrix of order $[n]$, ${}^{[n]}\psi$, is diagonalised by the Fourier matrix, ${}^{[n]}\mathrm{F}$, see Appendix A.2,

$$
\begin{aligned}
{}^{[n]}\Omega &= {}^{[n]}\mathrm{F}\,{}^{[n]}\psi\,{}^{[n]}\mathrm{F}^{\mathrm{H}} = diag\left(1, \omega, \omega^2, \omega^3, \ldots, \omega^{n-1}\right),\ \omega = e^{\frac{-2\pi\imath}{n}}, \\
\Rightarrow {}^{[n]}\psi^i &= {}^{[n]}\mathrm{F}^{\mathrm{H}}\left[{}^{[n]}\Omega^i\right]{}^{[n]}\mathrm{F}.
\end{aligned} \tag{3.7}
$$

This allows the definition of a circulant matrix in Eq. 3.4 to be reformulated into the left and right eigen-equations,

$$
{}^{[n]}\mathrm{F}\,{}^{[n]}\mathrm{C} = \left[\sum_{i=0}^{n-1} c_i\,{}^{[n]}\Omega^i\right]{}^{[n]}\mathrm{F} \quad \text{OR} \quad {}^{[n]}\mathrm{C}\,{}^{[n]}\mathrm{F}^{\mathrm{H}} = {}^{[n]}\mathrm{F}^{\mathrm{H}}\left[\sum_{i=0}^{n-1} c_i\,{}^{[n]}\Omega^i\right], \tag{3.8}
$$

and thus, the Fourier matrix, ${}^{[n]}\mathrm{F}$ and the Hermitian conjugate Fourier matrix, ${}^{[n]}\mathrm{F}^{\mathrm{H}}$, are identically the left and right eigen matrices of *all* circulant matrices. Furthermore, due to the fact that the Fourier matrix is complex-symmetric and unitary, ${}^{[n]}\mathrm{F}^{-1} = {}^{[n]}\mathrm{F}^{\mathrm{H}} \equiv \left[{}^{[n]}\mathrm{F}^{*}\right]^{\mathrm{T}} \equiv \left[{}^{[n]}\mathrm{F}^{\mathrm{T}}\right]^{*} \equiv {}^{[n]}\mathrm{F}^{*}$, and consequently, ${}^{[n]}\mathrm{F}^{*}$ and ${}^{[n]}\mathrm{F}^{-1}$ are also both right eigen matrices of *all* circulant matrices, where $*$ denotes the complex conjugate. Using this fact allows the trivial transposition of either of the aforementioned eigen equations, yielding the requisite diagonalisation of a level-one circulant system, where ${}^{[n]}\mathrm{c} = (c_0\ c_1\ c_2\ \ldots\ c_{n-2}\ c_{n-1})^{\mathrm{T}}$, is the first column of the circulant matrix ${}^{[n]}\mathrm{C}$.

$$
{}^{[n]}\mathrm{F}\,{}^{[n]}\mathrm{C}\,{}^{[n]}\mathrm{F}^{\mathrm{H}} = \left[\sum_{i=0}^{n-1} c_i\,{}^{[n]}\Omega^i\right] = diag\left(\sqrt{n}\left[{}^{[n]}\mathrm{F}\right]{}^{[n]}\mathrm{c}\right) = diag\left(\lambda_C\right) \tag{3.9}
$$

As a simple verification of Eq. 3.9, (for a formal treatment, see Appendix A.3),

for $n = 4$, the diagonalisation of a circulant matrix, ${}^{[4]}\mathrm{C}$, by the Fourier matrix, ${}^{[4]}\mathrm{F}$, can be shown symbolically as:

$$
\begin{aligned}
{}^{[4]}\mathrm{F}\,{}^{[4]}\mathrm{C}\,{}^{[4]}\mathrm{F}^{*} &= \frac{1}{\sqrt{n}}\begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & \omega & \omega^2 & \omega^3 \\ 1 & \omega^2 & \omega^4 & \omega^6 \\ 1 & \omega^3 & \omega^6 & \omega^9 \end{pmatrix}\begin{pmatrix} a & b & c & d \\ d & a & b & c \\ c & d & a & b \\ b & c & d & a \end{pmatrix}\frac{1}{\sqrt{n}}\begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & \omega & \omega^2 & \omega^3 \\ 1 & \omega^2 & \omega^4 & \omega^6 \\ 1 & \omega^3 & \omega^6 & \omega^9 \end{pmatrix}^{*} \\
&= \frac{1}{2}\begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & -i & -1 & i \\ 1 & -1 & 1 & -1 \\ 1 & i & -1 & -i \end{pmatrix}\begin{pmatrix} a & b & c & d \\ d & a & b & c \\ c & d & a & b \\ b & c & d & a \end{pmatrix}\frac{1}{2}\begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & -i & -1 & i \\ 1 & -1 & 1 & -1 \\ 1 & i & -1 & -i \end{pmatrix}^{*} \\
&= \begin{pmatrix} a+b+c+d & 0 & 0 & 0 \\ 0 & a-c+(b-d)i & 0 & 0 \\ 0 & 0 & a-b+c-d & 0 \\ 0 & 0 & 0 & a-c+(d-b)i \end{pmatrix}
\end{aligned}
$$

Using this diagonalisation of a circulant matrix by the Fourier matrix allows the product of a circulant matrix and arbitrary vector to be carried out in the Fourier Domain (FD),

$$
\begin{aligned}
{}^{[n]}\mathrm{C} &= {}^{[n]}\mathrm{F}^{-1} diag\left(\lambda_C\right){}^{[n]}\mathrm{F}, \\
{}^{[n]}\mathrm{y} = {}^{[n]}\mathrm{C}\,{}^{[n]}\mathrm{x} &= {}^{[n]}\mathrm{F}^{-1} diag\left(\lambda_C\right){}^{[n]}\mathrm{F}\,{}^{[n]}\mathrm{x} = {}^{[n]}\mathrm{F}^{-1}[\lambda_C].[{}^{[n]}\mathrm{F}\,{}^{[n]}\mathrm{x}],
\end{aligned} \tag{3.10}
$$

where the symbol '.' implies element-wise multiplication. This equation is a mathematical statement of the fact that the discrete-cyclic convolution, in explicit matrix form [206], is identically that of a circulant matrix multiplied with an arbitrary vector. Accordingly, the Convolution Theorem, $f \rightleftharpoons \mathrm{F}$, $g \rightleftharpoons \mathrm{G}$, $f * g \rightleftharpoons \mathrm{F.G}$, where the symbol '$*$' denotes convolution and '$\rightleftharpoons$' denotes a Fourier transform pair, can be employed to perform the multiplication of an arbitrary vector with a circulant matrix as an element-wise multiplication in the FD. Furthermore, it is clear from Eq. 3.9 that the circulant eigenvalues are a scalar multiple of the DFT of the first column, ${}^{[n]}\mathrm{c}$, of the circulant matrix, $\lambda_C = \sqrt{n}\left[{}^{[n]}\mathrm{F}\right]{}^{[n]}\mathrm{c}$. In the context of the practical application of the theory, this means that, not only does the theory reduce the computational complexity of the matrix-vector product from $O(n^2)$ to $O(n' \ln n')$, where $n' = 2n - 1$

for general Toeplitz matrices and $n' = 2n - 2$ for symmetric Toeplitz matrices, but it also *only* requires the first column of the circulant matrix be explicitly stored. This provides a *significant* reduction, from $O(n^2)$ to $O(n')$, in the memory required to carry out the matrix-vector product.

A noteworthy aside on circulant matrices is the fact that since the eigenvalues can be calculated with just the knowledge of the first column of the matrix alone, a host of useful operations can be efficiently performed on circulant matrices via the DFT of this first column. The determinant of a circulant can be calculated as the product of the eigenvalues, see Eq. 3.11. The product of two circulant matrices can be calculated using the fact that the eigenvalues of the product equal the product of the eigenvalues, see Eq. 3.12. Furthermore, for the inverse of a circulant matrix, the eigenvalues of the inverse are just the reciprocals of the eigenvalues of the circulant to be inverted, see Eq. 3.13.

**Determinant of a circulant matrix**

$$det\left[{}^{[n]}\mathrm{C}\right] = \prod_{i=0}^{n-1} \left(\lambda_C\right)_i, \lambda_C = \sqrt{n}\left[{}^{[n]}\mathrm{F}\right]{}^{[n]}\mathrm{c} \tag{3.11}$$

**Product of two circulant matrices**

$$\begin{aligned}
{}^{[n]}\mathrm{A} &= {}^{[n]}\mathrm{F}^{-1} diag\left(\lambda_A\right){}^{[n]}\mathrm{F}, \lambda_A = \sqrt{n}\left[{}^{[n]}\mathrm{F}\right]{}^{[n]}\mathrm{a} \\
{}^{[n]}\mathrm{B} &= {}^{[n]}\mathrm{F}^{-1} diag\left(\lambda_B\right){}^{[n]}\mathrm{F}, \lambda_B = \sqrt{n}\left[{}^{[n]}\mathrm{F}\right]{}^{[n]}\mathrm{b} \\
{}^{[n]}\mathrm{C} &= {}^{[n]}\mathrm{A}{}^{[n]}\mathrm{B} = {}^{[n]}\mathrm{F}^{-1} diag\left(\lambda_C\right){}^{[n]}\mathrm{F} = {}^{[n]}\mathrm{F}^{-1} diag\left(\lambda_A.\lambda_B\right){}^{[n]}\mathrm{F} \\
\lambda_C &= \sqrt{n}\left[{}^{[n]}\mathrm{F}\right]{}^{[n]}\mathrm{c} = \left(\lambda_A.\lambda_B\right) \Rightarrow {}^{[n]}\mathrm{c} = \frac{1}{\sqrt{n}}\left[{}^{[n]}\mathrm{F}^{-1}\right]\left(\lambda_A.\lambda_B\right)
\end{aligned} \tag{3.12}$$

**Inverse of a circulant matrix**

$$\begin{aligned}
{}^{[n]}\mathrm{D} &= {}^{[n]}\mathrm{F}^{-1} diag\left(\lambda_D\right){}^{[n]}\mathrm{F} = {}^{[n]}\mathrm{A}^{-1} = {}^{[n]}\mathrm{F}^{-1} diag\left(\frac{1}{\lambda_A}\right){}^{[n]}\mathrm{F} \\
\lambda_D &= \sqrt{n}\left[{}^{[n]}\mathrm{F}\right]{}^{[n]}\mathrm{d} = \left(\frac{1}{\lambda_A}\right) \Rightarrow {}^{[n]}\mathrm{d} = \frac{1}{\sqrt{n}}\left[{}^{[n]}\mathrm{F}^{-1}\right]\left(\frac{1}{\lambda_A}\right)
\end{aligned} \tag{3.13}$$

At this point it must be noted that Eq. 3.10 applies to circulant and *not* Toeplitz matrix-vector products. In order to use Eq. 3.10 to expedite the calculation of a Toeplitz matrix-vector product, for the general case, the Toeplitz matrix, ${}^{[n]}\mathrm{T}$, must be converted to its circulant counterpart, ${}^{[2n-1]}\mathrm{C}$, and the vector must be zero-padded to the dimension of the circulant matrix by appending $n$–1 zeroes. This zero-padding serves to negate the contribution from the $n$–1 extra columns that were created during the T-to-c conversion. In terms of the additional overheads that are associated with

the requisite T-to-c conversion of the matrix and the zero-padding of the vector to facilitate the convolution, both are $O(n)$ manipulations and are for all intents and purposes negligible.

The circulant matrix-vector product can be performed as the inverse Fourier Transform (iDFT) of the element-wise multiplication of a scalar multiple of the Fourier Transform of the first column of the circulant matrix and the Fourier Transform of the zero-padded vector. The first $n$ elements of the resultant are the required result for the original Toeplitz matrix-vector product, with the remaining $n-1$ entries being redundant. Note that for the symmetric Toeplitz case, the only difference is that the T-to-c conversion produces a circulant matrix of order $2n-2$ rather than $2n-1$.

### 3.3.2 Level-two diagonalisation

The basic level-one Toeplitz system is, for the most part, a mathematical entity and in order to treat realistic physical problems, more complicated and less idealised systems must be considered. The next level of complexity is a level-two Toeplitz system which is just a level-one system in which each element is itself a level-one system. Explicitly, a level-two Toeplitz matrix of order $[J][K]$, denoted ${}^{[J][K]}\mathrm{T}$, can be expressed as:

$$
{}^{[J][K]}\mathrm{T} = \begin{pmatrix}
{}^{[K]}\mathrm{T}_0 & {}^{[K]}\mathrm{T}_1 & {}^{[K]}\mathrm{T}_2 & \cdots & {}^{[K]}\mathrm{T}_{J-2} & {}^{[K]}\mathrm{T}_{J-1} \\
{}^{[K]}\mathrm{T}_{-1} & {}^{[K]}\mathrm{T}_0 & {}^{[K]}\mathrm{T}_1 & \ddots & {}^{[K]}\mathrm{T}_{J-3} & {}^{[K]}\mathrm{T}_{J-2} \\
{}^{[K]}\mathrm{T}_{-2} & {}^{[K]}\mathrm{T}_{-1} & {}^{[K]}\mathrm{T}_0 & \ddots & {}^{[K]}\mathrm{T}_{J-4} & {}^{[K]}\mathrm{T}_{J-3} \\
\vdots & \ddots & \ddots & \ddots & \ddots & \vdots \\
{}^{[K]}\mathrm{T}_{-(J-2)} & \ddots & \ddots & \ddots & \ddots & {}^{[K]}\mathrm{T}_1 \\
{}^{[K]}\mathrm{T}_{-(J-1)} & {}^{[K]}\mathrm{T}_{-(J-2)} & {}^{[K]}\mathrm{T}_{-(J-3)} & \cdots & {}^{[K]}\mathrm{T}_{-1} & {}^{[K]}\mathrm{T}_0
\end{pmatrix} \tag{3.14}
$$

Each of the $i = 0, \ldots, J-1$ independent entries, ${}^{[K]}\mathrm{T}_i$, is a simple Toeplitz matrix of order $[K]$, as defined by Eq. 3.1. Like the simple level-one T-to-c conversion, ${}^{[n]}\mathrm{T} \rightarrow {}^{[2n-1]}\mathrm{C}$, shown in §3.3.1, the conversion from a level-two Toeplitz of order $[J][K]$, ${}^{[J][K]}\mathrm{T}$, to level-two circulant of order $[J'][K'] = [2J-1][2K-1]$, ${}^{[J'][K']}\mathrm{C}$, is quite an elementary procedure requiring two steps. In an analogous manner to that of the level-one Toeplitz matrix, the first block column of the Toeplitz matrix, ${}^{[K]}\mathrm{T}_{col} = \left({}^{[K]}\mathrm{T}_0 \; {}^{[K]}\mathrm{T}_{-1} \cdots {}^{[K]}\mathrm{T}_{-(J-2)} \; {}^{[K]}\mathrm{T}_{-(J-1)}\right)^{\mathrm{T}}$, is taken and the entries of

the first block row, excluding the initial block entry, in reverse order, ${}^{[K]}\mathrm{T}_{row} = \left({}^{[K]}\mathrm{T}_{J-1}\ {}^{[K]}\mathrm{T}_{J-2} \cdots {}^{[K]}\mathrm{T}_2\ {}^{[K]}\mathrm{T}_1\right)^{\mathrm{T}}$, are appended to ${}^{[K]}\mathrm{T}_{col}$.

$$
\begin{aligned}
{}^{[J'][K]}\mathrm{C} &= \begin{pmatrix} {}^{[K]}\mathrm{T}_{col} \\ {}^{[K]}\mathrm{T}_{row} \end{pmatrix} \\
&= \left({}^{[K]}\mathrm{T}_0\ \ {}^{[K]}\mathrm{T}_{-1}\ \ \cdots\ \ {}^{[K]}\mathrm{T}_{-(J-1)}\ \ {}^{[K]}\mathrm{T}_{J-1}\ \ {}^{[K]}\mathrm{T}_{J-2}\ \ \cdots\ \ {}^{[K]}\mathrm{T}_2\ \ {}^{[K]}\mathrm{T}_1\right)^{\mathrm{T}}
\end{aligned}
$$

Subsequently, all that remains to complete the level-two Toeplitz, ${}^{[J][K]}\mathrm{T}$, to level-two circulant, ${}^{[J'][K']}\mathrm{C}$, conversion, is the task of converting the constituent level-one Toeplitz blocks of order $[K]$, ${}^{[K]}\mathrm{T}_i$, to their circulant counterparts, ${}^{[K']}\mathrm{C}_i$, of order $[K'] = [2K-1]$,

$$
{}^{[J'][K']}\mathrm{C} = \begin{pmatrix}
{}^{[K']}\mathrm{C}_0 & {}^{[K']}\mathrm{C}_{J'-1} & {}^{[K']}\mathrm{C}_{J'-2} & \cdots & {}^{[K']}\mathrm{C}_2 & {}^{[K']}\mathrm{C}_1 \\
{}^{[K']}\mathrm{C}_1 & {}^{[K']}\mathrm{C}_0 & {}^{[K']}\mathrm{C}_{J'-1} & \ddots & {}^{[K']}\mathrm{C}_3 & {}^{[K']}\mathrm{C}_2 \\
{}^{[K']}\mathrm{C}_2 & {}^{[K']}\mathrm{C}_1 & {}^{[K']}\mathrm{C}_0 & \ddots & {}^{[K']}\mathrm{C}_4 & {}^{[K']}\mathrm{C}_3 \\
\vdots & \ddots & \ddots & \ddots & \ddots & \vdots \\
{}^{[K']}\mathrm{C}_{J'-2} & \ddots & \ddots & \ddots & \ddots & {}^{[K']}\mathrm{C}_{J'-1} \\
{}^{[K']}\mathrm{C}_{J'-1} & {}^{[K']}\mathrm{C}_{J'-2} & {}^{[K']}\mathrm{C}_{J'-3} & \cdots & {}^{[K']}\mathrm{C}_1 & {}^{[K']}\mathrm{C}_0
\end{pmatrix}, \tag{3.15}
$$

where for $j = 0, \ldots, J'-1$, ${}^{[K']}\mathrm{C}_j = {}^{[K']}\mathrm{C}_{[J'-j] mod J'}$. Formally, this level-two circulant, ${}^{[J'][K']}\mathrm{C}$, of order $[J'][K'] = [2J-1][2K-1]$ in Eq. 3.15 has the form,

$$
{}^{[J'][K']}\mathrm{C} = \sum_{j=0}^{J'-1} {}^{[J']}\psi^j \otimes \left[{}^{[K']}\mathrm{C}_j\right], \tag{3.16}
$$

where ${}^{[J']}\psi$ is the downward-shift permutation matrix of order $[J']$, $\otimes$ denotes the Kronecker tensor product and each of the ${}^{[K']}\mathrm{C}_j$ is a level-one circulant of order $[K']$,

$$
{}^{[K']}\mathrm{C}_j = \sum_{i=0}^{K'-1} \mathrm{C}_{j,i} \left[{}^{[K']}\psi^i\right], \tag{3.17}
$$

where ${}^{[K']}\psi$ is the downward-shift permutation matrix of order $[K']$. A step-by-step example explaining the construction of such a level-two matrix is presented in Appendix A.4. For the simple level-one Toeplitz system discussed in §3.3.1, the crux of

the approach was that the level-one circulant system was diagonalised by the Fourier matrix, see Eqs. 3.9 & 3.10. So the key to the application of the approach to the prescribed level-two system is to identify the complimentary Fourier structure which diagonalises it. To this end, several delimiting observations can be made. The constituent circulant blocks, ${}^{[K']}\mathrm{C}_j$, $j = 0, \ldots, J' - 1$, in Eqs. 3.15 & 3.16, are simple level-one circulant matrices, and thus, each is diagonalised by the Fourier matrix of order $[K']$, ${}^{[K']}\mathrm{F}$. Furthermore, if each of these block elements of order $[K']$, ${}^{[K']}\mathrm{C}_j$, $j = 0, \ldots, J' - 1$, in Eq. 3.15 was just a single element, then the full matrix, ${}^{[J'][K']}\mathrm{C}$, of order $[J'][K']$, would simply be a level-one circulant, ${}^{[J']}\mathrm{C}$ of order $[J']$, and would be diagonalised by the Fourier matrix ${}^{[J']}\mathrm{F}$. This hierarchical diagonalisation by the Fourier matrices, ${}^{[J']}\mathrm{F}$ and ${}^{[K']}\mathrm{F}$, leads to the requisite diagonalising Fourier structure through their Kronecker tensor product, ${}^{[J'][K']}\mathrm{F} = {}^{[J']}\mathrm{F} \otimes {}^{[K']}\mathrm{F}$. This diagonalisation of the circulant matrix, ${}^{[J'][K']}\mathrm{C}$, by ${}^{[J'][K']}\mathrm{F}$ can be easily verified using Eqs. 3.16 & 3.17, see Appendix A.5.

Using the fact that a level-two circulant matrix, ${}^{[J'][K']}\mathrm{C}$, is diagonalised by the Fourier matrix, ${}^{[J'][K']}\mathrm{F}$, allows the level-two matrix-vector product, ${}^{[M']}\mathrm{y} = {}^{[J'][K']}\mathrm{C}{}^{[M']}\mathrm{x}$, where $[M'] = [J' \times K'] = [2J-1] \times [2K-1]$, to be implemented as:

$$
\begin{aligned}
{}^{[M']}\mathrm{y} = {}^{[J'][K']}\mathrm{C}{}^{[M']}\mathrm{x} &= {}^{[J'][K']}\mathrm{F}^{\mathrm{H}} \left[{}^{[J'][K']}\Lambda\right] {}^{[J'][K']}\mathrm{F}{}^{[M']}\mathrm{x} \\
&= {}^{[J'][K']}\mathrm{F}^{-1} \left[{}^{[J'][K']}\Lambda\right] {}^{[J'][K']}\mathrm{F}{}^{[M']}\mathrm{x} \\
&= {}^{[J'][K']}\mathrm{F}^{-1} diag\left(\sqrt{J'K'}\left[{}^{[J'][K']}\mathrm{F}\right] {}^{[J'][K']}\mathrm{c}\right) {}^{[J'][K']}\mathrm{F}{}^{[M']}\mathrm{x} \\
&= {}^{[J'][K']}\mathrm{F}^{-1} \left[\sqrt{J'K'}\left[{}^{[J'][K']}\mathrm{F}\right] {}^{[J'][K']}\mathrm{c}\right] . \left[{}^{[J'][K']}\mathrm{F}{}^{[M']}\mathrm{x}\right] \qquad (3.18)
\end{aligned}
$$

Furthermore, it is an opportune mathematical result that the Kronecker tensor product, ${}^{[J'][K']}\mathrm{F} = {}^{[J']}\mathrm{F} \otimes {}^{[K']}\mathrm{F}$, is equivalent to the 2D DFT. This fact allows Eq. 3.18 to be reformulated in terms of the 2D DFT.

$$
\begin{aligned}
{}^{[J'][K']}\mathrm{F}{}^{[J'][K']}\mathrm{c} \equiv \mathrm{Y}_{t,u} &= \frac{1}{\sqrt{J'K'}} \sum_{j=0}^{J'-1} \sum_{i=0}^{K'-1} \left[{}^{[J'][K']}\mathrm{c2D}_{i,j}\right] \exp\left[-2\pi\imath\left(t\frac{i}{K'} + u\frac{j}{J'}\right)\right] \\
{}^{[J'][K']}\mathrm{F}{}^{[M']}\mathrm{x} \equiv \mathrm{X}_{t,u} &= \frac{1}{\sqrt{J'K'}} \sum_{j=0}^{J'-1} \sum_{i=0}^{K'-1} \left[{}^{[M']}\mathrm{x2D}_{i,j}\right] \exp\left[-2\pi\imath\left(t\frac{i}{K'} + u\frac{j}{J'}\right)\right] \\
\mathrm{Q}_{t,u} &= \left[\sqrt{J'K'}\mathrm{Y}_{t,u}\right] . \left[\mathrm{X}_{t,u}\right]
\end{aligned}
$$

$$i = 0 \ldots K'-1$$
$$j = 0 \ldots J'-1$$

$$
{}^{[J'][K']}\mathrm{c} = \begin{pmatrix} {}^{[K']}\mathrm{c}_0 \\ {}^{[K']}\mathrm{c}_1 \\ {}^{[K']}\mathrm{c}_2 \\ \vdots \\ {}^{[K']}\mathrm{c}_{J'-2} \\ {}^{[K']}\mathrm{c}_{J'-1} \end{pmatrix} = \begin{pmatrix} \mathrm{c}_{0,0} \\ \mathrm{c}_{0,1} \\ \mathrm{c}_{0,2} \\ \vdots \\ \mathrm{c}_{0,K'-2} \\ \mathrm{c}_{0,K'-1} \\ \mathrm{c}_{1,0} \\ \mathrm{c}_{1,1} \\ \mathrm{c}_{1,2} \\ \vdots \\ \mathrm{c}_{1,K'-2} \\ \mathrm{c}_{1,K'-1} \\ \mathrm{c}_{2,0} \\ \mathrm{c}_{2,1} \\ \mathrm{c}_{2,2} \\ \vdots \\ \mathrm{c}_{2,K'-2} \\ \mathrm{c}_{2,K'-1} \\ \vdots \\ \mathrm{c}_{J'-2,0} \\ \mathrm{c}_{J'-2,1} \\ \mathrm{c}_{J'-2,2} \\ \vdots \\ \mathrm{c}_{J'-2,K'-2} \\ \mathrm{c}_{J'-2,K'-1} \\ \mathrm{c}_{J'-1,0} \\ \mathrm{c}_{J'-1,1} \\ \mathrm{c}_{J'-1,2} \\ \vdots \\ \mathrm{c}_{J'-1,K'-2} \\ \mathrm{c}_{J'-1,K'-1} \end{pmatrix}
$$

$$\uparrow$$
$${}^{[K']}\mathrm{c}_j$$

$$
{}^{[J'][K']}\mathrm{F}\,{}^{[J'][K']}\mathrm{c} = \left[{}^{[J']}\mathrm{F} \otimes {}^{[K']}\mathrm{F}\right] {}^{[J'][K']}\mathrm{c} \equiv \mathrm{Y}_{t,u}
$$

$$
\mathrm{Y}_{t,u} = \frac{1}{\sqrt{J'K'}} \sum_{j=0}^{J'-1} \sum_{i=0}^{K'-1} \left[{}^{[J'][K']}\mathrm{c2D}_{i,j}\right] \times \exp\left[-2\pi i \left(t\frac{i}{K'} + u\frac{j}{J'}\right)\right]
$$

$$t = 0 \ldots K'-1, u = 0 \ldots J'-1$$

$$\mathrm{c}_{j,i} \longleftarrow$$

$$
{}^{[J'][K']}\mathrm{c2D}_{i,j} = \mathrm{c}_{j,i} = \mathrm{c}_{jK'+i}
$$

$$
\rightarrow \begin{pmatrix} \mathrm{c}_{0,0} & \mathrm{c}_{1,0} & \mathrm{c}_{2,0} & \cdots & \mathrm{c}_{J'-2,0} & \mathrm{c}_{J'-1,0} \\ \mathrm{c}_{0,1} & \mathrm{c}_{1,1} & \mathrm{c}_{2,1} & \cdots & \mathrm{c}_{J'-2,1} & \mathrm{c}_{J'-1,1} \\ \mathrm{c}_{0,2} & \mathrm{c}_{1,2} & \mathrm{c}_{2,2} & \cdots & \mathrm{c}_{J'-2,2} & \mathrm{c}_{J'-1,2} \\ \vdots & \vdots & \vdots & \cdots & \vdots & \vdots \\ \mathrm{c}_{0,K'-2} & \mathrm{c}_{1,K'-2} & \mathrm{c}_{2,K'-2} & \cdots & \mathrm{c}_{J'-2,K'-2} & \mathrm{c}_{J'-1,K'-2} \\ \mathrm{c}_{0,K'-1} & \mathrm{c}_{1,K'-1} & \mathrm{c}_{2,K'-1} & \cdots & \mathrm{c}_{J'-2,K'-1} & \mathrm{c}_{J'-1,K'-1} \end{pmatrix}
$$

$$
= {}^{[J'][K']}\mathrm{c2D}
$$

$$
\mathrm{Y}_{t,u} = \left[{}^{[J'][K']}\mathrm{F}\,{}^{[J'][K']}\mathrm{c}\right]_{uK'+t}
$$

$$t = 0 \ldots K'-1, u = 0 \ldots J'-1$$

$$
\left[{}^{[J'][K']}\mathrm{F}\,{}^{[J'][K']}\mathrm{c}\right]_a = \mathrm{Y}_{t,u} = \mathrm{Y}_{a mod K', int\left(\frac{a}{K'}\right)}
$$

$$a = 0 \ldots [J'K'] - 1$$

Figure 3.1: Outlines the transformation of the matrix structure for ${}^{[J'][K']}\mathrm{c} \longrightarrow {}^{[J'][K']}\mathrm{c2D}$.

$$\begin{aligned} \mathrm{y}^{2\mathrm{D}}{}_{t,u} &= \frac{1}{\sqrt{J'K'}} \sum_{j=0}^{J'-1} \sum_{i=0}^{K'-1} [\mathrm{Q}_{i,j}] \exp\left[2\pi\imath\left(t\frac{i}{K'} + u\frac{j}{J'}\right)\right] \\ t &= 0,\ldots,K'-1, u = 0,\ldots,J'-1 \end{aligned} \tag{3.19}$$

Hence, the product of a level-two Toeplitz matrix, ${}^{[J][K]}\mathrm{T}$, and an arbitrary vector, ${}^{[M]}\mathrm{x}$, where, $M = [J \times K]$, can be performed as the 2D iDFT of the element-wise multiplication of the 2D DFT of the first column of the corresponding level-two circulant matrix, ${}^{[J'][K']}\mathrm{C}$, and the 2D DFT of the zero-padded vector, ${}^{[M']}\mathrm{x}$, in the FD. The required result for the original level-two Toeplitz matrix-vector product is then obtained by taking the resultant and discarding the last $K-1$ rows and $J-1$ columns. This reduces the computational complexity of the level-two Toeplitz matrix-vector product, ${}^{[M]}\mathrm{y} = {}^{[J][K]}\mathrm{T}{}^{[M]}\mathrm{x}$, from $O(M^2)$ to $O(M' \ln M')$ and storage from $O(M^2)$ to $O(M')$. Fig. 3.1 outlines the transformation of the matrix structure for ${}^{[J'][K']}\mathrm{c} \longrightarrow {}^{[J'][K']}\mathrm{c}^{2\mathrm{D}}$ to help elucidate the above reformulation.

### 3.3.3 Level-three diagonalisation

With the development of the level-two theory complete, it is a straight forward task to extend the theory to the treatment of level-three systems, ${}^{[N]}\mathrm{y} = {}^{[P][J][K]}\mathrm{T}{}^{[N]}\mathrm{x}$, $N = P \times J \times K$. The T-to-c conversion is performed on each of the three levels just as it was performed on each of the levels in the level-two system. The resulting level-three circulant, ${}^{[P'][J'][K']}\mathrm{C}$, may be expounded mathematically as,

$$ {}^{[P'][J'][K']}\mathrm{C} = \sum_{k=0}^{P'-1} \left[{}^{[P']}\psi^k\right] \otimes \left[{}^{[J'][K']}\mathrm{C}_k\right], \tag{3.20}$$

where each of the ${}^{[J'][K']}\mathrm{C}_k$ is a level-two circulant of order $[J'][K']$,

$$ {}^{[J'][K']}\mathrm{C}_k = \sum_{j=0}^{J'-1} \left[{}^{[J']}\psi^j\right] \otimes \left[{}^{[K']}\mathrm{C}_{k,j}\right], \tag{3.21}$$

where each of the ${}^{[K']}\mathrm{C}_{k,j}$ is a level-one circulant of order $[K']$,

$$
{}^{[K']}\mathrm{C}_{k,j} \quad = \quad \sum_{i=0}^{K'-1} \mathrm{C}_{k,j,i} \left[ {}^{[K']}\psi^{i} \right] . \tag{3.22}
$$

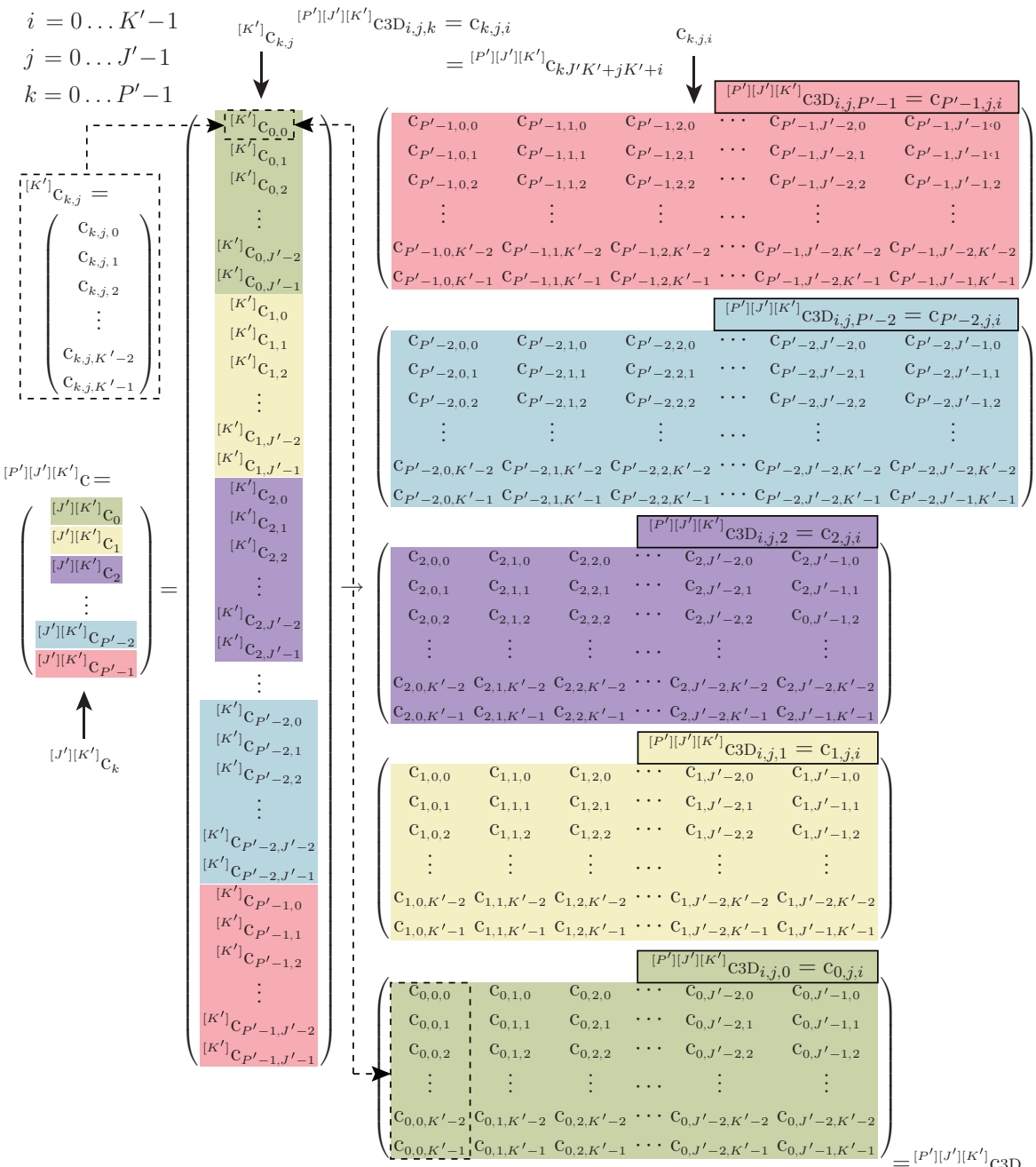
As with the level-two system, a step-by-step example explaining the construction of such a level-three matrix is presented in Appendix A.6. In an analogous manner to the level-two system, the level-three hierarchical diagonalisation by the Fourier matrices, ${}^{[P']}\mathrm{F}$, ${}^{[J']}\mathrm{F}$ and ${}^{[K']}\mathrm{F}$, leads to the identification of the complimentary diagonalising Fourier structure which is manifested through the Kronecker tensor product of these Fourier matrices, ${}^{[P'][J'][K']}\mathrm{F} = {}^{[P']}\mathrm{F} \otimes {}^{[J']}\mathrm{F} \otimes {}^{[K']}\mathrm{F}$. As with the level-two case, this diagonalisation of the level-three circulant matrix, ${}^{[P'][J'][K']}\mathrm{C}$, by ${}^{[P'][J'][K']}\mathrm{F}$ can be readily established using Eqs. 3.20→3.22, see Appendix A.7.

As a result of this diagonalisation, the level-three matrix-vector product, ${}^{[N']}\mathrm{y} = {}^{[P'][J'][K']}\mathrm{C}^{[N']}\mathrm{x}$, where $[N'] = [P' \times J' \times K'] = [2P-1] \times [2J-1] \times [2K-1]$, can be carried out as:

$$
\begin{aligned}
{}^{[N']}\mathrm{y} \quad &= \quad {}^{[P'][J'][K']}\mathrm{C}^{[N']}\mathrm{x} \\
&= \quad {}^{[P'][J'][K']}\mathrm{F}^{\mathrm{H}} \left[ {}^{[P'][J'][K']}\Lambda \right] {}^{[P'][J'][K']}\mathrm{F}^{[N']}\mathrm{x} \\
&= \quad {}^{[P'][J'][K']}\mathrm{F}^{-1} \left[ {}^{[P'][J'][K']}\Lambda \right] {}^{[P'][J'][K']}\mathrm{F}^{[N']}\mathrm{x} \\
&= \quad {}^{[P'][J'][K']}\mathrm{F}^{-1} diag \left( \sqrt{P'J'K'} \left[ {}^{[P'][J'][K']}\mathrm{F} \right] {}^{[P'][J'][K']}\mathrm{c} \right) {}^{[P'][J'][K']}\mathrm{F}^{[N']}\mathrm{x} \\
&= \quad {}^{[P'][J'][K']}\mathrm{F}^{-1} \left[ \sqrt{P'J'K'} \left[ {}^{[P'][J'][K']}\mathrm{F} \right] {}^{[P'][J'][K']}\mathrm{c} \right] . \left[ {}^{[P'][J'][K']}\mathrm{F}^{[N']}\mathrm{x} \right] .
\end{aligned} \tag{3.23}
$$

Furthermore, as with the level-two case, the Kronecker tensor product ${}^{[P'][J'][K']}\mathrm{F} = {}^{[P']}\mathrm{F} \otimes {}^{[J']}\mathrm{F} \otimes {}^{[K']}\mathrm{F}$ is equivalent to the 3D DFT.

$$
\begin{aligned}
&{}^{[P'][J'][K']}\mathrm{F}^{[P'][J'][K']}\mathrm{c} \equiv \mathrm{Y}_{t,u,v} \\
&= \frac{1}{\sqrt{P'J'K'}} \sum_{k=0}^{P'-1} \sum_{j=0}^{J'-1} \sum_{i=0}^{K'-1} \left[ {}^{[P'][J'][K']}\mathrm{c3D}_{i,j,k} \right] \exp \left[ -2\pi\imath \left( t\frac{i}{K'} + u\frac{j}{J'} + v\frac{k}{P'} \right) \right] \\
&{}^{[P'][J'][K']}\mathrm{F}^{[N']}\mathrm{x} \equiv \mathrm{X}_{t,u,v} \\
&= \frac{1}{\sqrt{P'J'K'}} \sum_{k=0}^{P'-1} \sum_{j=0}^{J'-1} \sum_{i=0}^{K'-1} \left[ {}^{[N']}\mathrm{x3D}_{i,j,k} \right] \exp \left[ -2\pi\imath \left( t\frac{i}{K'} + u\frac{j}{J'} + v\frac{k}{P'} \right) \right] \\
&\mathrm{Q}_{t,u,v} = \left[ \sqrt{P'J'K'}\mathrm{Y}_{t,u,v} \right] . \left[ \mathrm{X}_{t,u,v} \right]
\end{aligned}
$$




$$
\begin{aligned}
{}^{[P']}{}^{[J']}{}^{[K']}\mathrm{F}\,{}^{[P']}{}^{[J']}{}^{[K']}\mathrm{C} &= \left[{}^{[P']}\mathrm{F} \otimes {}^{[J']}\mathrm{F} \otimes {}^{[K']}\mathrm{F}\right] {}^{[P']}{}^{[J']}{}^{[K']}\mathrm{C} \equiv \mathrm{Y}_{t,u,v} \\
\mathrm{Y}_{t,u,v} &= \frac{1}{\sqrt{P'J'K'}} \sum_{k=0}^{P'-1} \sum_{j=0}^{J'-1} \sum_{i=0}^{K'-1} \left[{}^{[P']}{}^{[J']}{}^{[K']}\mathrm{C3D}_{i,j,k}\right] \exp\left[-2\pi i \left(t\frac{i}{K'} + u\frac{j}{J'} + v\frac{k}{P'}\right)\right] \\
t &= 0 \dots K'-1, u = 0 \dots J'-1, v = 0 \dots P'-1 \\
\mathrm{Y}_{t,u,v} &= \left[{}^{[P']}{}^{[J']}{}^{[K']}\mathrm{F}\,{}^{[P']}{}^{[J']}{}^{[K']}\mathrm{C}\right]_{vJ'K'+uK'+t} \\
\left[{}^{[P']}{}^{[J']}{}^{[K']}\mathrm{F}\,{}^{[P']}{}^{[J']}{}^{[K']}\mathrm{C}\right]_a &= \mathrm{Y}_{t,u,v} = \mathrm{Y}_{a mod K', int\left(\frac{a}{K'}\right) mod J', int\left(\frac{a}{J'K'}\right)} \\
a &= 0 \dots [P'J'K'] - 1
\end{aligned}
$$

Figure 3.2: Outlines the transformation of the matrix structure for ${}^{[P']}{}^{[J']}{}^{[K']}\mathrm{C} \rightarrow {}^{[P']}{}^{[J']}{}^{[K']}\mathrm{C3D}$.

$$\mathrm{y}^{3\mathrm{D}}{}_{t,u,v} = \frac{1}{\sqrt{P'J'K'}} \sum_{k=0}^{P'-1} \sum_{j=0}^{J'-1} \sum_{i=0}^{K'-1} [\mathrm{Q}_{i,j,k}] \exp\left[2\pi\imath\left(t\frac{i}{K'} + u\frac{j}{J'} + v\frac{k}{P'}\right)\right] \quad (3.24)$$
$$t = 0, \ldots, K'-1, u = 0, \ldots, J'-1, v = 0, \ldots, P'-1$$

Eqs. 3.23 & 3.24 are equivalent mathematical statements of the fact that the product of a level-three Toeplitz matrix, ${}^{[P][J][K]}\mathrm{T}$, and an arbitrary vector, ${}^{[N]}\mathrm{x}$, can be performed as the 3D iDFT of the element-wise multiplication of a scalar multiple of the 3D DFT of the first column of the corresponding level-three circulant matrix, ${}^{[P'][J'][K']}\mathrm{C}$, and the 3D DFT of the zero-padded vector, ${}^{[N']}\mathrm{x}$, in the FD. The required result for the original level-three Toeplitz matrix-vector product is then obtained by taking the resultant and extracting the $[0 \rightarrow K-1] \times [0 \rightarrow J-1] \times [0 \rightarrow P-1]$ quadrant. This reduces the computational complexity of the level-three Toeplitz matrix-vector product, ${}^{[N]}\mathrm{y} = {}^{[P][J][K]}\mathrm{T}{}^{[N]}\mathrm{x}$, from $O(N^2)$ to $O(N' \ln N')$ and storage from $O(N^2)$ to $O(N')$. Fig. 3.2 outlines the transformation of the matrix structure for ${}^{[P'][J'][K']}\mathrm{c} \longrightarrow {}^{[P'][J'][K']}\mathrm{c}_{3\mathrm{D}}$ in order to help elucidate this equivalence and elaborate the transformation of the matrix structure for the above reformulation.

## 3.4 The nature of the problem

**Although** the DDA is based on a 3D lattice, in its natural state, for arbitrary particle geometries, the system does not possess the necessary Toeplitz structure. However, the DDA system does exhibit the requisite Toeplitz structure for rectangular particles [207]. Thus, in order to obtain Toeplitz structure for systems based on particles of arbitrary geometry, the original particle with $N_d$ dipoles must be embedded in a $N = K \times J \times P$ lattice and the diagonal terms, $\mathbf{A}_{ii}$, neglected [13]. $K$, $J$ & $P$ are the $\hat{\mathbf{x}}$, $\hat{\mathbf{y}}$ & $\hat{\mathbf{z}}$ dimensions of the smallest cuboid that completely encompasses the particle. The reason for neglecting the diagonal terms is that the terms corresponding to unoccupied sites in the new extended lattice are zero, whereas the terms for occupied sites for the original diagonally dominant DDA system are non-zero and not necessarily equal, thus destroying the Toeplitz symmetry. By neglecting these terms the diagonal is identically zero yielding the constant diagonal required for Toeplitz structure. Once the Toeplitz symmetries have been exploited to perform the matrix-vector product (without these diagonal terms) it is straight forward to include the diagonal contributions. With these modifications, the structure of the interaction matrix A is that of a complex-symmetric level-three tensor-Toeplitz

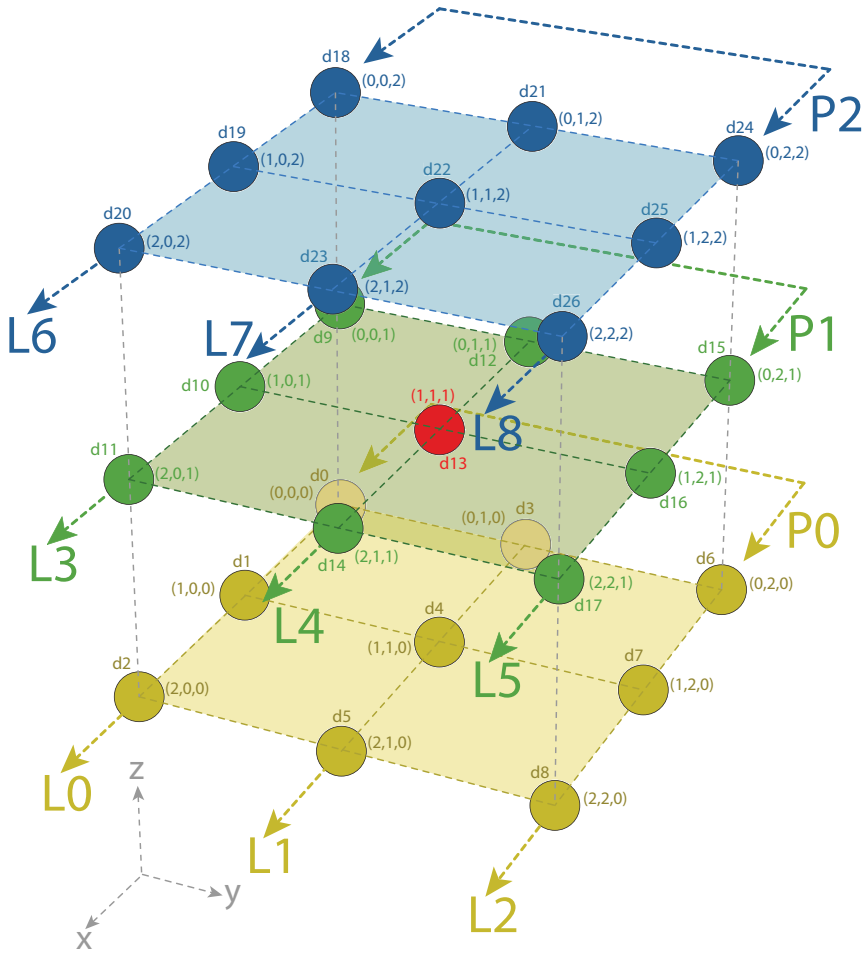



Figure 3.3: The exemplar $K = J = P = 3$ dipole array used to explain the nature of the DDA problem.

matrix of order $[P][J][K]$, denoted ${}^{[P][J][K]}\mathrm{T}$, in which the constituent tensor elements are complex-symmetric but not necessarily Toeplitz. The extra terms that have been added to the system from the extension from the $N_d$ dipoles in the original particle to the $N = K \times J \times P$ lattice can simply be negated by zeroing the corresponding entries in the matrix-vector product resultant within the iterative scheme.

The goal is to multiply this quite involved Toeplitz matrix with an arbitrary vector of length $N = P \times J \times K$, denoted ${}^{[N]}\mathrm{x}$, whose constituent elements are Cartesian vectors. The first task is to formalise the structure of the problem. Accordingly, Fig. 3.3 depicts the exemplar system that will be employed, as it is one of the most basic lattices which still exhibits and elucidates all the characteristics of the complete structure. As delineated by this figure, the lattice can be considered as an ensemble of $xy$-planes and lines oriented in the $\hat{\mathbf{x}}$ direction. Subsequently, the following definitions

can be adopted:

- $K$: Number of sites in the lines oriented in the $\hat{\mathbf{x}}$ direction, $[K=3]$
- $J$: Number of $\hat{\mathbf{x}}$ oriented lines in each $xy$-plane, $[J=M/K=3]$
- $P$: Number of $xy$-planes in the lattice, $[P=N/M=3]$
- $L$: Number of $\hat{\mathbf{x}}$ oriented lines in the lattice, $[L=N/K=J\times P=9]$
- $M$: Number of sites in each $xy$-plane, $[M=K\times J=9]$
- $Q$: Number of sites in each $xz$-plane, $[Q=K\times P=9]$
- $N$: Number of sites in the lattice, $[N=K\times J\times P=27]$

The interaction of the dipoles, situated on the $N = 27$ lattice sites of the cubic lattice in Fig. 3.3 produce a $N \times N$ complex-symmetric level-three tensor-Toeplitz interaction matrix. The fact that the constituent matrix elements are $3\times 3$ complex-symmetric tensors is not an issue. The complex-symmetric level-three tensor-Toeplitz matrix can be considered computationally as an assemblage of nine complex-symmetric level-three single-element Toeplitz matrices composed of the nine tensor components, $xx, xy, xz, yx, yy, yz, zx, zy, zz$. Furthermore, since the tensor elements are symmetric, $xy = yx, xz = zx$ & $yz = zy$, only six of the nine tensor components are independent and need to be considered explicitly, $xx, xy, xz, yy, yz, zz \equiv {}^{[P][J][K]}\mathrm{T0}$, ${}^{[P][J][K]}\mathrm{T1}$, ..., ${}^{[P][J][K]}\mathrm{T5}$. The structure of each of these complex-symmetric level-three single-element Toeplitz component matrices can be conceptualised as depicted in Fig. 3.4. Analogously, the arbitrary vector can be split computationally into three separate $x, y, z$ component vectors, ${}^{[N]}\mathrm{x0}$, ${}^{[N]}\mathrm{x1}$, ${}^{[N]}\mathrm{x2}$. Consequently, this means that the components can be treated separately and the full tensor/vector-element problem is reduced to an ensemble of smaller single-element problems whose contributions will be manifested ultimately through Fourier Domain tensor-vector products. For the tensor components, the three levels within these component matrices can be readily understood by examination of this diagram and can be envisaged as being derived from three levels of interaction. The first level, designated with a circumscribed '1', which encompasses the entire matrix, may be construed as a 'Plane' interaction level, i.e., in Fig. 3.4, [P0:P0] signifies the interaction between all the dipoles within plane P0, [P0:P1] signifies the interaction between the dipoles in plane P0 and the dipoles in plane P1, etc. The $P=3$ $xy$-planes, P0, P1, P2, interact, creating a $P\times P$ block Toeplitz matrix where the blocks are $M\times M=3\times 3$ blocks.

Each of these $M\times M$ blocks constitute the second level, designated with a circumscribed '2' and can be interpreted as a 'Line' interaction level. In an analogous

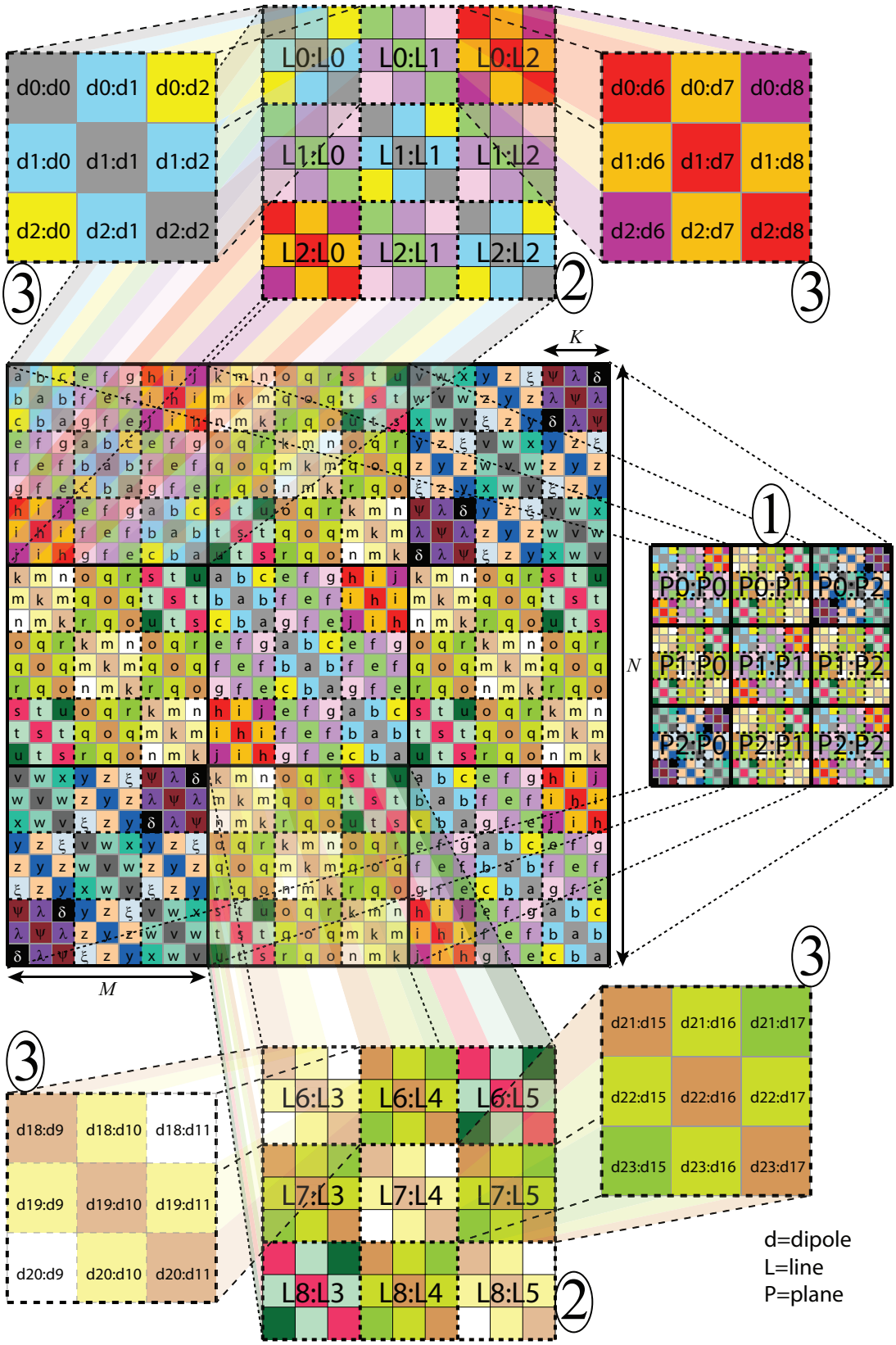



Figure 3.4: The structure of the nine component matrices of the full interaction matrix is that of a level-three Toeplitz matrix of order $[P=3][J=3][K=3]$, denoted ${}^{[3][3][3]}\mathrm{T}$.

manner to the first level, [L0 : L0] signifies the interaction between all the dipoles within line L0, [L6 : L4] signifies the interaction between the dipoles in line L6 and the dipoles in line L4, etc. In Fig. 3.4, two of these $M \times M$ blocks, [P0 : P0] and [P2 : P1], have been enlarged to illustrate their internal line structure. Concentrating on one $M \times M$ block, say, [P2 : P1], the $J = 3$ lines oriented in the $\hat{\mathbf{x}}$ direction in plane P2, [L6, L7, L8], interact with the $J = 3$ lines oriented in the $\hat{\mathbf{x}}$ direction in plane P1, [L3, L4, L5], creating a $J \times J$ block Toeplitz matrix where the blocks are $K \times K = 3 \times 3$ blocks. Each of these $K \times K$ blocks constitute the third level. This third and final level, designated with a circumscribed '3', can be considered to be a 'Dipole' interaction level. Here, [d21 : d16], for example, signifies the interaction between dipole d21 and dipole d16 and so on. In Fig. 3.4, two delineative $K \times K$ blocks have been extracted from both of the $M \times M$ blocks, [P0 : P0] and [P2 : P1], from earlier. As before, concentrating on an arbitrary block, say, the $K \times K$ block, [L7 : L5], the $K = 3$ constituent dipoles in line L7, [d21, d22, d23], interact with the $K = 3$ constituent dipoles in line L5, [d15, d16, d17], creating a $K \times K$ Toeplitz block.

A slight caveat on this structure is that since these nine tensor components are from the dyadic product in Eq. 1.2, there are sign considerations to be taken into account. Sign information for the three diagonal components, $xx, yy$ & $zz$, is irrelevant as they are squared quantities, and thus, their component matrices are complex-symmetric on all three levels. However, for the off-diagonal terms, the sign information leads to non-symmetries in sign only in the levels, a fact that will be important for the practical solution of the system. The $xy = yx$ component contains sign differences on the *lowest* $\equiv x \equiv$ *dipole* and *middle* $\equiv y \equiv$ *line* levels, the $xz = zx$ component on the *lowest* $\equiv x \equiv$ *dipole* and *highest* $\equiv z \equiv$ *plane* levels and the $yz = zy$ component on the *middle* $\equiv y \equiv$ *line* and *highest* $\equiv z \equiv$ *plane* levels.

## 3.5 Standard DFT algorithm

It was shown in §3.3.3 that the product of a level-three single-element Toeplitz matrix, ${}^{[P][J][K]}\mathrm{T}$, and an arbitrary vector, ${}^{[N]}\mathrm{x}$, can be performed as a 3D convolution using the first column of the corresponding level-three circulant matrix, ${}^{[P'][J'][K']}\mathrm{C}$, and the zero-padded vector, ${}^{[N']}\mathrm{x}$, see Eqs. 3.23 & 3.24. The aim, as explained in §3.4, is to multiply a level-three tensor-Toeplitz matrix, where the constituent tensor elements are complex-symmetric but not necessarily Toeplitz, with an arbitrary vector whose elements are Cartesian vectors. To this end, the six tensor

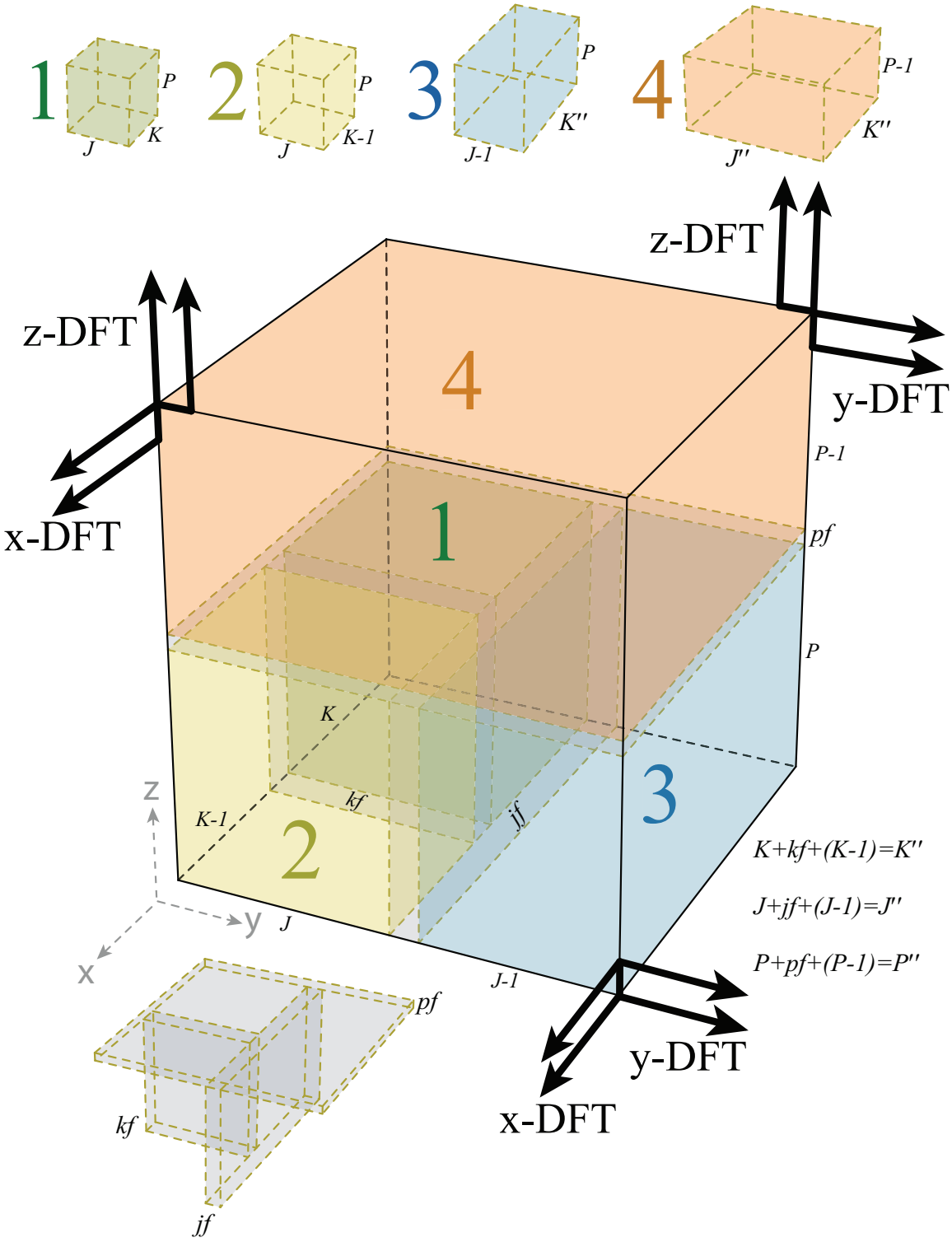



Figure 3.5: Outlines the T-to-c conversion of the six tensor component arrays, the zero-padding of the three vector component arrays and subsequent DFTs & iDFT for the original algorithm.

| | $xx$ | $xy$ | $xz$ | $yy$ | $yz$ | $zz$ |
|---|---|---|---|---|---|---|
| $x$ DFT | +1 | -1 | -1 | +1 | +1 | +1 |
| $y$ DFT | +1 | -1 | +1 | +1 | -1 | +1 |
| $z$ DFT | +1 | +1 | -1 | +1 | -1 | +1 |

Table 3.1: Since the nine tensor components are from the dyadic product in Eq. 1.2, there are sign considerations to be taken into account. This table lists the signs to account for the non-symmetries, in sign only, for the T-to-c conversions of the six independent tensor components.

components, $xx, xy, xz, yy, yz, zz$, of the complex level-three tensor-Toeplitz matrix that need to be considered explicitly and the three vector components, $x, y, z$, of the arbitrary vector are extracted producing six complex level-three single-element Toeplitz matrices, ${}^{[P][J][K]}\mathrm{T0}$, ${}^{[P][J][K]}\mathrm{T1}$, ..., ${}^{[P][J][K]}\mathrm{T5}$, and three complex vectors, ${}^{[N]}\mathrm{x0}$, ${}^{[N]}\mathrm{x1}$, ${}^{[N]}\mathrm{x2}$. Next, in order to allow the use of 3D DFTs, see Eq. 3.24, the first column of each of the six matrices and the three vector components must be arranged into separate 3D structures, ${}^{[P][J][K]}\mathrm{t0_{3D}}$, ${}^{[P][J][K]}\mathrm{t1_{3D}}$, ..., ${}^{[P][J][K]}\mathrm{t5_{3D}}$ and ${}^{[N]}\mathrm{x0_{3D}}$, ${}^{[N]}\mathrm{x1_{3D}}$, ${}^{[N]}\mathrm{x2_{3D}}$, see Fig. 3.2. To clarify this point, consider the transformations of the $xx$ component, ${}^{[P][J][K]}\mathrm{T0} \rightarrow {}^{[P][J][K]}\mathrm{t0_{3D}}$, and the $x$ component, ${}^{[N]}\mathrm{x0} \rightarrow {}^{[N]}\mathrm{x0_{3D}}$. The first column of the matrix ${}^{[P][J][K]}\mathrm{T0}$ can be expressed as,

$$
{}^{[P][J][K]}\mathrm{t0} = \begin{pmatrix} {}^{[J][K]}\mathrm{t0}_0 \\ {}^{[J][K]}\mathrm{t0}_1 \\ {}^{[J][K]}\mathrm{t0}_2 \\ \vdots \\ {}^{[J][K]}\mathrm{t0}_{P-2} \\ {}^{[J][K]}\mathrm{t0}_{P-1} \end{pmatrix}, \quad {}^{[J][K]}\mathrm{t0}_k = \begin{pmatrix} {}^{[K]}\mathrm{t0}_{k,0} \\ {}^{[K]}\mathrm{t0}_{k,1} \\ {}^{[K]}\mathrm{t0}_{k,2} \\ \vdots \\ {}^{[K]}\mathrm{t0}_{k,J-2} \\ {}^{[K]}\mathrm{t0}_{k,J-1} \end{pmatrix}, \quad \mathrm{t0}_{k,j} = \begin{pmatrix} \mathrm{t0}_{k,j,0} \\ \mathrm{t0}_{k,j,1} \\ \mathrm{t0}_{k,j,2} \\ \vdots \\ \mathrm{t0}_{k,j,K-2} \\ \mathrm{t0}_{k,j,K-1} \end{pmatrix},
$$

and the transformation ${}^{[P][J][K]}\mathrm{t0} \rightarrow {}^{[P][J][K]}\mathrm{t0_{3D}}$ is then governed by:

$$
\begin{aligned}
&{}^{[P][J][K]}\mathrm{t0_{3D}}_{i,j,k} = {}^{[P][J][K]}\mathrm{t0}_{kJK+jK+i} \\
&i = 0, \ldots, K-1, j = 0, \ldots, J-1, k = 0, \ldots, P-1
\end{aligned}
$$

Analogously, ${}^{[N]}\mathrm{x0} = \begin{pmatrix} \mathrm{x0}_0 & \mathrm{x0}_1 & \mathrm{x0}_2 & \cdots & \mathrm{x0}_{N-2} & \mathrm{x0}_{N-1} \end{pmatrix}^{\mathrm{T}}$, undergoes the transformation, ${}^{[N]}\mathrm{x0} \rightarrow {}^{[N]}\mathrm{x0_{3D}}$, governed by:

$$
\begin{aligned}
&{}^{[N]}\mathrm{x0}_{\mathrm{3D}_{i,j,k}} = {}^{[N]}\mathrm{x0}_{kJK+jK+i} \\
&i = 0,\ldots,K-1, j = 0,\ldots,J-1, k = 0,\ldots,P-1
\end{aligned}
$$

For DFT applicability, each of the six tensor component arrays must undergo T-to-c conversion and the three vector component arrays must be zero-padded. For a level-three system, the T-to-c conversion involves taking each tensor component array in turn, denoted by section '1' in Fig. 3.5, treating each line in the $\hat{\mathbf{x}}$, $\hat{\mathbf{y}}$ & $\hat{\mathbf{z}}$ directions as the first column of a level-one Toeplitz matrix and mirroring it, excluding the initial entry, across its last entry and multiplying the mirrored elements by the appropriate entry from Table 3.1. The reason for this is that for a general T-to-c conversion, one would usually append the first row, excluding the initial entry, in reverse order. However, here, the Toeplitz matrix is either symmetric or non-symmetric by sign only, so it is possible to just mirror the column itself and then multiply the mirrored entries by '$-1$' if necessary. The $\hat{\mathbf{x}}$ conversion creates '2', the $\hat{\mathbf{y}}$ conversion creates '3' and the $\hat{\mathbf{z}}$ conversion creates '4' in Fig. 3.5, i.e., for the $xx$ component, ${}^{[P][J][K]}\mathrm{t0}_{\mathrm{3D}} \rightarrow {}^{[P'][J'][K']}\mathrm{c0}_{\mathrm{3D}}$, the three steps are ${}^{[P][J][K]}\mathrm{t0}_{\mathrm{3D}} \rightarrow {}^{[P][J][K']}\mathrm{t0}_{\mathrm{3D}} \rightarrow {}^{[P][J'][K']}\mathrm{t0}_{\mathrm{3D}} \rightarrow {}^{[P'][J'][K']}\mathrm{c0}_{\mathrm{3D}}$, governed by:

$$
\begin{aligned}
{}^{[P][J][K']}\mathrm{t0}_{\mathrm{3D}_{i,j,k}} &= \begin{cases} {}^{[P][J][K]}\mathrm{t0}_{\mathrm{3D}_{i,j,k}} & i = 0,\ldots,K-1; \\ {}^{[P][J][K]}\mathrm{t0}_{\mathrm{3D}_{K'-i,j,k}} & i = K,\ldots,K'-1. \end{cases} \\
j &= 0,\ldots,J-1, k = 0,\ldots,P-1 \\
{}^{[P][J'][K']}\mathrm{t0}_{\mathrm{3D}_{i,j,k}} &= {}^{[P][J][K']}\mathrm{t0}_{\mathrm{3D}_{i,J'-j,k}} \\
i &= 0,\ldots,K'-1, j = J,\ldots,J'-1, k = 0,\ldots,P-1 \\
{}^{[P'][J'][K']}\mathrm{c0}_{\mathrm{3D}_{i,j,k}} &= {}^{[P][J'][K']}\mathrm{t0}_{\mathrm{3D}_{i,j,P'-k}} \\
i &= 0,\ldots,K'-1, j = 0,\ldots,J'-1, k = P,\ldots,P'-1
\end{aligned}
$$

The zero-padding of the three vector component arrays, i.e., ${}^{[N]}\mathrm{x0}_{\mathrm{3D}} \rightarrow {}^{[N']}\mathrm{x0}_{\mathrm{3D}}$, is much simpler. Each component is trivially embedded in the $[0 \rightarrow K-1] \times [0 \rightarrow J-1] \times [0 \rightarrow P-1]$ quadrant of a $K' \times J' \times P'$ array of zeroes, i.e., it would occupy section '1' in Fig. 3.5 and the rest of the array, '2', '3' & '4', would be identically zero. This zero-padding negates the contribution from the $K' \times J' \times P' - K \times J \times P$ extra elements that are produced during the T-to-c conversion. Implementing Eq. 3.24

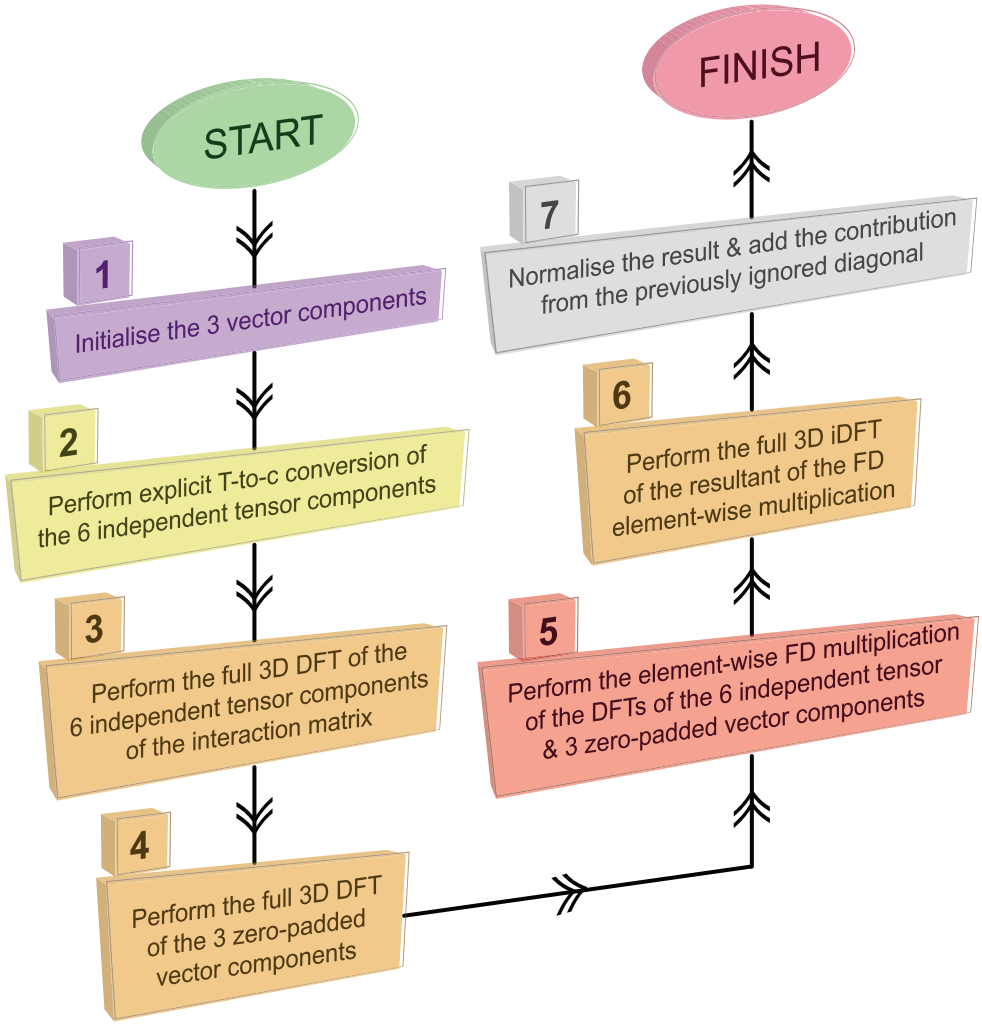



Figure 3.6: A flowchart for the DFT matrix-vector product as prescribed by the standard algorithm.

requires the DFT of the six tensor component arrays, ${}^{[P'][J'][K']}\mathrm{c0}_{\mathrm{3D}}$, ${}^{[P'][J'][K']}\mathrm{c1}_{\mathrm{3D}}$, ..., ${}^{[P'][J'][K']}\mathrm{c5}_{\mathrm{3D}}$, and the three vector component arrays, ${}^{[N']}\mathrm{x0}_{\mathrm{3D}}$, ${}^{[N']}\mathrm{x1}_{\mathrm{3D}}$ & ${}^{[N']}\mathrm{x2}_{\mathrm{3D}}$, yielding $\mathrm{Y1}, \mathrm{Y2}, \ldots, \mathrm{Y5}$ and $\mathrm{X0}, \mathrm{X1}, \mathrm{X2}$, i.e., for ${}^{[P'][J'][K']}\mathrm{c0}_{\mathrm{3D}}$ and ${}^{[N']}\mathrm{x0}_{\mathrm{3D}}$:

$$\mathrm{Y0}_{t,u,v} = \frac{1}{\sqrt{P'J'K'}} \sum_{k=0}^{P'-1} \sum_{j=0}^{J'-1} \sum_{i=0}^{K'-1} \left[ {}^{[P'][J'][K']}\mathrm{c0}_{\mathrm{3D}_{i,j,k}} \right] \times \exp\left[ -2\pi\imath \left( t\frac{i}{K'} + u\frac{j}{J'} + v\frac{k}{P'} \right) \right]$$

$$\begin{aligned}
\mathrm{X0}_{t,u,v} &= \frac{1}{\sqrt{P'J'K'}} \sum_{k=0}^{P'-1} \sum_{j=0}^{J'-1} \sum_{i=0}^{K'-1} \left[{}^{[N']}\mathrm{x0}_{\mathrm{3D}i,j,k}\right] \exp\left[-2\pi\imath\left(t\frac{i}{K'} + u\frac{j}{J'} + v\frac{k}{P'}\right)\right] \\
t &= 0,\ldots,K'-1, u=0,\ldots,J'-1, v=0,\ldots,P'-1
\end{aligned}$$

The same procedure for the other five tensor components and the remaining two vector components yields the needed Fourier Transforms, Y1, Y2, Y3, Y4 & Y5 and X1 & X2. Note that for the practical application of the theory, for maximum performance, as was mention in §3.2, if necessary, the transformations described above are all zero-padded to ensure that, as mentioned previously, the final dimensions of the component arrays permit the use of fast DFT algorithms, i.e., for FFTW [178], a dimension which satisfies $2^a3^b5^c7^d11^e13^f$, where $a, b, c, d$ are arbitrary and $(e+f \leq 1)$. These are illustrated in Figs. 3.5, 3.7 & 3.8 as $kf$, $jf$ and $pf$ and increase the dimensions to $K'' = K' + kf = 2K - 1 + kf$, $J'' = J' + jf = 2J - 1 + jf$ and $P'' = P' + pf = 2P - 1 + pf$. The contributions from these six tensor components and three vector components to the full circulant matrix-vector multiplication is then manifested as the element-wise tensor-vector product of these transforms.

$$\mathrm{Q0}_{t,u,v} = \sqrt{P'J'K'}\left(\mathrm{Y0}_{t,u,v}\mathrm{X0}_{t,u,v} + \mathrm{Y1}_{t,u,v}\mathrm{X1}_{t,u,v} + \mathrm{Y2}_{t,u,v}\mathrm{X2}_{t,u,v}\right) \quad (3.25)$$

$$\mathrm{Q1}_{t,u,v} = \sqrt{P'J'K'}\left(\mathrm{Y1}_{t,u,v}\mathrm{X0}_{t,u,v} + \mathrm{Y3}_{t,u,v}\mathrm{X1}_{t,u,v} + \mathrm{Y4}_{t,u,v}\mathrm{X2}_{t,u,v}\right) \quad (3.26)$$

$$\mathrm{Q2}_{t,u,v} = \sqrt{P'J'K'}\left(\mathrm{Y2}_{t,u,v}\mathrm{X0}_{t,u,v} + \mathrm{Y4}_{t,u,v}\mathrm{X1}_{t,u,v} + \mathrm{Y5}_{t,u,v}\mathrm{X2}_{t,u,v}\right) \quad (3.27)$$

$$t = 0,\ldots,K'-1, u=0,\ldots,J'-1, v=0,\ldots,P'-1$$

The inverse DFT (iDFT) of Q0, Q1 & Q2 provides y0$_{\mathrm{3D}}$, y1$_{\mathrm{3D}}$ & y2$_{\mathrm{3D}}$, which contain the requisite $x$, $y$ & $z$ component vectors of the resultant of the matrix-vector product of the complex level-three tensor-circulant matrix, ${}^{[P'][J'][K']}\mathrm{C}$, and the zero-padded arbitrary complex vector, ${}^{[N']}\mathrm{x}$, whose elements are Cartesian vectors, i.e., for Q0:

$$\begin{aligned}
\mathrm{y0}_{\mathrm{3D}t,u,v} &= \frac{1}{\sqrt{P'J'K'}} \sum_{k=0}^{P'-1} \sum_{j=0}^{J'-1} \sum_{i=0}^{K'-1} \left[\mathrm{Q0}_{i,j,k}\right] \exp\left[2\pi\imath\left(t\frac{i}{K'} + u\frac{j}{J'} + v\frac{k}{P'}\right)\right] \\
t &= 0,\ldots,K'-1, u=0,\ldots,J'-1, v=0,\ldots,P'-1
\end{aligned}$$

The $x, y, z$ components of the required answer for the original tensor-Toeplitz matrix-vector product, ${}^{[N]}\mathrm{y} = {}^{[P][J][K]}\mathrm{T}{}^{[N]}\mathrm{x}$, are obtained by extracting the $[0 \rightarrow K-$

$1] \times [0 \rightarrow J-1] \times [0 \rightarrow P-1]$ quadrant of y0$_{3D}$, y1$_{3D}$ and y2$_{3D}$, respectively, i.e., '1' in Fig. 3.5. The other elements in each array are redundant and can be neglected. This reduces the complexity from $O([3N_d]^2)$ operations and storage to $O(12N' \ln N')$ and $O(9N')$, respectively. A caveat on this method is that, if using the FFTW [178] to compute the DFTS, the FFTW library does not normalise the transforms. This means that the eigenvalues are identically the Fourier Transform of the first column of the circulant matrix, ${}^{[P'][J'][K']}\Lambda = diag\left(\left[{}^{[P'][J'][K']}\mathrm{F}\right] {}^{[P'][J'][K']}\mathrm{c}\right)$, and the scaling factor in Eqs. 3.25→3.27 is not required. However, the final results must be normalised.

$$
\begin{aligned}
\mathrm{Y0}_{t,u,v} &= \sum_{k=0}^{P'-1}\sum_{j=0}^{J'-1}\sum_{i=0}^{K'-1}\left[{}^{[P'][J'][K']}\mathrm{c0}_{3D\,i,j,k}\right]\exp\left[-2\pi\imath\left(t\frac{i}{K'}+u\frac{j}{J'}+v\frac{k}{P'}\right)\right] \\
\mathrm{X0}_{t,u,v} &= \sum_{k=0}^{P'-1}\sum_{j=0}^{J'-1}\sum_{i=0}^{K'-1}\left[{}^{[N']}\mathrm{x0}_{3D\,i,j,k}\right]\exp\left[-2\pi\imath\left(t\frac{i}{K'}+u\frac{j}{J'}+v\frac{k}{P'}\right)\right] \\
\mathrm{Q0}_{t,u,v} &= \mathrm{Y0}_{t,u,v}\mathrm{X0}_{t,u,v}+\mathrm{Y1}_{t,u,v}\mathrm{X1}_{t,u,v}+\mathrm{Y2}_{t,u,v}\mathrm{X2}_{t,u,v} && (3.28)\\
\mathrm{Q1}_{t,u,v} &= \mathrm{Y1}_{t,u,v}\mathrm{X0}_{t,u,v}+\mathrm{Y3}_{t,u,v}\mathrm{X1}_{t,u,v}+\mathrm{Y4}_{t,u,v}\mathrm{X2}_{t,u,v} && (3.29)\\
\mathrm{Q2}_{t,u,v} &= \mathrm{Y2}_{t,u,v}\mathrm{X0}_{t,u,v}+\mathrm{Y4}_{t,u,v}\mathrm{X1}_{t,u,v}+\mathrm{Y5}_{t,u,v}\mathrm{X2}_{t,u,v} && (3.30)\\
\mathrm{y0}_{3D\,t,u,v} &= \frac{1}{N'}\sum_{k=0}^{P'-1}\sum_{j=0}^{J'-1}\sum_{i=0}^{K'-1}\left[\mathrm{Q0}_{i,j,k}\right]\exp\left[2\pi\imath\left(t\frac{i}{K'}+u\frac{j}{J'}+v\frac{k}{P'}\right)\right] \\
N' &= [P'\times J'\times K'] = [2P-1]\times[2J-1]\times[2K-1] \\
t &= 0,\ldots,K'-1, u=0,\ldots,J'-1, v=0,\ldots,P'-1
\end{aligned}
$$

To aid visualisation and help explicate the structure of the prescribed algorithm, Fig. 3.6 displays a complete flowchart for the DFT matrix-vector product illustrating the various components that make up the algorithm. This algorithm constitutes the standard algorithm upon which all improvements are predicated. The three vector components are zero-padded by embedding each component in a larger array of zeroes and the six independent tensor components undergo explicit T-to-c conversion. The full 3D DFT is then applied to each of the vector and tensor components. The transforms are then element-wise multiplied and the full 3D iDFT is applied to each of the components of the resultant. Finally, the components are normalised and the contribution from the previously ignored diagonal is included.

## 3.6 Algorithmic optimisation

In order to permit the use of DFTs, the standard algorithm requires the T-to-c conversion of six tensor component arrays, ${}^{[P][J][K]}\mathrm{t0_{3D}} \rightarrow {}^{[P][J][K]}\mathrm{t5_{3D}}$, and the zero-padding of three vector component arrays, ${}^{[N]}\mathrm{x0_{3D}} \rightarrow {}^{[N]}\mathrm{x2_{3D}}$. Subsequently, the standard algorithm performs nine forward DFTs, FD element-wise multiplication of the transformed tensor and vector components, and finally, three iDFTs. Due to the T-to-c conversions that are required for DFT applicability, the six tensor components arrays contain a high degree of symmetry, i.e., for ${}^{[P'][J'][K']}\mathrm{c0_{3D}}$, the lines in the $\hat{\mathbf{x}}$ direction can be defined as ${}^{[K']}\mathrm{c0_{3D}}_{j,k}, j = 0, \ldots, J'-1, k = 0, \ldots, P'-1$, in the $\hat{\mathbf{y}}$ direction as ${}^{[J']}\mathrm{c0_{3D}}_{i,k}, i = 0, \ldots, K'-1, k = 0, \ldots, P'-1$, and in the $\hat{\mathbf{z}}$ direction as ${}^{[P']}\mathrm{c0_{3D}}_{i,j}, i = 0, \ldots, K'-1, j = 0, \ldots, J'-1$, and the symmetry as:

$$
{}^{[K']}\mathrm{c0_{3D}}_{j,k} = \begin{pmatrix} \mathrm{c0_{3D}}_{0,j,k} \\ \mathrm{c0_{3D}}_{1,j,k} \\ \mathrm{c0_{3D}}_{2,j,k} \\ \vdots \\ \mathrm{c0_{3D}}_{K'-2,j,k} \\ \mathrm{c0_{3D}}_{K'-1,j,k} \end{pmatrix} \qquad \begin{aligned} & j = 0, \ldots, J'-1, k = 0, \ldots, P'-1 \\ & j = \begin{cases} j, & \text{if } j < J; \\ J'-j, & \text{if } j \geq J. \end{cases} \\ & k = \begin{cases} k, & \text{if } k < P; \\ P'-k, & \text{if } k \geq P. \end{cases} \end{aligned}
$$

$$
{}^{[J']}\mathrm{c0_{3D}}_{i,k} = \begin{pmatrix} \mathrm{c0_{3D}}_{i,0,k} \\ \mathrm{c0_{3D}}_{i,1,k} \\ \mathrm{c0_{3D}}_{i,2,k} \\ \vdots \\ \mathrm{c0_{3D}}_{i,J'-2,k} \\ \mathrm{c0_{3D}}_{i,J'-1,k} \end{pmatrix} \qquad \begin{aligned} & i = 0, \ldots, K'-1, k = 0, \ldots, P'-1 \\ & i = \begin{cases} i, & \text{if } i < K; \\ K'-i, & \text{if } i \geq K. \end{cases} \\ & k = \begin{cases} k, & \text{if } k < P; \\ P'-k, & \text{if } k \geq P. \end{cases} \end{aligned}
$$

$$
{}^{[P']}\mathrm{c0_{3D}}_{i,j} = \begin{pmatrix} \mathrm{c0_{3D}}_{i,j,0} \\ \mathrm{c0_{3D}}_{i,j,1} \\ \mathrm{c0_{3D}}_{i,j,2} \\ \vdots \\ \mathrm{c0_{3D}}_{i,j,P'-2} \\ \mathrm{c0_{3D}}_{i,j,P'-1} \end{pmatrix} \qquad \begin{aligned} & i = 0, \ldots, K'-1, j = 0, \ldots, J'-1 \\ & i = \begin{cases} i, & \text{if } i < K; \\ K'-i, & \text{if } i \geq K. \end{cases} \\ & j = \begin{cases} j, & \text{if } j < J; \\ J'-j, & \text{if } j \geq J. \end{cases} \end{aligned}
$$

The prescribed scheme makes no assumptions in regard to the structure of the six tensor components arrays, carrying out the T-to-c conversion and applying the full 3D DFT to the resultant as depicted in Fig. 3.5. Since a $K' \times J' \times P'$ 3D DFT can be carried out as an ensemble of 1D DFTs, $J' \times P'$ in the $\hat{\mathbf{x}}$, $K' \times P'$ in the $\hat{\mathbf{y}}$ and $K' \times J'$ in

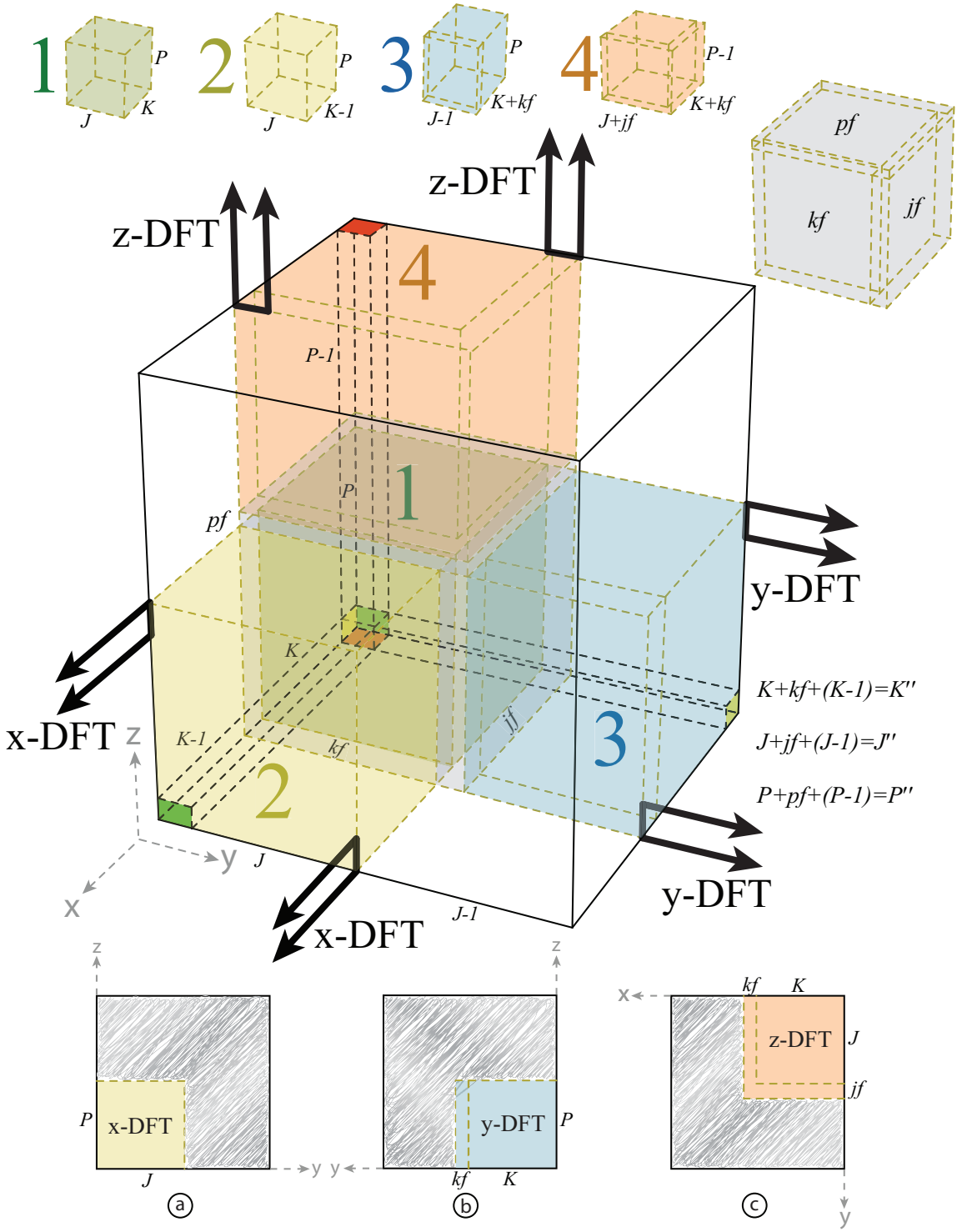



Figure 3.7: Outlines the implicit T-to-c conversion and simultaneous DFT of the tensor component arrays as implemented in the new algorithm.

the $\hat{\mathbf{z}}$ direction, a new algorithm which exploits these symmetries can be constructed. Rather than expanding the six tensor component arrays from a $K \times J \times P$ Toeplitz structure to a $K' \times J' \times P'$ circulant structure and applying 3D DFTs, the T-to-c conversion and subsequent DFT can be amalgamated into a single computationally efficient operation. Each of the lines in the $\hat{\mathbf{x}}$ direction, ${}^{[K]}\mathrm{t0}_{3\mathrm{D}_{j,k}}, j = 0, \ldots, J-1, k = 0, \ldots, P-1$, in the $\hat{\mathbf{y}}$ direction, ${}^{[J]}\mathrm{t0}_{3\mathrm{D}_{i,k}}, i = 0, \ldots, K-1, k = 0, \ldots, P-1$, and in the $\hat{\mathbf{z}}$ direction, ${}^{[P]}\mathrm{t0}_{3\mathrm{D}_{i,j}}, i = 0, \ldots, K-1, j = 0, \ldots, J-1$, can be taken in turn and copied into an external 1D array of length $K' = 2K-1$, $J' = 2J-1$ and $P' = 2P-1$ respectively. The $K$, $J$ and $P$ copied elements can be treated as the first column of a level-one Toeplitz matrix and the T-to-c conversion performed by mirroring the elements, excluding the initial entry, across the last entry and multiplying the mirrored elements by the appropriate entry from Table 3.1. The DFTs of these external vectors can be computed and subsequently, the first $K$, $J$ and $P$ elements of the resultants can be copied back into the 3D tensor component array overwriting the original data. This drastically reduces the number of DFTs that have to be performed. Taking into account the zero-padding to ensure $K'' = K'+kf = 2K-1+kf$, $J'' = J'+jf = 2J-1+jf$ and $P'' = P'+pf = 2P-1+pf$ to permit the use of fast DFT algorithms, in the $\hat{\mathbf{x}}$ **direction**, see Fig. 3.7a, $\hat{\mathbf{y}}$ **direction**, see Fig. 3.7b & $\hat{\mathbf{z}}$ **direction**, see Fig. 3.7c, the number of transforms is reduced from $6\,[J'' \times P''] \rightarrow 6\,[J \times P]$, $6\,[K'' \times P''] \rightarrow 6\,[(K+kf) \times P]$ and $6\,[K'' \times J''] \rightarrow 6\,[(K+kf) \times (J+jf)]$, respectively, which is a reduction of approximately 75%. Furthermore, since the full T-to-c conversion is never explicitly performed, the storage required for the six tensor component arrays is reduced from $6\,[K'' \times J'' \times P''] \rightarrow 6\,[(K+kf) \times (J+jf) \times (P+pf)]$ complex numbers, which is a reduction of approximately 88%. An ancillary computational benefit is that the scheme only requires the creation of three 1D DFT plans for the three external vectors that are used to compute the $\hat{\mathbf{x}}$, $\hat{\mathbf{y}}$ and $\hat{\mathbf{z}}$ DFTs. Thus, if using the FFTW library [178] to compute the DFTs, substantial time improvements can be obtained, especially if using FFTW_MEASURE, which instructs FFTW to run and measure the execution time of several DFTs in order to find the best way to compute the transform.

For the three zero-padded vector component arrays, the realisation that $\approx \frac{7}{8}$ of each is by definition, identically zero, can be used to expedite the calculations. For the ${}^{[K']}\mathrm{x0}_{3\mathrm{D}}$ case, the lines in the $\hat{\mathbf{x}}$, $\hat{\mathbf{y}}$ & $\hat{\mathbf{z}}$ directions can be defined as:

$$
{}^{[K']}\mathrm{x0}_{3\mathrm{D}_{j,k}} = \begin{pmatrix} \mathrm{x0}_{3\mathrm{D}_{0,j,k}} \\ \mathrm{x0}_{3\mathrm{D}_{1,j,k}} \\ \mathrm{x0}_{3\mathrm{D}_{2,j,k}} \\ \vdots \\ \mathrm{x0}_{3\mathrm{D}_{K'-2,j,k}} \\ \mathrm{x0}_{3\mathrm{D}_{K'-1,j,k}} \end{pmatrix} \qquad j = 0, \ldots, J'-1, k = 0, \ldots, P'-1
$$

$$
{}^{[J']}\mathrm{x0}_{3\mathrm{D}_{i,k}} = \begin{pmatrix} \mathrm{x0}_{3\mathrm{D}_{i,0,k}} \\ \mathrm{x0}_{3\mathrm{D}_{i,1,k}} \\ \mathrm{x0}_{3\mathrm{D}_{i,2,k}} \\ \vdots \\ \mathrm{x0}_{3\mathrm{D}_{i,J'-2,k}} \\ \mathrm{x0}_{3\mathrm{D}_{i,J'-1,k}} \end{pmatrix} \qquad i = 0, \ldots, K'-1, k = 0, \ldots, P'-1
$$

$$
{}^{[P']}\mathrm{x0}_{3\mathrm{D}_{i,j}} = \begin{pmatrix} \mathrm{x0}_{3\mathrm{D}_{i,j,0}} \\ \mathrm{x0}_{3\mathrm{D}_{i,j,1}} \\ \mathrm{x0}_{3\mathrm{D}_{i,j,2}} \\ \vdots \\ \mathrm{x0}_{3\mathrm{D}_{i,j,P'-2}} \\ \mathrm{x0}_{3\mathrm{D}_{i,j,P'-1}} \end{pmatrix} \qquad i = 0, \ldots, K'-1, j = 0, \ldots, J'-1
$$

For computational efficiency, the DFTs of the three zero-padded vector component arrays, the subsequent element-wise FD multiplication of the tensor and vector components, and the iDFTs of the resultant are amalgamated into a conjoint operation. Unlike the tensor component DFTs which are performed in the $\hat{\mathbf{x}}$ direction, the $\hat{\mathbf{y}}$ direction and finally the $\hat{\mathbf{z}}$ direction, the DFT of the vector components begin with the $\hat{\mathbf{y}}$ direction, then the $\hat{\mathbf{x}}$ direction, and lastly, the $\hat{\mathbf{z}}$ direction, see Fig. 3.8, and the iDFT of the resultant of the FD multiplication is performed in the $\hat{\mathbf{z}}$, $\hat{\mathbf{x}}$ and $\hat{\mathbf{y}}$ directions. The reason for this is twofold. Firstly, it allows the use of a $xz$-scratch plane for the DFTs in the $\hat{\mathbf{x}}$ & $\hat{\mathbf{z}}$ directions, the FD multiplication of the tensor and vector components and the iDFTs of the resultant in the $\hat{\mathbf{x}}$ & $\hat{\mathbf{z}}$ directions, which provides large reductions in the memory required. Secondly, it supports the distributed FD multiplication in the MPI version of the algorithm, which will be discussed in detail in §4.3. As a result of this optimisation, the storage for the three vector components, ignoring the storage required for the three $xz$-scratch planes, is reduced from $3\,[K'' \times J'' \times P''] \rightarrow 3\,[K \times J'' \times P]$ complex numbers, which is a reduction

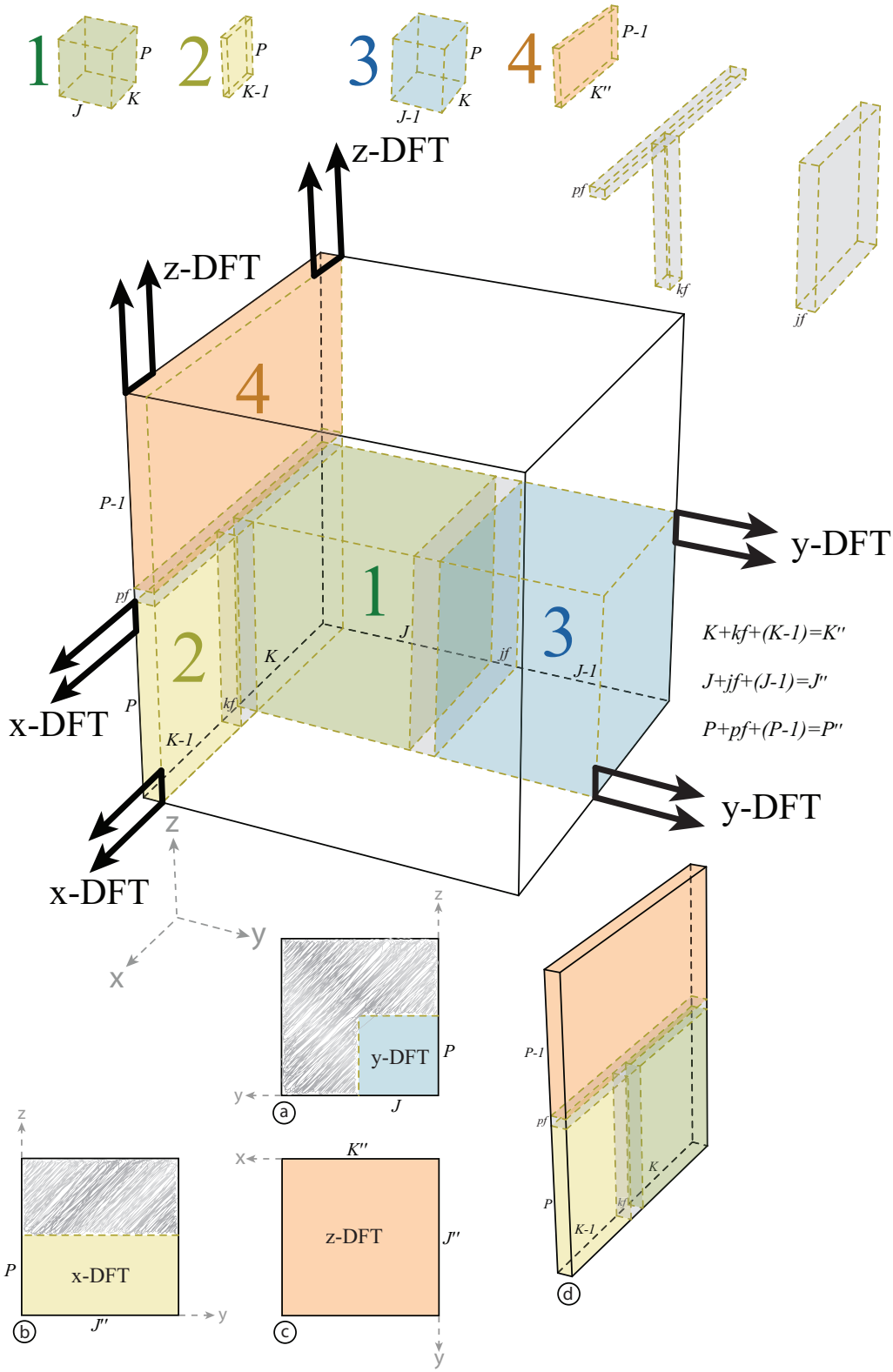



Figure 3.8: Outlines the zero-padding and the subsequent optimised DFT of the three vector component arrays and also helps to envisage the iDFT where only transforms that contribute to the final requisite answer are performed.

of approximately 75%.

The new optimised algorithm proceeds as follows, firstly, each of the three vector components is embedded into a zero-padded $[K \times J'' \times P]$ array. This is illustrated in Fig. 3.8 where section '1' denotes the embedded vector component and the rest of the array, comprising of section '3' and the extra zero-padding to permit the use of fast DFT algorithms, is set to zero. Once embedded the DFT of the three components is performed in the $\hat{\mathbf{y}}$ direction. Then, each of the $[K \times P]$ $xz$-planes is, in turn, embedded into a larger $[K'' \times P'']$ external zero-padded scratch $xz$-plane, see Fig. 3.8d. Subsequently, the $\hat{\mathbf{x}}$ direction & $\hat{\mathbf{z}}$ direction DFTs, the element-wise FD multiplication of the tensor & vector components and the $\hat{\mathbf{z}}$ direction & $\hat{\mathbf{x}}$ direction iDFTs of the resultant are performed 'on-the-fly' for this $xz$-plane. The initial $[K \times P]$ block can now be copied back into the 3D vector component array overwriting the original data. Finally, once this has been done for each of the $J''$ $xz$-planes, the iDFT of the three components can be performed in the $\hat{\mathbf{y}}$ direction.

The reason that this is all possible is due to the fact that only non-zero DFTs and only iDFTs that will contribute to the final requisite answer need be performed. As far as the DFTs are concerned, in the $\hat{\mathbf{y}}$ direction, only the lines ${}^{[J']}\mathrm{x0\,3D}_{i,k}, i = 0, \ldots, K-1, k = 0, \ldots, P-1$ contain any non-zero values and need be considered. Subsequently, in the $\hat{\mathbf{x}}$ direction, only the lines ${}^{[K']}\mathrm{x0\,3D}_{j,k}, j = 0, \ldots, J'-1, k = 0, \ldots, P-1$ need be transformed, and in the $\hat{\mathbf{z}}$ direction, all the ${}^{[P']}\mathrm{x0\,3D}_{i,j}, i = 0, \ldots, K'-1, j = 0, \ldots, J'-1$ lines contain non-zero values and must be included. Taking into account the zero-padding to ensure $K'' = K' + kf = 2K - 1 + kf$, $J'' = J' + jf = 2J - 1 + jf$ and $P'' = P' + pf = 2P - 1 + pf$ to permit the use of fast DFT algorithms, in the $\hat{\mathbf{y}}$ **direction**, see Fig. 3.8a, $\hat{\mathbf{x}}$ **direction**, see Fig. 3.8b & $\hat{\mathbf{z}}$ **direction**, see Fig. 3.8c, the number of transforms is reduced from $3\,[K'' \times P''] \rightarrow 3\,[K \times P]$, $3\,[J'' \times P''] \rightarrow 3\,[J'' \times P]$ and '*no reduction possible*', respectively, which is a reduction of approximately 42%.

The FD element-wise multiplications can be carried out just as detailed in Eqs. 3.25 $\rightarrow$ 3.27, with one minor caveat. The six tensor component arrays never underwent the full T-to-c conversion, however, since the DFT preserves the mirror symmetries, when accessing Y0 $\rightarrow$ Y5, the pertinent data, $i = 0, \ldots, K'-1, j = 0, \ldots, J'-1, k = 0, \ldots, P'-1$, can be extracted from Y0 $\rightarrow$ Y5 using the criteria:

$$i = \begin{cases} i, & \text{if } i < K; \\ K' - i, & \text{if } i \geq K. \end{cases} \qquad i = 0, \ldots, K'-1 \tag{3.31}$$

$$j = \begin{cases} j, & \text{if } j < J; \\ J' - j, & \text{if } j \geq J. \end{cases} \qquad j = 0, \ldots, J'-1 \tag{3.32}$$

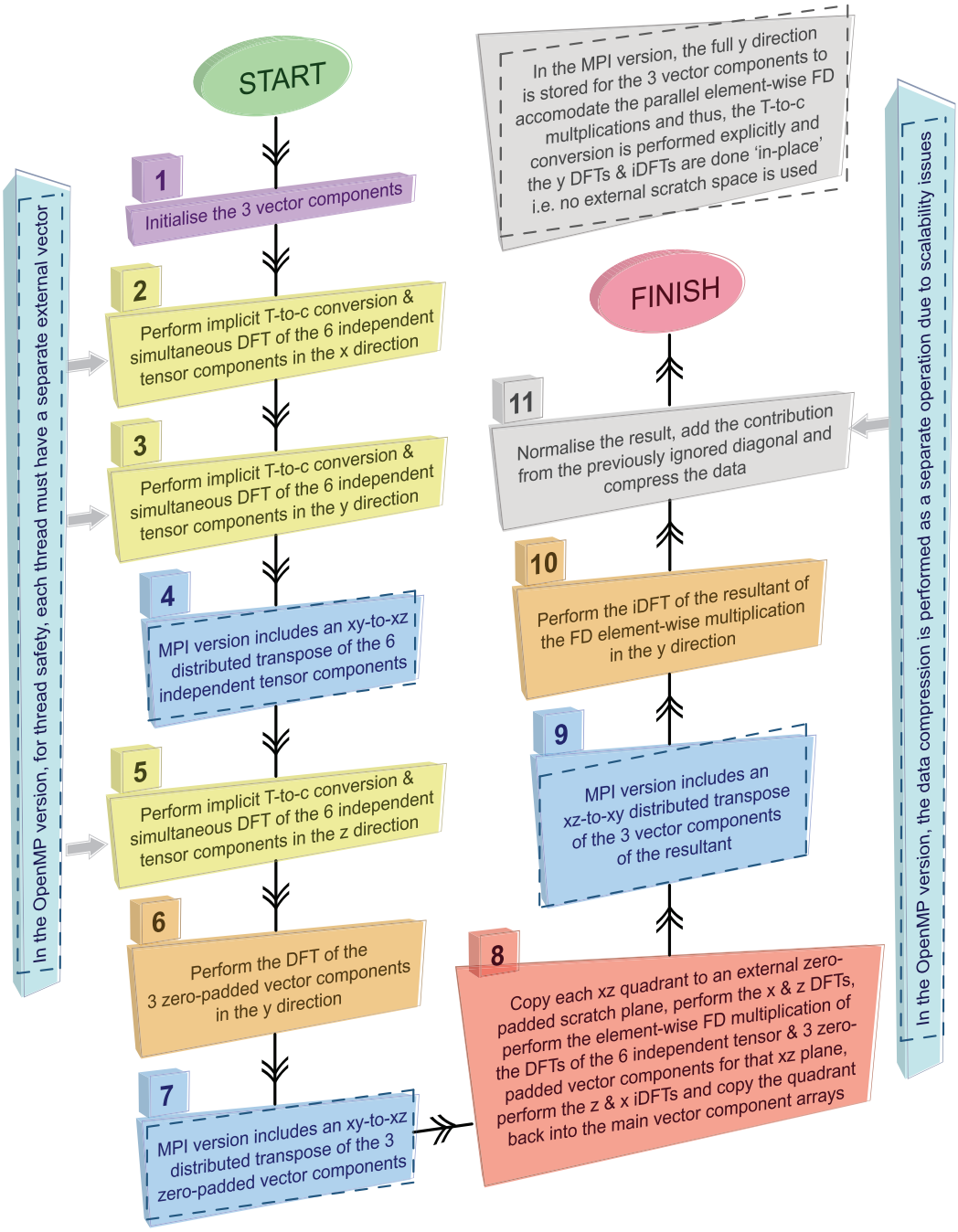



Figure 3.9: A flowchart for the DFT matrix-vector product as prescribed by the new algorithm. Includes the serial, OpenMP and MPI implementations.

$$k = \begin{cases} k, & \text{if } k < P; \\ P' - k, & \text{if } k \geq P. \end{cases} \qquad k = 0, \ldots, P' - 1 \tag{3.33}$$

For the iDFT of the resultant of the FD multiplication, i.e., of Q0, Q1 & Q2, only transforms that will contribute to the final requisite answer need be performed, see Fig. 3.8. For the Q0 case, the lines in the $\hat{\mathbf{x}}$, $\hat{\mathbf{y}}$ & $\hat{\mathbf{z}}$ directions can be defined as:

$$^{[P']}\mathrm{Q0}_{i,j} = \begin{pmatrix} \mathrm{Q0}_{i,j,0} \\ \mathrm{Q0}_{i,j,1} \\ \mathrm{Q0}_{i,j,2} \\ \vdots \\ \mathrm{Q0}_{i,j,P'-2} \\ \mathrm{Q0}_{i,j,P'-1} \end{pmatrix} \qquad i = 0, \ldots, K' - 1, j = 0, \ldots, J' - 1$$

$$^{[J']}\mathrm{Q0}_{i,k} = \begin{pmatrix} \mathrm{Q0}_{i,0,k} \\ \mathrm{Q0}_{i,1,k} \\ \mathrm{Q0}_{i,2,k} \\ \vdots \\ \mathrm{Q0}_{i,J'-2,k} \\ \mathrm{Q0}_{i,J'-1,k} \end{pmatrix} \qquad i = 0, \ldots, K' - 1, k = 0, \ldots, P' - 1$$

$$^{[K']}\mathrm{Q0}_{j,k} = \begin{pmatrix} \mathrm{Q0}_{0,j,k} \\ \mathrm{Q0}_{1,j,k} \\ \mathrm{Q0}_{2,j,k} \\ \vdots \\ \mathrm{Q0}_{K'-2,j,k} \\ \mathrm{Q0}_{K'-1,j,k} \end{pmatrix} \qquad j = 0, \ldots, J' - 1, k = 0, \ldots, P' - 1$$

In the $\hat{\mathbf{z}}$ direction, all the lines $^{[P']}\mathrm{Q0}_{i,j}, i = 0, \ldots, K' - 1, j = 0, \ldots, J' - 1$ contribute, through the subsequent $\hat{\mathbf{x}}$ and $\hat{\mathbf{y}}$ transforms, to the final answer. Subsequently, in the $\hat{\mathbf{x}}$ direction, only the lines $^{[K']}\mathrm{Q0}_{j,k}, j = 0, \ldots, J' - 1, k = 0, \ldots, P - 1$ contribute, through the subsequent $\hat{\mathbf{y}}$ transform, to the final answer. Finally in the $\hat{\mathbf{y}}$ direction, only the lines $^{[J']}\mathrm{Q0}_{i,k}, i = 0, \ldots, K - 1, k = 0, \ldots, P - 1$ contribute and need be considered. Taking into account the zero-padding to ensure $K'' = K' + kf = 2K - 1 + kf$, $J'' = J' + jf = 2J - 1 + jf$ and $P'' = P' + pf = 2P - 1 + pf$ to permit the use of fast iDFT algorithms, in the $\hat{\mathbf{z}}$ **direction**, see Fig. 3.8c, $\hat{\mathbf{x}}$ **direction**, see Fig. 3.8b & $\hat{\mathbf{y}}$ **direction**, see Fig. 3.8a, the number of transforms is reduced from '*no reduction possible*', $3\,[J'' \times P''] \rightarrow 3\,[J'' \times P]$ and $3\,[K'' \times P''] \rightarrow 3\,[K \times P]$,

| COMPUTATION | Original Algorithm | New Algorithm |
|---|---|---|
| Y0→Y5:$\hat{\mathbf{x}}$ | $O(6N'' \ln K'')$ | $O(6JPK'' \ln K'')$ |
| Y0→Y5:$\hat{\mathbf{y}}$ | $O(6N'' \ln J'')$ | $O(6[K+kf]PJ'' \ln J'')$ |
| Y0→Y5:$\hat{\mathbf{z}}$ | $O(6N'' \ln P'')$ | $O(6[K+kf][J+jf]P'' \ln P'')$ |
| X0→X2:$\hat{\mathbf{x}}$ | $O(3N'' \ln K'')$ | $O(3J''PK'' \ln K'')$ |
| X0→X2:$\hat{\mathbf{y}}$ | $O(3N'' \ln J'')$ | $O(3KPJ'' \ln J'')$ |
| X0→X2:$\hat{\mathbf{z}}$ | $O(3N'' \ln P'')$ | $O(3N'' \ln P'')$ |
| Q0→Q2:$\hat{\mathbf{x}}$ | $O(3N'' \ln K'')$ | $O(3J''PK'' \ln K'')$ |
| Q0→Q2:$\hat{\mathbf{y}}$ | $O(3N'' \ln J'')$ | $O(3KPJ'' \ln J'')$ |
| Q0→Q2:$\hat{\mathbf{z}}$ | $O(3N'' \ln P'')$ | $O(3N'' \ln P'')$ |
| **STORAGE** | **Original Algorithm** | **New Algorithm** |
| Y0→Y5 | $O(6N'')$ | $O(6[(K+kf)\times(J+jf)\times(P+pf)])$ |
| X0→X2 | $O(3N'')$ | $O(3[K\times J''\times P])$ |

Table 3.2: Complexity estimations for the original and new algorithms.

respectively, which is a reduction of almost 42%.

The approximate complexity improvements for both computation and storage in comparison to the original algorithm are given in Table 3.2. As a quantatative illustration of the incontrovertible benefits associated with the prescribed optimisations, the specific improvements for an arbitrary case where $K = J = P = 255$, and consequently, $K' = J' = P' = 509$ & $K'' = J'' = P'' = 512$, will be detailed in Tables 6.1 & 6.2 in §6.2.1. However, to succinctly summarise the findings, for this example, in comparison to the original algorithm, the new algorithm requires over 62%, 75% and 37% less transforms in the $\hat{\mathbf{x}}$, $\hat{\mathbf{y}}$ and $\hat{\mathbf{z}}$ directions, respectively, which equates to a reduction of almost 60% in the total number of transforms that need to be performed in order to carry out the matrix-vector product. Furthermore, the memory required to store both the six independent tensor components and three zero-padded vector components has been reduced significantly. The fact that the T-to-c conversion of the six tensor components was never explicitly performed means that the storage for the associated arrays has been reduced by over 87%. For the three vector components, the use of an external $xz$-scratch plane provides a reduction of over 75%. Together, these optimisations reduced the memory required to store the tensor and vector components by over 83%.

Finally, to aid visualisation and help elucidate the structure of the prescribed algorithm, Fig. 3.9 displays a flowchart for the complete DFT matrix-vector product illustrating the various components that make up the new algorithm.

*"Even he, to whom most things that most people would think were pretty smart were pretty dumb, thought it was pretty smart."*

**Douglas Adams**

# 4

# DFT kernel - Parallel Implementation

## 4.1 Introduction

Over and above the serial improvements, due to the fact that the scheme has been reduced from using 3D to 1D DFTs, parallel techniques can be employed to further enhance the performance. Parallel computing essentially involve using one of the many parallel programming APIs to enable multiple processors to collaborate on a single problem. The implementations can be divided into two architecture dependant categories, that of shared-memory and distributed-memory. For shared-memory systems, as the name suggests, there is a single address space to which all processors have access, whereas for distributed-memory systems, each node is autonomous, with its own local memory, and thus, message passing is required to exchange data between processors. The chapter begins will a discussion of the OpenMP-based shared-memory implementation. It includes a description of the various library routines & compiler directives required to create the parallel algorithm and introduces the notion of and importance of thread safety. The focus then turns to the much more complicated MPI-based distributed-memory implementation. The description of the distributed algorithm begins with the custom domain decomposition and parallel memory allocation methodologies which form the basis on which the rest of the distributed algorithm is built. Subsequently, with the entire 3D system plane-wise decomposed across the distributed environment, the design considerations

for the distributed transposes that are necessary to facilitate the requisite 3D DFTs are discussed. The subtleties of the internal message passing protocols are explained and the custom local transposes, which are used to streamline the host algorithm, are described. With the underlying infrastructure in hand, the final sections deal with the actual custom distributed transposes themselves, which can treat completely arbitrary systems decomposed across an arbitrary number of nodes. The first is based on an exploitation of the internal message passing protocols and, due to the fact that there are certain associated portability restrictions, a fully-portable algorithm, based on explicit buffering and the use of local transposes, is detailed.

## 4.2 Shared-memory implementation - OpenMP

**OpenMP** defines a collection of compiler directives, library routines and environment variables which allow users to create and manage parallel programs while permitting portability. The OpenMP API uses the fork-join model of parallel execution, i.e., OpenMP program begins as a single thread of execution, called the initial thread. When any thread encounters a parallel construct, the thread creates a team of itself and zero or more additional threads and becomes the master of the new team. All members of the new team execute the code inside the parallel construct. At the end of the parallel construct there is an implicit barrier, after which, only the master thread continues execution of the code. However, any number of parallel constructs can be specified in a single program [1]. In the C programming language, OpenMP directives are specified with the '**#pragma**' mechanism. This '**#pragma**' directive is a compiler specific directive which compiler vendors may use for their own purposes. The ANSI standard specifies that the '**#pragma**' directive has an arbitrary, implementation-defined effect. It is the method specified by the C standard for providing additional information to the compiler, beyond what is conveyed in the language itself.

### 4.2.1 Library routines and compiler directives

**For** shared-memory architectures with an available OpenMP compliant compiler, OpenMP [1] provides a simple and flexible interface for developing parallel applications. It allows for gradual parallelisation through the incremental addition of directives which makes code development and debugging much

easier than for its distributed counterparts. OpenMP enabled code is more readily understood and maintained than distributed implementations and can even be run as a serial code if required. Its ease of use stems from the fact that apart from the OpenMP directives, the serial code itself usually does not require any modification. This was self-evident during the development of the OpenMP version of the new DFT algorithm discussed in §3.6, where apart from the addition of the OpenMP directives and the fact that the FFTW plan creation routines are not thread-safe, the serial code required no modifications in order to implement the parallelisation. OpenMP's mainstay is essentially '**for**' loop parallelisation and although the OpenMP API incorporates a vast number of options and features, as with most parallel implementations, very few are required to be able to construct fully functional and efficient parallel code. To see this explicitly, two extremely basic and concise code examples have been added, see Excerpts 4.1 & 4.2, which between them, implement *every* library routine and compiler directive that was needed to create the OpenMP version of the new DFT algorithm. Excerpt 4.1 focuses on applying the various OpenMP library routines that were used and Excerpt 4.2 concentrates on the use of the OpenMP compiler directives. The function of the two examples is quite inconsequential with the aim being to illustrate the context and implementation of the OpenMP routines and directives.

Excerpt 4.1 uses '`omp_set_num_threads`()' to explicitly set the number of threads to be used for subsequent parallel regions to 4. At this point only a single thread is active and '**#pragma** omp parallel' is used to create a parallel construct. When the single thread encounters this parallel construct, a team of threads is created to execute the parallel region, in this case 4. The thread which encountered the parallel construct becomes the master thread of the new team, with a thread number of '0' for the duration of the parallel region. All threads in the new team, including the master thread, execute the region. Within this parallel region, '`omp_get_num_threads`()' is used to get the number of active threads. Of course this is purely an illustrative gesture as the number of threads to be used in the execution of parallel regions was already explicitly set to 4. Furthermore, it would be pointless to get every thread in the same team to query the number of threads in that team so execution is restricted to just the master thread using '**#pragma** omp master'. Finally, each thread uses the '`omp_get_thread_num`()' library routine to obtain its unique thread number.

Code Excerpt 4.1: Example demonstrating the relevant OpenMP library routines for setting the number of parallel threads for subsequent parallel regions, querying the

number of active threads and acquiring a unique thread identifier.

```
#include <stdio.h>
#include <omp.h>

int main(void){

   int threads,thread;

   omp_set_num_threads(4); /* Set the no. of available threads to 4 */

#pragma omp parallel /* omp_get_num_threads() must be executed in a parallel region */
   {
#pragma omp master /* Only executed by the master thread */
      {
         threads=omp_get_num_threads(); /* This gets the number of active threads */
      }
      thread=omp_get_thread_num(); /* Get thread number of the current thread */
      printf("Hello! I am thread %d of %d\n",thread,threads);
   }

   return 0;
}

/* Output:
Hello! I am thread 0 of 4
Hello! I am thread 3 of 4
Hello! I am thread 1 of 4
Hello! I am thread 2 of 4 */
```

Excerpt 4.2 covers the OpenMP compiler directives and starts just as Excerpt 4.1 with a call to '`omp_set_num_threads`()' to explicitly set the number of threads to be used for subsequent parallel regions to 4. '**#pragma** omp parallel **for**' is used to set all 10 rows in '`matrix`[`j`][`i`]', `j`= $0 \ldots 9$, `i`= $0 \ldots 4$ to `i`, i.e., $\{0, 1, 2, 3, 4\}$. Although this system is trivially small and would never be parallelised in reality, for the purpose of illustration, it is ideal. '**#pragma** omp parallel **for**' is an amalgamation of '**#pragma** omp parallel' and '**#pragma** omp **for**'. It essentially instructs the compiler to distribute loop iterations within the team of threads created by the parallel directive. Note that the main loop iteration variable, '`j`', is implicitly '`private`' by default, and thus, only the variable '`i`' need be declared as '`private`'. To control the distribution of loop iterations '`schedule`(`dynamic`)' is used which dynamically assigns iterations as the threads request them.

For the main '**#pragma** omp parallel **for**' construct, '`j`', is once again implicitly '`private`' by default and '`i`' is declared as '`private`'. The construct performs the simple task of adding all the entries of '`matrix`[`j`][`i`]' and for each iteration, '`j`'=0...9, adds a constant '`k`'. The reason for '`k`' is just to show that '`k`' is essentially a global variable, visible to all active threads, and thus, does not need to be made '`private`'. Since each of the 4 active threads executes a only a portion of the iterations, no thread has the complete '`sum`', and thus, the '`reduction`' clause must be used to gather the result. Here, for '`sum`', a private copy is created on each thread, and is initialised to 0. At the end of the region, the original '`sum`' is updated with the values of the private copies using the '+' operator. The construct also has a '`flag`=0' passed to it to illustrate the use of '`firstprivate`', '`lastprivate`' and the '**#pragma** omp critical' directive. The '`firstprivate`' directive declares '`flag`' to be '`private`' and initialises

it with the value that the corresponding original item had when the construct is encountered, i.e., '0'. The '#pragma omp critical' directive, which due to 'flag', is only executed once by each thread, simply restricts execution of the associated structured block to a single thread at a time. Finally, 'lastprivate' causes the original 'flag' to be updated after the end of the region. For each of the aforementioned OpenMP library routines and compiler directives, a more formal description of the subtleties is presented in Appendices B.1, B.2 & B.2.1.

Code Excerpt 4.2: Example showing the use of some of the OpenMP compiler directives.

```
#include <stdio.h>
#include <omp.h>

int main(void){

    int i,j,matrix[10][5],flag,k=4,sum;

    omp_set_num_threads(4); /* Set the no. of available threads to 4 */

#pragma omp parallel for private(i) schedule(dynamic)
    for(j=0;j<10;j++){ /* Set matrix[j][]={0,1,2,3,4} */
        for(i=0;i<5;i++){
            matrix[j][i]=i;
        }
    }

    flag=0;
#pragma omp parallel for private(i) firstprivate(flag) lastprivate(flag) reduction(+:sum)
    for(j=0;j<10;j++){ /* Sum the values of matrix[][] */
        for(i=0;i<5;i++){
            sum+=matrix[j][i]; /* sum=0+1+2+3+4=10 */
        }
        sum+=k; /* add constant sum=sum+4 */
        if(!flag){
#pragma omp critical /* Only 1 thread at a time can execute */
                printf("I am thread %d and I am the only one\
 currently executing this section!\n",omp_get_thread_num());
                flag=1;
        }
    }
    printf("flag=%d\n",flag);
    printf("sum=%d\n",sum);

    return 0;
}

/* Output:
I am thread 0 and I am the only one currently executing this section!
I am thread 3 and I am the only one currently executing this section!
I am thread 1 and I am the only one currently executing this section!
I am thread 2 and I am the only one currently executing this section!
flag=1
sum=140 */
```

### 4.2.2 Thread safety

Thread safety is of paramount importance in the development of software for multi-threaded environments. With several threads operating within a single address space it is crucial that no two threads step on each others' toes. This can be achieved either through appropriate flow control or by creating 'private' copies of the pertinent memory for each thread.

For the OpenMP implementation of the DFT algorithm, the fact that the FFTW plan creation routines are not thread-safe means that the '#pragma omp critical'

compiler directive must be used to restrict the plan creation to a single thread at a time. Furthermore, within the DFT algorithm, for the implicit T-to-c conversion and simultaneous DFT of the six independent tensor components discussed in §3.6, the pertinent data was copied to an external 1D scratch array, treated as the first column of a level-one Toeplitz matrix and underwent a T-to-c conversion by mirroring the elements, excluding the initial entry, across the last entry and multiplying the mirrored elements by the appropriate entry from Table3.1. The DFTs of these external vectors were computed and subsequently, the relevant elements were copied from the resultants into the 3D tensor component array overwriting the original data. For the DFTs of the three zero-padded vector component arrays, the subsequent element-wise FD multiplication of the tensor and vector components and the iDFTs of the resultant, each of the vector component $[P \times K]$ $xz$-planes was, in turn, embedded into a larger $[P'' \times K'']$ external zero-padded scratch $xz$-plane. Subsequently, the $\hat{\mathbf{x}}$ direction & $\hat{\mathbf{z}}$ direction DFTs, the element-wise FD multiplication of the tensor and vector components and the $\hat{\mathbf{z}}$ direction & $\hat{\mathbf{x}}$ direction iDFTs of the resultant were performed 'on-the-fly' for this $xz$-plane. The initial $[P \times K]$ block was then copied back into the 3D vector component array overwriting the original data. For the threaded version, this requires that each thread has its own '`private`' external scratch arrays for the $\hat{\mathbf{x}}$, $\hat{\mathbf{y}}$ & $\hat{\mathbf{z}}$ directions. This is accomplished by using the '`omp_get_num_threads`()' library routine to get the number of available threads and then allocating an array of 'thread' external vectors for each of the three directions. During execution, each thread accesses its own external vector using its unique thread number via the '`omp_get_thread_num`()' library routine.

## 4.3 Distributed-memory implementation - MPI

The Message Passing Interface (MPI) is a portable standard for message passing and is designed to support parallel computing through point-to-point & collective communication and process synchronisation in a distributed memory environment. Distributed computing is the process of administrating the activities of autonomous machines with separate and distinct address spaces to work cooperatively to improve the efficiency of a specific computational task. Although MPI-based applications are, in general, far more complex than their shared-memory analogue, the essentially 'plug-and-play' nature of the shared-memory OpenMP environment is offset by the price and limited availability of large scale shared-memory

machines. Distributed systems based on commodity components are less expensive, easier to construct and offer higher scalability (as long as the communication requirements are reasonable) than the shared-memory systems which can be prone to memory contention issues. Furthermore, code built for distributed systems using MPI can just as easily be run on either shared-memory or distributed-memory architectures and can be developed using any of a number of freely available & portable implementations of MPI, including MPICH1 [208, 2], MPICH2 [3] or Open MPI [4].

The extra complexity of MPI-based codes is due to the fact that with distributed systems, each processor has its own local memory, and thus, the developer must be concerned with data distribution and the intricacies of interprocessor communication, i.e., every communication must be dealt with explicitly and communication deadlock must be avoided. In contrast to OpenMP, MPI requires significantly more programming and structural change to the serial code to implement the parallelisation and can be much harder to debug as a result. Its performance depends heavily on the amount of communication required, the performance of the communication network between the nodes and on the load balance. Load balancing, which tries to ensure that work is distributed evenly to the available processors so that no one node is overwhelmed, is part and parcel of the adopted domain decomposition scheme. Like a chain whose strength is determined by its weakest link, if the load balance is not optimal, then the efficiency of the MPI implementation is governed by the most encumbered node.

Whereas the OpenMP version of new DFT matrix-vector product algorithm required only minor alteration to the serial code, the MPI version is a completely different entity whose creation required the resolution of several key issues. The blueprint had to include mechanisms for data distribution & memory allocation and since the algorithm includes 3D DFTs, custom-built distributed transposes had to be designed that could transpose the six tensor and three vector components arrays from $xy$-to-$xz$ plane configuration and back again.

### 4.3.1 Domain decomposition

**Domain** decomposition and memory allocation constitute the fundamental chassis on which the rest of the MPI algorithm operates and are paramount to the overall efficiency of the end product. The infrastructure has to be simple enough so as not to impede or over complicate the development of the core algorithm yet still maintain optimality. To achieve this, custom data distri-

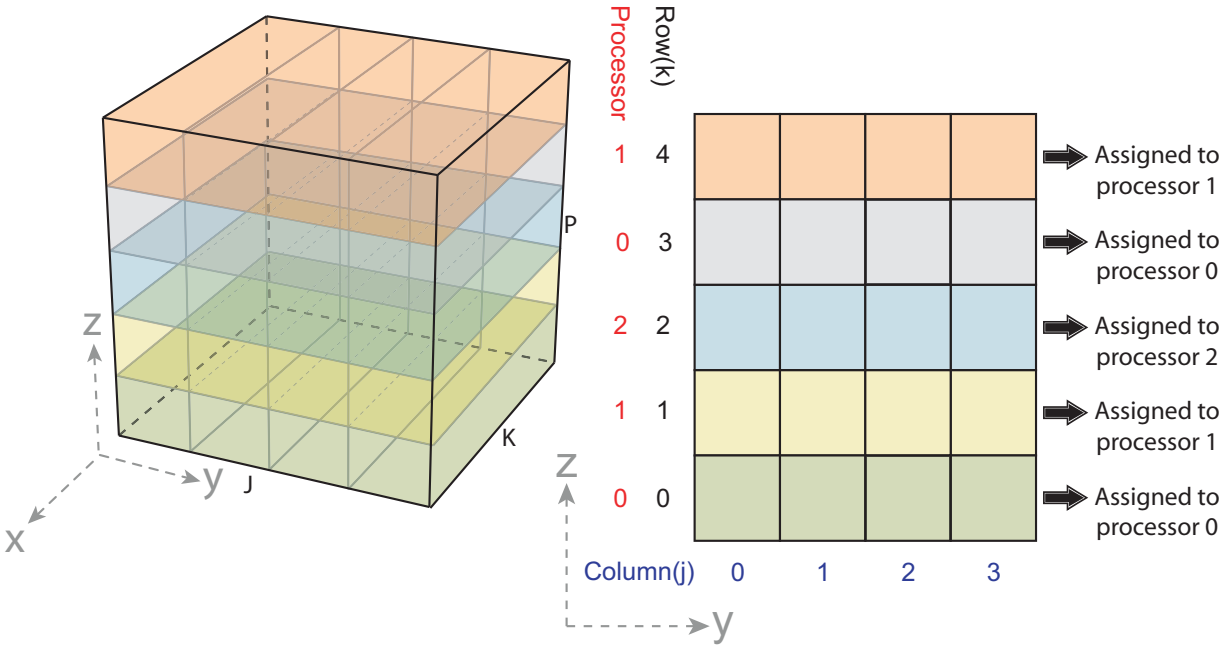



Figure 4.1: Domain decomposition implemented using 'cyclic-plane' distribution whereby planes are cyclically assigned to the 3 available processors, *rank=0,1,2*, using the *modulo* function.

bution mechanisms were designed based on simple but effective concepts to permit the efficient location of and access to pertinent data. The domain decomposition strategy, which will be discussed here, is of course, the pivotal consideration and the memory allocation aspect, which is ultimately dependent on the data distribution, will be discussed in the §4.3.2.

The basic domain decomposition employs a 'cyclic-plane' distribution whereby planes are cyclically assigned to the available processors using the *modulo* function, see Fig. 4.1. Apart from the inherent 'to-the-nearest-plane' load balancing which this affords, the distribution scheme also allows for trivial '**for**' loop parallelisation, see Excerpt 4.3. This is an ideal attribute for the DFT matrix-vector algorithm as it is predominantly composed of '**for**' loop based operations. Furthermore, it is worth a brief interjection here to mention that, due to the fact that targets predominantly do not have a constant number of occupied sites per plane, this domain decomposition scheme has load balancing ramifications far beyond that of the DFT matrix-vector product kernel. As was discussed in §2.6, the custom-built iterative solution schemes that are driven by this DFT engine only store information corresponding to occupied lattice sites, which more often than not, provides a significant reduction in the mem-

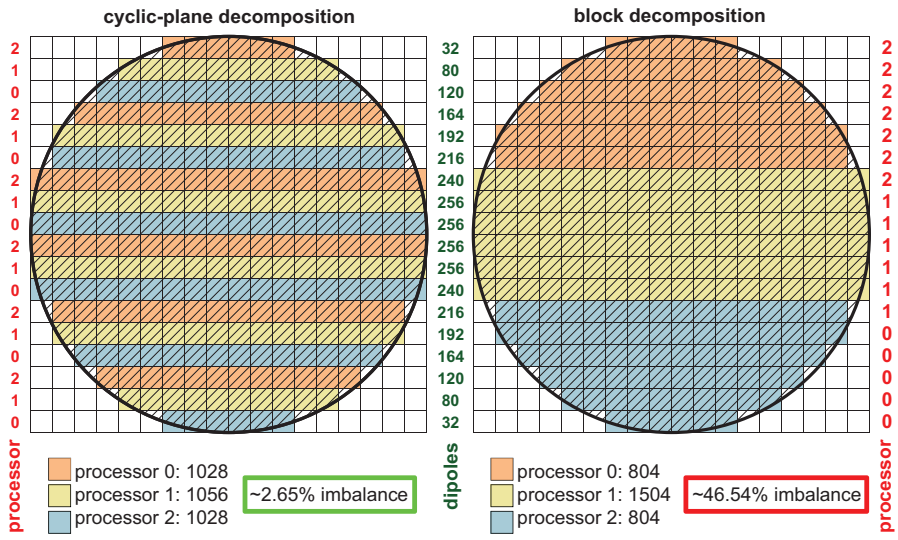



Figure 4.2: Load balance comparison for block and 'cyclic-plane' decomposition for a simplistic spherical target based on an $18\times18\times18$ computational grid.

ory required to store the iterative solution vectors. The example that was used was that of a spherical target based on an $18\times18\times18$ computational grid and since only 3112 of the 5832 lattice sites corresponded to occupied lattice sites, a decrease of approximately 47% in the memory required to store these vector was obtained. In terms of the physical distribution of occupied lattice sites, for the spherical target, there is quite an imbalance in the number of occupied sites per plane and as a consequence, if this computational domain is 'block' decomposed, as is shown on the right of Fig. 4.2, then, even for this simplistic case, the imbalance of 46.54% in the data distribution dwarfs the negligible 2.65% imbalance for the 'cyclic-plane' decomposition shown on the left of the same figure.

Code Excerpt 4.3: Example of '**for**' loop parallelisation using 'cyclic-plane' data distribution.

```
/* np:       Number of processors
   myrank:   Processor rank
   k:        Loop variable running over the P planes, k=0,...,P-1 */

for(k=0;k<P;k++){
    if(k%np==myrank){ /* Plane k is assigned to proc k%np */

        ...

    }
}
```

### 4.3.2 Parallel memory allocation

For memory allocation, it was decided to custom-design a mechanism that would integrate flawlessly with the data distribution scheme and allow for the implementation of complete parallel memory allocation. It would have been extremely easy to simply allocate the full memory serially, so that each node possessed an entire copy and just worked on its assigned segment. However, in terms of resources, this would have been extremely wasteful and would more importantly have sabotaged the applicability and scalability of the system. By allocating the memory in parallel, a decrease in memory requirements per node that is directly proportional to the number of nodes available can be achieved, allowing hitherto impracticable systems to be treated.

To facilitate parallel memory allocation, algorithms that can determine the memory requirements for each node are required and furthermore, since the planes are to be physically distributed among the available nodes, a way of determining on which node a specified plane is located and indeed, which plane it is on that node, are crucial components. The computation of the memory requirements for a specific node is trickier than it sounds. The possibility of non-cubic systems and/or system sizes that are not divisible by the number of processors must be catered for. The situation is further complicated by the fact that each of the six independent tensor components & three zero-padded vector components must undergo a complete physical 3D transpose across the entire distributed cluster, yielding memory requirements per node that may differ *before* & *after* these transposes.

The basic parallel setup and transpose strategy is shown in Fig. 4.3. It displays a schematic of the two possible configurations for a $[P \times J = 5 \times 4]$ system distributed across $np = 3$ nodes, where $K$, the length in the $\hat{\mathbf{x}}$-direction, can be arbitrary. Fig. 4.3(a) shows the layout for $xy$-plane configuration, where the $P = 5$ $xy$-planes of width $J = 4$ are decomposed across the 3 available nodes, and Fig. 4.3(b) shows $xz$-plane configuration, where the $J = 4$ $xz$-planes of width $P = 5$ are decomposed across the same nodes. The complication lies in the fact that, not only are the constituent planes to be assigned to different nodes but they are to physically exist on separate nodes, i.e., in $xy$-plane configuration, processor 0 stores $xy$-planes {0 & 3}=(8 $\hat{\mathbf{x}}$-direction lines), processor 1 stores $xy$-planes {1 & 4}=(8 $\hat{\mathbf{x}}$-direction lines) and processor 2 stores $xy$-plane {2}=(4 $\hat{\mathbf{x}}$-direction lines) as shown in Fig. 4.3(a). To exacerbate the complexity, in $xz$-configuration, processor 0 should store $xz$-planes {0 & 3}=(10 $\hat{\mathbf{x}}$-direction lines), processor 1 should store $xz$-plane {1}=(5 $\hat{\mathbf{x}}$-direction

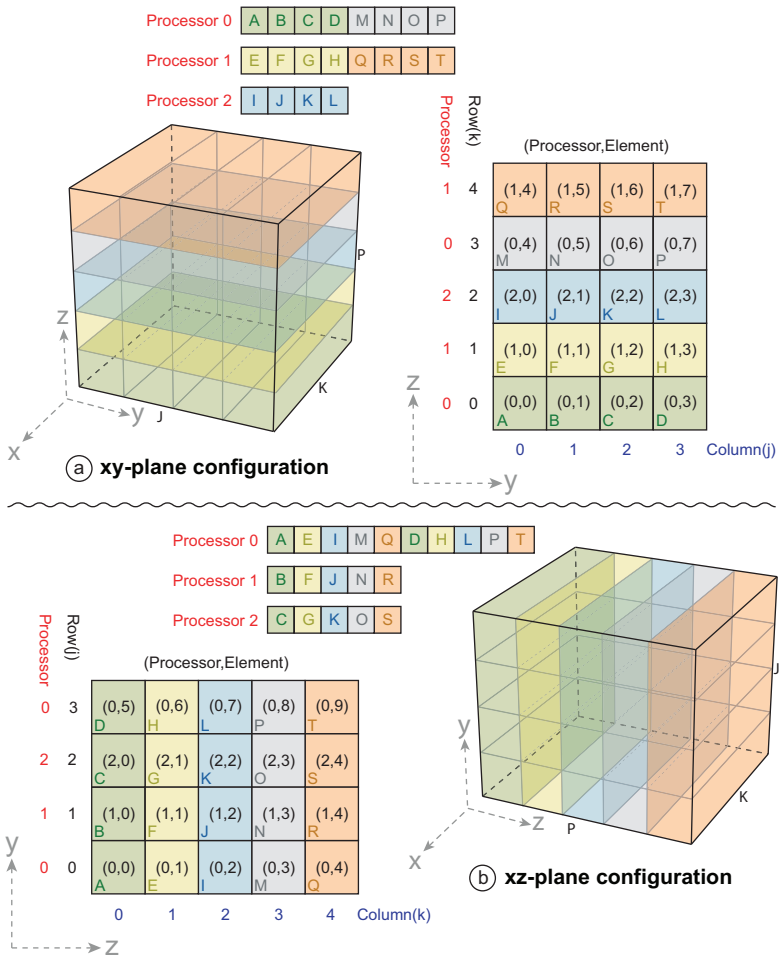



Figure 4.3: Schematic for parallel memory allocation across 3 nodes showing the allocation in $xy$-plane (a) and in $xz$-plane (b) configurations.

lines) and processor 2 should store $xz$-plane $\{2\}$=(5 $\hat{\mathbf{x}}$-direction lines) as shown in Fig. 4.3(b). The key to the success of the parallel memory allocation is to calculate the maximum memory requirement of each node in the $xy$-plane & $xz$-plane configurations and to allocate the larger of the two. In order to accomplish this a custom algorithm was constructed that is performed identically by each processor and, depending on the MPI rank of the processor, determines the storage requirements for both the $xy$-plane & $xz$-plane configurations for that node, setting a local variable *alloc* to the larger. The algorithm is based on the rules:

- **xy-plane configuration:**

$$\begin{aligned} planesY &= \begin{cases} Znp + 1 = (int)(P/np) + 1, & \text{if } myrank < P\%np; \\ Znp = (int)(P/np), & \text{if } myrank \geq P\%np. \end{cases} \\ limitY &= planesY \times K \end{aligned} \tag{4.1}$$

- **xz-plane configuration:**

$$\begin{aligned} planesZ &= \begin{cases} Ynp + 1 = (int)(J/np) + 1, & \text{if } myrank < J\%np; \\ Ynp = (int)(J/np), & \text{if } myrank \geq J\%np. \end{cases} \\ limitZ &= planesZ \times K \end{aligned} \tag{4.2}$$

- **Memory allocation**

$$alloc = \begin{cases} limitZ, & \text{if } limitZ \geq limitY; \\ limitY, & \text{if } limitY > limitZ. \end{cases} \tag{4.3}$$

Here, the symbol "%" indicates the use of the *modulo* function and the various variables have the following interpretation:

- $np$ is the number of available processors.
- $myrank$ is the MPI rank which uniquely identifies each processor.
- $alloc$ the maximum memory allocation required locally.
- **xy-plane configuration:**
  - $P$ is the number of $xy$-planes.
  - $J$ is the width of an $xy$-plane.
  - $Znp$ is the base number of $xy$-planes for any node.
  - $planesY$ is the number of $xy$-planes locally on the processor.
  - $limitY$ is the number of lines in the $\hat{\mathbf{x}}$-direction locally.
- **xz-plane configuration:**
  - $J$ is the number of $xz$-planes.
  - $P$ is the width of an $xz$-plane.

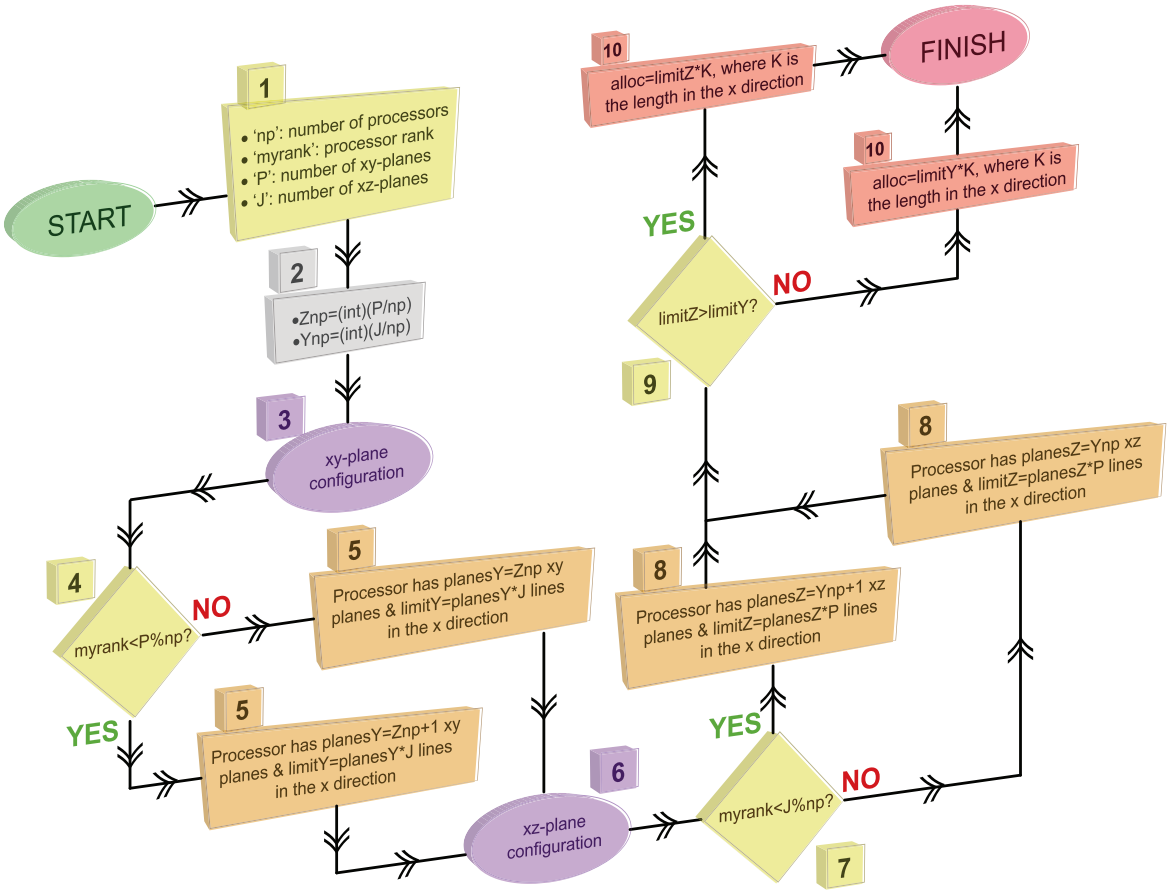



Figure 4.4: Flowchart for the algorithm which determines the maximum storage required by a node to implement both $xy$-plane & $xz$-plane configurations.

- $Ynp$ is the base number of $xz$-planes for any node.
- $planesZ$ is the number of $xz$-planes locally on the processor.
- $limitZ$ is the number of lines in the $\hat{\mathbf{x}}$-direction locally.

A basic flowchart for the algorithm is shown in Fig. 4.4. This method is extremely efficient as there is no catchall scenario where some all-encompassing safe value is arrived at and allocated on all processors. Each node determines its own memory requirements exactly and allocates itself precisely the right amount of memory.

The final data distribution and memory allocation hurdle is that, although the system is implemented using 'cyclic-plane' distribution, so that in $xy$-plane configuration, plane $k$, $k = 0 \ldots P-1$, is assigned to processor $k\%np$, and in $xz$-plane configuration, plane $j$, $j = 0 \ldots J-1$, is assigned to processor $j\%np$, with the memory

allocated in parallel, so that each processor only has a fraction of the entire system stored, in $xy$-plane configuration, plane $k$, $k=0\ldots P-1$, is no longer plane $k$ on processor $k\%np$, and in $xz$-plane configuration, plane $j$, $j=0\ldots J-1$, is no longer plane $j$ on processor $j\%np$. They would only have been planes $k$ & $j$ on a serially allocated system. For the parallel case, since the planes are allocated cyclically across all $np$ processors, in $xy$-plane configuration, plane $k$, $k=0\ldots P-1$, is on processor $k\%np$ and is plane $(int)(k/np)$ on that processor, and in $xz$-plane configuration, plane $j$, $j=0\ldots J-1$, is on processor $j\%np$ and is plane $(int)(j/np)$ on that processor. To show the practical application of this theory, Excerpt 4.4 has been included which demonstrates '**for**' loop parallelisation and plane localisation for a $P\times J\times K$ 3D system allocated as a 1D array for computational efficiency.

Code Excerpt 4.4: Example of parallel data access using 'cyclic-plane' distribution.

```
#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>

int np,myrank;

int main(int argc,char **argv){

    MPI_Init(&argc,&argv); /* Initialise MPI */
    MPI_Comm_rank(MPI_COMM_WORLD,&myrank); /* Get processor rank */
    MPI_Comm_size(MPI_COMM_WORLD,&np); /* Get number of processors */

    int i,j,k,K=3,J=4,P=5,M=K*J,Q=K*P,index0,index1,*matrix;

    matrix=(int*)malloc(sizeof(int)*K*J*P);

    ...

    for(k=0;k<P;k++){ /* xy-plane configuration */
        if(k%np==myrank){ /* Plane k is on proc k%np and is plane (int)(k/np) on that proc */
            plane=((int)((double)k/(double)np));
            for(j=0;j<J;j++){
                for(i=0;i<K;i++){
                    /* Would have been 'index1=k*M+j*K+i' for a serially allocated system */
                    index1=plane*M+j*K+i;
                    matrix[index1]=...;
                }
            }
        }
    }

    ...(Perform an xy-to-xz transpose)...
    /* Details of the transpose mechanism are irrelevant here and will be dealt with in a
       separate dedicated section, see §4.3.3 */


    for(j=0;j<J;j++){ /* xz-plane configuration */
        if(j%np==myrank){ /* Plane j is on proc j%np and is plane (int)(j/np) on that proc */
            plane=((int)((double)j/dnp));
            for(k=0;k<P;k++){
                for(i=0;i<K;i++){
                    /* Would have been 'j*Q+k*K+i' for a serially allocated system */
                    index1=plane*Q+k*K+i;
                    matrix[index1]=...;
                }
            }
        }
    }

    ...

    free(matrix);

    MPI_Finalize(); /* Terminate MPI Environment */

    return 0;
}
```

For the MPI version of the DFT matrix-vector product, the memory allocation for the six independent tensor components requires slight modification. Unlike the serial

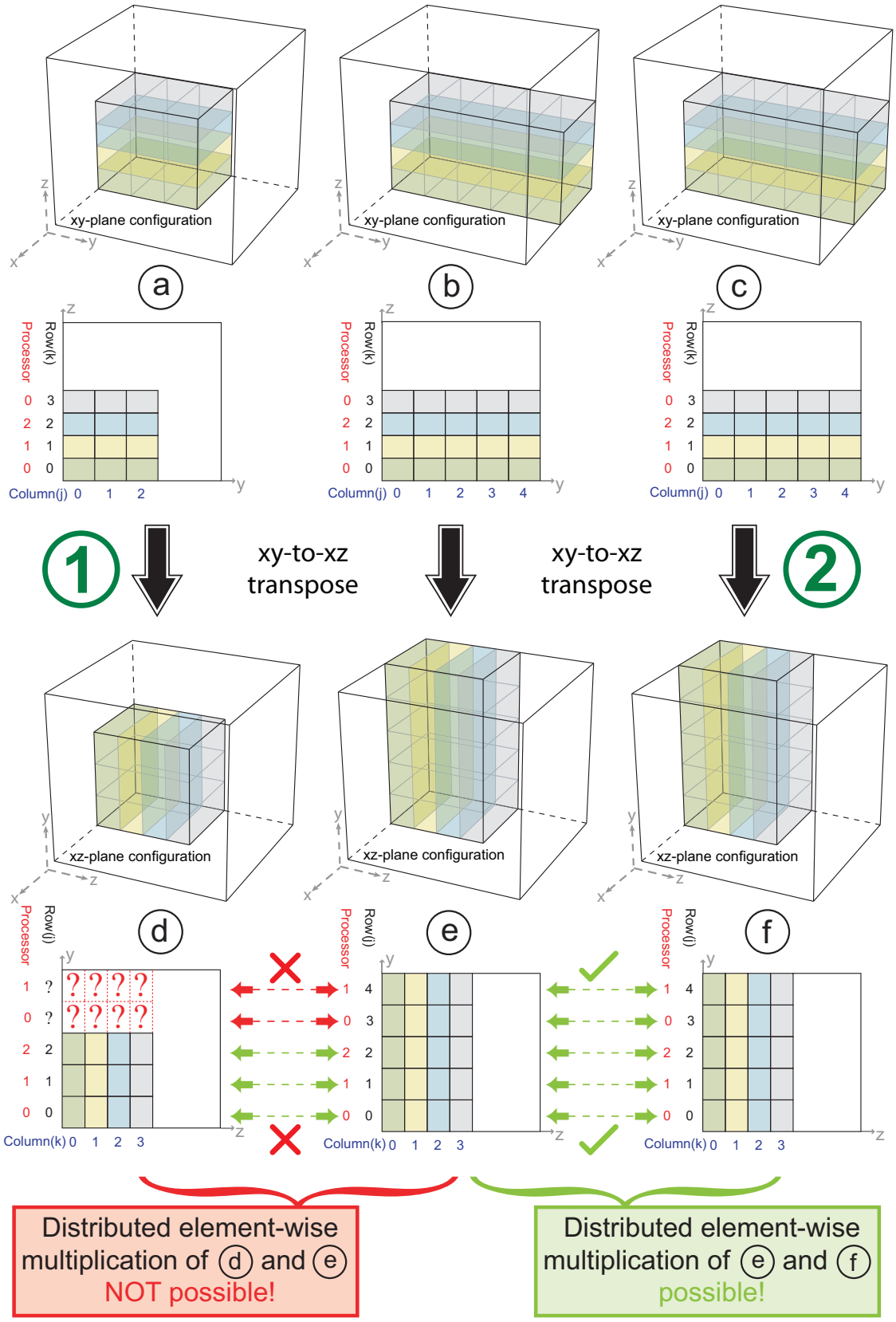



Figure 4.5: Schematic showing the distributed element-wise multiplication and the extra allocation required for each of the six independent tensor components for the MPI version of the DFT matrix-vector kernel.

and OpenMP versions, which for computational efficiency, only store section '1' in Fig. 3.7 for each component, the MPI version must also store section '3'. The reason for this is that for the serial cases, all the relevant data for the element-wise FD multiplication of the DFT of the six tensor components and the three vector components, see Eqs. 3.25→3.27, was local by definition, and thus, only section '1' was required as the algorithm exploited the symmetries within the T-to-c conversion and all pertinent data could be accessed using the criteria in Eqs. 3.31→3.33. For the parallel case, due to the 'cyclic-plane' distribution and parallel memory allocation, the planes are distributed across the available nodes, and thus, the relevant planes' symmetric counterpart, that is required to implement the aforementioned criteria, may not be known to the local processor. This is demonstrated in Fig. 4.5 where, for a system comprised of $np=3$ nodes, **(b)** & **(e)** show a schematic for the storage of the vector component arrays in $xy$-plane & $xz$-plane configurations, **(a)** & **(d)**, a schematic for the storage of the tensor component arrays in $xy$-plane & $xz$-plane configurations for scenario **(1)**, where the distributed implementation stores only section '1' from Fig. 3.7, and **(c)** & **(f)**, show a schematic for the storage of the tensor component arrays in $xy$-plane & $xz$-plane configurations for scenario **(2)**, where both section '1' and '3' from Fig. 3.7 are stored explicitly. The element-wise FD multiplication of the six tenor components & three vector component arrays takes place in $xz$-plane configuration. For scenario **(1)**, it is not possible to perform this multiplication as the system is decomposed across the available nodes in the $\hat{\mathbf{y}}$ direction and does not conform to the requirements of Eq. 3.32. For plane $j$, $j = J \ldots J'-1$, plane $J'-j$ may be located on a different node, and thus, precludes the calculation. However, for scenario **(2)**, by storing section '3' from Fig. 3.7, there is a perfect one-to-one correlation between the $xz$-planes of the tensor **(f)** and vector **(e)** component planes allowing for the calculation of the element-wise multiplication in parallel.

One might say that, theoretically, it would be possible to simply leave the original allocation as is, i.e., allocation scenario **(1)**, and to overcome the problem using communication, where the host for the relevant planes' symmetric counterpart could be located and the appropriate data could be sent to the required processor enabling the use of the criterion in Eq. 3.32. However, scenario **(2)** is by far the preferred option as communication is extremely inefficient in comparison to local calculation and should always be the parallel developer's last resort. The fact that the use of scenario **(2)** requires more memory than the serial algorithm when run on a single node is neither here nor there, as firstly, the MPI version is *not* intended to be run

in serial, and more importantly, for two or more nodes, the linear decrease in the memory required per node afforded by the parallel memory allocation makes the extra storage required by section '3' inconsequential.

From a practical standpoint, the modifications required in order to implement scenario **(2)** are quite straightforward. At the memory allocation stage, the addition of section '3' requires only the adjustment of $J$ by increasing it in Eq. 4.2 to the full width of the tensor array, i.e., **(a)**→**(c)** in Fig. 4.5. This increase in size produces a mild domino effect throughout the rest of the algorithm, which apart from having to alter various loop control limits to allow for the extended size of the tensor arrays, includes slightly modifying the implicit T-to-c conversion and simultaneous DFT of the six tensor components. The $\hat{\mathbf{x}}$ direction can be left as is but for the $\hat{\mathbf{y}}$ direction, since the full width is now allocated, the T-to-c conversion and DFT no longer need to be implicit. With section '3' allocated, the external vector that was used to perform the $\hat{\mathbf{y}}$ direction operations can be removed and they can be performed in-place using the tensor array itself. For the $\hat{\mathbf{z}}$ direction, the only alteration is to extend the treatment of the tensor arrays to the full width in the $\hat{\mathbf{y}}$ direction. With scenario **(1)** replaced, for the element-wise multiplication, the criteria for accessing pertinent data from the six tensor components, Y0 → Y5, can be rewritten as Eqs. 4.4→4.6, i.e., each processor now performs the element-wise multiplication of the tensor and vector components for only the $xz$-planes assigned to it.

$$i = \begin{cases} i, & \text{if } i < K; \\ K' - i, & \text{if } i \geq K. \end{cases} \qquad i = 0 \ldots K' - 1 \tag{4.4}$$

$$j = (int)(j/\text{np}), \text{myrank} = j\%\text{np} \qquad j = 0 \ldots J' - 1 \tag{4.5}$$

$$k = \begin{cases} k, & \text{if } k < P; \\ P' - k, & \text{if } k \geq P. \end{cases} \qquad k = 0 \ldots P' - 1 \tag{4.6}$$

### 4.3.3 Distributed transpose design considerations

From an MPI developer's point of view, the implementation of the DFT matrix-vector product algorithm poses quite a daunting task. The creation of the data distribution and the parallel allocation schemes constitute only the tip of the 'parallel iceberg'. The difficulty lies in the fact that the algorithm contains not one, but two 3D DFTs and a 3D iDFT. The fundamental precept for the creation of efficient distributed code is the minimisation of both the frequency and

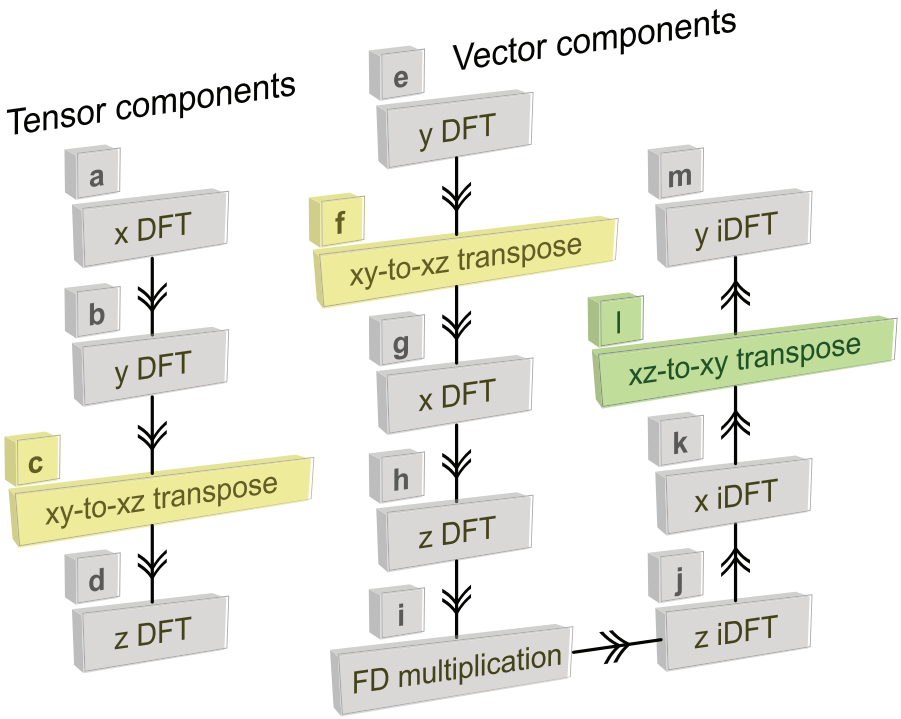



Figure 4.6: Distributed transpose requirements for the DFT kernel.

the amount of inter-processor communication by practically any means necessary. For the parallel environment, these DFTs constituent a worst case scenario where, due to the plane-wise decomposition of the data, individual processors have local knowledge of only two of the three dimensions, and thus, in order to facilitate parallel 3D DFTs, *full*-system 3D distributed transposes must be designed.

The distributed transpose requirements for the DFT kernel are shown in Fig. 4.6. Initially, there is not a problem as, with the system in $xy$-plane configuration, for each of the six independent tensor components, each node can perform the $\hat{\mathbf{x}}$ & $\hat{\mathbf{y}}$ stages of the DFTs on the local data, see Figs. 4.6a & 4.6b, respectively. However, in this $xy$-plane configuration, the system is decomposed across the available nodes in the $\hat{\mathbf{z}}$ direction and hence, it is impossible to compute the $\hat{\mathbf{z}}$ DFTs for these components. To compute the $\hat{\mathbf{z}}$ DFTs, see Fig. 4.6d, the *whole* 3D system must be transposed $xy$-to-$xz$, see Fig. 4.6c, across all nodes (an all-to-all communication event). In a similar fashion, for each of the three zero-padded vector components, each node works on local data, but in this case, just the $\hat{\mathbf{y}}$ stage of the DFTs, see Fig. 4.6e, and after an $xy$-to-$xz$ transpose, see Fig. 4.6f, the $\hat{\mathbf{x}}$ & $\hat{\mathbf{z}}$ stages of the DFTs are performed, see Figs. 4.6g & 4.6h, respectively. With both the tensor and vector components

in $xz$-plane configuration, the FD element-wise multiplication of the transforms, see Fig. 4.6l, and subsequent $\hat{\mathbf{z}}$ & $\hat{\mathbf{x}}$ stages of the iDFTs, are calculated, see Figs. 4.6j & 4.6k, respectively. Finally, in order to complete the iDFTs, an $xz$-to-$xy$ transpose, see Fig. 4.6l, returns the system to $xy$-plane configuration to permit the $\hat{\mathbf{y}}$ iDFTs, see Fig. 4.6m. In total, this requires three distributed transposes, two $xy$-to-$xz$ and one $xz$-to-$xy$. To complicate matters, the tensor and vector components arrays may be non-square, and thus, the distributed transposes must be able to cope seamlessly with either square or non-square systems. (Note: the reference 'square' is used rather than 'cubic', as for the design and construction of the transpose algorithms, it is the $\hat{\mathbf{y}}$ & $\hat{\mathbf{z}}$ dimensions which take precedence, with the $\hat{\mathbf{x}}$ dimension simply specifying the size of the data blocks). The final and most problematic restriction for the design and implementation of the distributed transposes is that they be performed in-place or as near to in-place as is possible. This is necessitated by the rationale that if the systems are large enough to warrant the application of parallel techniques, then memory should not be squandered by the use of out-of-place algorithms, which would seriously undermine the scalability and applicability advancements afforded by the custom-designed data distribution and parallel memory allocation schemes discussed earlier.

### 4.3.4 Message passing protocols

For the development of distributed algorithms, the overall efficiency is heavily dependent on the nature of the messages being sent. Small messages allow for easy in-place operation but the inherent associated message passing overheads can be substantial. On the other hand, larger messages reduce the communication overhead but can impair the requisite in-place nature of the transpose. A balance needs to be struck between the message size and the associated communication overhead so that the in-place transpose can be performed efficiently and with minimal overheads. With this in mind, it is important to be familiar with some of the finer details of the MPI implementation. For example, the different internal data exchange protocols for short and long message passing. Depending on the protocol used, both the message envelope and data (the envelope is relatively small & includes information such as the message source, destination, length, tag, and communicator) may be sent immediately & asynchronously to the receive node and the operation may complete without an acknowledgement from a matching receive.

Conversely, for a synchronous send, a 'handshake' occurs, where only the envelope alone is sent to inform the receive node that a send has been posted and an acknowledgement is required to indicate that the destination is ready before sending the data. The following is a summary of the main message passing protocols:

**Message passing protocols**

- **Eager protocol**: The eager protocol is ideal for smaller message sizes. Here, it is assumed that the receiving node can store the data and the envelope & data are sent immediately (as soon as a send is posted), regardless of whether or not a matching receive has been posted. The onus is on the receive node to buffer unexpected messages, and thus, although it offers low latency, if matching receives are not posted, it can result in the receive node being overwhelmed for large message sizes.

- **Rendezvous protocol**: The rendezvous protocol is ideal for larger message sizes as the send and receive nodes are synchronised for the transfer. When a send is posted, only the message envelope is sent to the receive node. When the receive has been posted and the appropriate envelope has been received, the receive node sends an acknowledgement to the send node (a request for the data) and only once the send node has received the request does the transfer of the message data take place. In this case, the receive node only has to buffer the envelopes.

- **Short protocol**: The short protocol is a slight variation of the eager protocol where both the envelope and data all fit into a single packet and the message data is sent with the envelope.

- **Get protocol**: The get protocol is used by shared-memory implementations or when there is special hardware support for remote memory operations. In this case data is read directly by the receiver, usually via a '*memcpy*' operation.

The switch-over point from short to long message handling, i.e., from eager to rendezvous protocols, is often referred to as the 'EAGER_LIMIT'. Each of the various MPI implementations define such a limit and have differing methods of manipulating the value. For MPICH1 [2, 208], the default value is 128000 Bytes but this can be changed at compile time by adjusting the value in the files:

1. *mpich-*'RELEASE'*/mpid/ch2/chcancel.c*
2. *mpich-*'RELEASE'*/mpid/ch2/chinit.c*
3. *mpich-*'RELEASE'*/mpid/ch_shmem/shmeminit.c*

For MPICH2 [3], this threshold is set by the value of *MPIDI_CH3_EAGER_MAX_MSG_SIZE* in the files *mpich2-*'RELEASE'*/src/mpid/ch3/channels/*/include/mpidi_ch3_post.h*. For Open MPI [4], which uses a Modular Component Architecture (MCA) framework for tuning Open MPI's run-time behaviour, the boundary can be controlled dynamically using the MCA parameters '*btl_tcp_eager_limit*' (default 64kB) and '*btl_sm_eager_limit*' (default 4kB), where '*btl*', '*tcp*' & '*sm*' stand for 'Byte-transfer layer (point-to-point byte movement)', 'Transmission control protocol' & 'Shared-memory', respectively. These MCA parameters can be passed as direct command line arguments to the MPI execution script '*mpirun*' or '*mpiexec*', '*–mca btl_tcp_eager_limit value*' & '*–mca btl_sm_eager_limit value*', they can be set as environment variables, '*export OMPI_MCA_<param_name>=value*', or they can be set in the default configuration file '$HOME/.openmpi/mca-params.conf'.

The disparate treatment of data where, depending on the size of the data to be communicated, different protocols are implemented, means that MPI developers must keep their wits about them. If not careful, code which seems to work flawlessly for small test sizes, may be prone to deadlock and be useless for larger systems. Even for code based on the non-blocking 'Isend' routine, deadlock can occur, as for a large message (rendezvous protocol), only the envelope is sent and the sender waits for the receiver to post a receive and collect the data. The 'Isend' returns immediately, however, the sender cannot access the send buffer until the send completes, which will not happen until the receive node has received the data. Thus, even though the send was non-blocking, the send process is blocked until a receive is posted. If one is not then deadlock occurs.

### 4.3.5 Local transposes

Local transposes may seem like a trivial concern, however, their fundamental role in the design and creation of efficient and portable distributed-transpose algorithms cannot be overstated. For the most part, their functionality is either directly incorporated into the distributed algorithm itself or the distributed algorithm is based on an abstraction of the local model. Distributed transposes, albeit a necessary evil, can have their impact and complexity reduced by the appropriate use of local computation. The use of local transposes instantly simplifies the subsequent

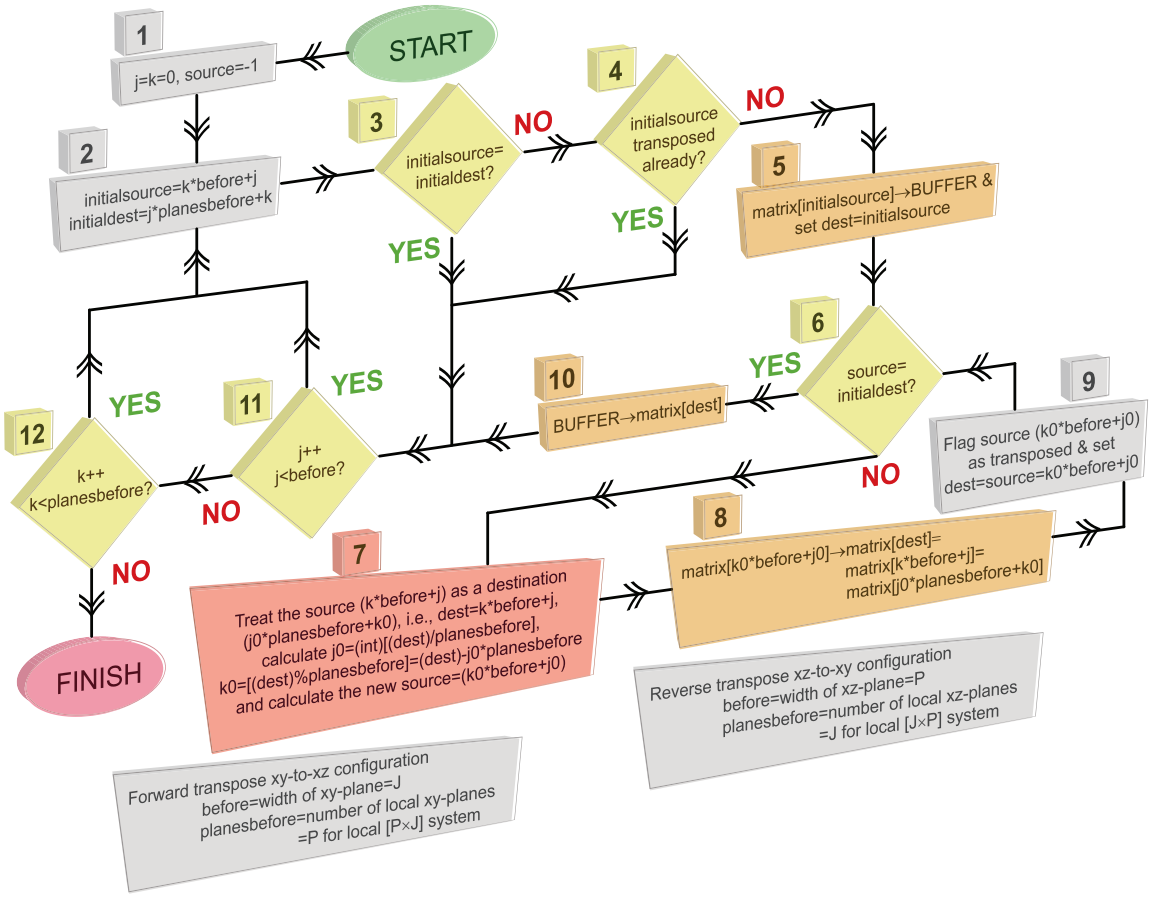



Figure 4.7: Flowchart for the local non-square transpose algorithm.

algorithms as, although the data is still distributed, the transposition aspect of the operation has been performed locally. The developer no longer has to be concerned with the communication and simultaneous transposition of the data 'on-the-fly' and can now focus on just the communication of the relevant data to the appropriate nodes.

The algorithmic implementation of a local transpose depends on the nature of the data to be transposed. If the local system is square, where as before, the reference 'square' is used rather than 'cubic', as it is the $\hat{\mathbf{y}}$ & $\hat{\mathbf{z}}$ dimensions which take precedence, with the $\hat{\mathbf{x}}$ dimension simply specifying the size of the data block, then the algorithm consists of a trivial flip across the diagonal. Please note that, although the discussion focusses on the local transposition of 2D $[P \times J]$ systems, the extension to 3D $[P \times J \times K]$ systems simply involves the replacement of the notion of the transposition of single elements with that of memory sections of length $K$. Square systems are

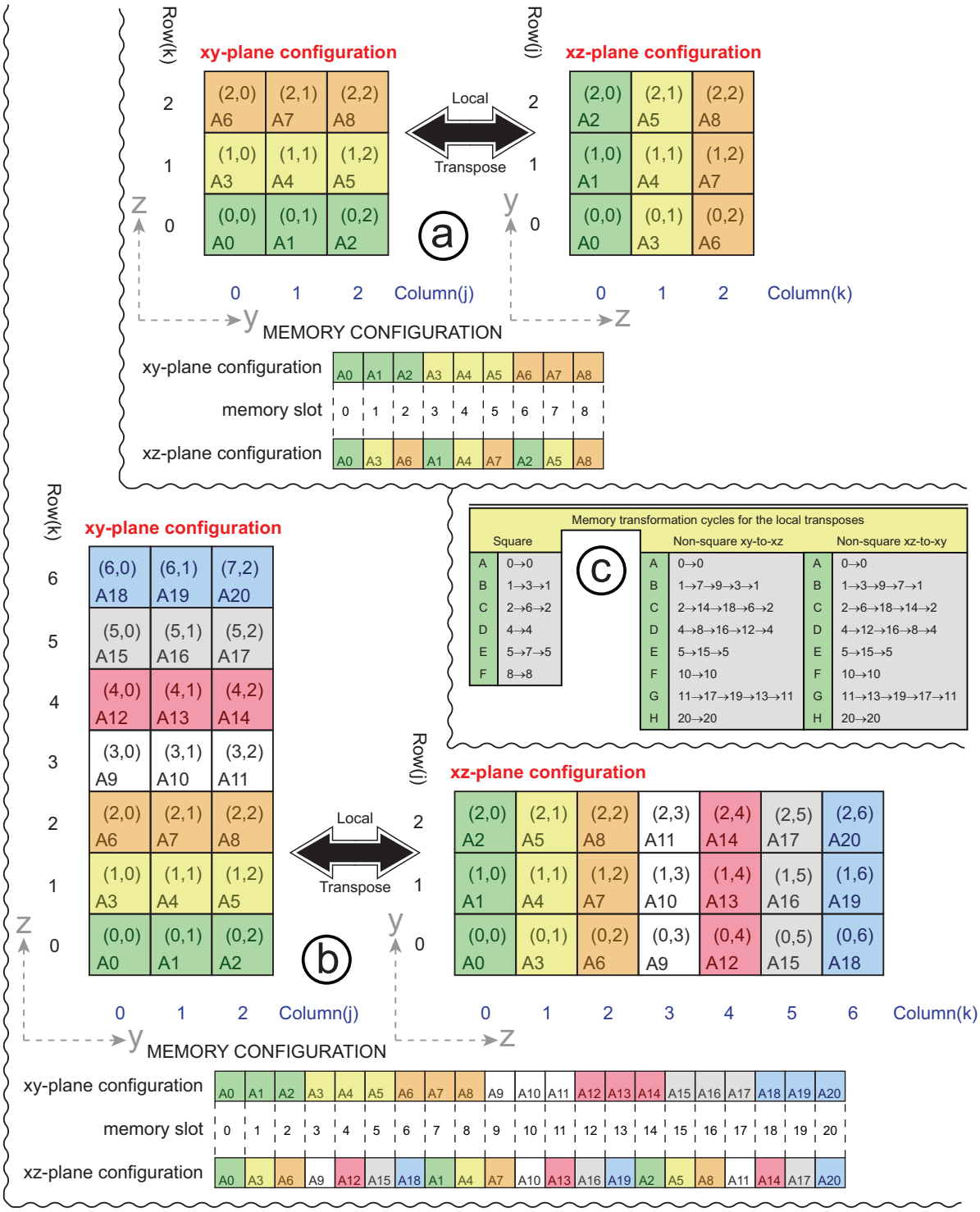



Figure 4.8: Memory layout for the $xy$-plane and $xz$-plane configurations for both the square $[P \times J = 3 \times 3]$ (a) & non-square $[P \times J = 7 \times 3]$ (b) local transposes and (c), the memory transformation cycles involved in each transpose.

all well and good, except for the fact that they are the extreme exception. If the requisite distributed-transpose algorithm is to integrate seamlessly with the rest of the prescribed parallel framework, it must be able to transparently cope with user-defined systems of arbitrary dimension which are decomposed across an arbitrary number of nodes. Any attempt to enforce a restriction that only data decompositions which would result in square data subsets on every constituent node would be foolhardy and only serve to subvert any and all advances made thus far. To circumvent the need for any such constraints, a custom local transpose for non-square systems had to be designed which could treat arbitrary system sizes and whose performance was comparable to that of the square algorithm. Although, a local transpose, either square or non-square, may not seem to be a noteworthy consideration, the complicating factor that warrants this discourse is the requirement that the complete distributed transpose, and thus, the underlying local transposes be done in-place. This stipulation negates the simple point-to-point nature associated with the out-of-place transpose and extreme care must be taken to ensure than no pertinent data is overwritten during the transpose operation.

The non-square local transpose algorithm, see Fig. 4.7 for a detailed flowchart, is based on an abstraction of the square algorithm. For the square $[P \times J = 3 \times 3]$ system depicted in Fig. 4.8a, the transformation of the memory layout, which is shown in the left side of Fig. 4.8c, is given by the relation, $[kJ + j] \rightleftharpoons [jP + k]$, $j = 0 \ldots J-1$, $k = 0 \ldots P-1$. As expected, due to the fact that the system is square, there is a one-to-one correspondence between the *before* and *after* cases, i.e., *memory location* 1 $\rightarrow$ *memory location* 3 and vice versa. For the case of a non-square system, the is still a correlation, however, since there is no longer a symmetry about a main diagonal, the one-to-one correspondence becomes a more complex cyclic relationship. Consider the non-square $[P \times J = 7 \times 3]$ system depicted in Fig. 4.8b. The memory layout is still governed by the same transformation, however, for the non-square case, $J \neq P$, and thus, taking the example of the forward $xy$-to-$xz$ transpose, although *memory location* 1 $\rightarrow$ *memory location* 7, the converse is not true. The disparity of $J$ and $P$ gives rise to the non-square transformation cycles shown in the right side of Fig. 4.8c.

Using the forward $xy$-to-$xz$ transpose for illustrative purposes, for both the square and non-square systems, the source location is given by $[kJ + j]$ and the destination by $[jP + k]$, i.e., what was in memory location $[kJ + j]$ *before* the transpose, is in memory location $[jP + k]$ *after* the transpose. In the case of the square system, which possesses the one-to-one correspondence, this transformation requires the

buffering of one of the entries for each transaction, i.e., $[kJ+j] \rightarrow$ BUFFER, $[jP+k] \rightarrow [kJ+j]$ and BUFFER $\rightarrow [jP+k]$. However, for the non-square system, the cyclic property means that only one buffer is required per cycle. A transposition cycle within the non-square algorithm begins with an '*initial source*' $[kJ+j]$ and destination $[jP+k]$. It buffers the '*initial source*' $[kJ+j] \rightarrow$ BUFFER and treats the '*initial source*' $[kJ+j]$ as if it were a destination $[j_0P+k_0]$ and calculates its corresponding source $[k_0J+j_0]$ by extracting $j_0$ & $k_0$ using $j_0 = (int)\,[(kJ+j)/P]$ & $k_0 = [(kJ+j)-j_0P] = [(kJ+j)\%P]$. It copies this new source to the destination $[k_0J+j_0] \rightarrow [j_0P+k_0] = [kJ+j]$, subsequently, treats this new source $[k_0J+j_0]$ as a destination $[j_1P+k_1]$, extracts $j_1$ & $k_1$ to calculate the corresponding source for the new transfer $[k_1J+j_1]$ and this is repeated until the whole sequence loops around on itself, i.e., when the source $[k_nJ+j_n] = [jP+k]$, which was the '*initial destination*'. To finish, the buffered data is copied to the '*initial destination*' BUFFER $\rightarrow [jP+k] = [k_nJ+j_n]$. To clarify this procedure, for the forward $xy$-to-$xz$ transpose, the following lists the simplified pseudo-code for the non-square transposition cycle '**B**' in Fig. 4.8c, i.e., [**1**$\rightarrow$**7**$\rightarrow$**9**$\rightarrow$**3** $\rightarrow$ **1**].

- The '*initial source*' is **1** and the '*initial destination*' is **7**
- Buffer the '*initial source*', [**1**$\rightarrow$BUFFER]
- Treat **1** as a *destination* and calculate its *source*, i.e., **3**, move [**3**$\rightarrow$**1**]
- Treat **3** as a *destination* and calculate its *source*, i.e., **9**, move [**9**$\rightarrow$**3**]
- Treat **9** as a *destination* and calculate its *source*, i.e., **7**, move [**7**$\rightarrow$**9**]
- Now the *source*=**7**='*initial destination*', copy [BUFFER$\rightarrow$**7**] = [**1**$\rightarrow$**7**]

The full transposition of this illustrative non-square $[P \times J = 7 \times 3]$ example, depicted in Fig. 4.8b, requires the five distinct transposition cycles, '**B**', '**C**', '**D**', '**E**' & '**G**' in Fig. 4.8c. The other three cycles, '**A**', '**F**' & '**H**', where the '*initial source*' is equal to the '*initial destination*', have no data movement and are ignored. The complete memory transformation details for the cycles involved in the forward $xy$-to-$xz$ transposition of the given example are shown in Table 4.1.

An interesting point, which gives further insight into the dynamics of the custom algorithm, is that the $j$ & $k$ values and hence, the '*initial source*' for the constituent cycles listed in Table 4.1, are not sequential, i.e., '**A**'='**0**', '**B**'='**1**', '**C**'='**2**', '**D**'='**4**', '**E**'='**5**', '**F**'='**10**', '**G**'='**11**' & '**H**'='**20**'. A fundamental requirement of the algorithm and that of any transpose algorithm, is that once an element has been transposed, it should not be touched again. This may seem like an obvious statement, however, it is a critical design issue. The custom algorithm loops through $j = 0 \ldots J-1$ &

| Local non−square memory transposition cycles | | | | | | | |
|---|---|---|---|---|---|---|---|
| | **j** | **k** | **source** | → | **destination** | $\mathbf{j_0}$ | $\mathbf{k_0}$ |
| | | | **kJ+j** | → | **jP+k** | **(int)([kJ+j]/P)** | **[(kJ+j)−j0∗P]=[(kJ+j)%P]** |
| **A** | **0** | **0** | 0 | → | 0 | **0** | **0** |
| **B** | **1** | **0** | 1 | → | 7 | 0 | 1 |
| | 0 | 1 | 3 | → | 1 | 0 | 3 |
| | 0 | 3 | 9 | → | 3 | 1 | 2 |
| | 1 | 2 | 7 | → | 9 | **1** | **0** |
| **C** | **2** | **0** | 2 | → | 14 | 0 | 2 |
| | 0 | 2 | 6 | → | 2 | 0 | 6 |
| | 0 | 6 | 18 | → | 6 | 2 | 4 |
| | 2 | 4 | 14 | → | 18 | **2** | **0** |
| **D** | **1** | **1** | 4 | → | 8 | 0 | 4 |
| | 0 | 4 | 12 | → | 4 | 1 | 5 |
| | 1 | 5 | 16 | → | 12 | 2 | 2 |
| | 2 | 2 | 8 | → | 16 | **1** | **1** |
| **E** | **2** | **1** | 5 | → | 15 | 0 | 5 |
| | 0 | 5 | 15 | → | 5 | **2** | **1** |
| **F** | **1** | **3** | 10 | → | 10 | **1** | **3** |
| **G** | **2** | **3** | 11 | → | 17 | 1 | 4 |
| | 1 | 4 | 13 | → | 11 | 1 | 6 |
| | 1 | 6 | 19 | → | 13 | 2 | 5 |
| | 2 | 5 | 17 | → | 19 | **2** | **3** |
| **H** | **2** | **6** | 20 | → | 20 | **2** | **6** |

Table 4.1: Memory transformation cycles for the non-square local forward $xy$-to-$xz$ transpose of the $[P\times J=7\times 3]$ example depicted in Fig. 4.8b.

$k=0\ldots P-1$ and to ensure that the algorithm is not stepping on its own toes and transposing previously transposed elements, a bit-fielded array, see §2.2, is used to indicate if a '*source*', which may or may not delimit the start of a unique transposition cycle, has already been transposed as part of a prior cycle. Once a single transpose transaction has been completed, the appropriate bit within the bit-fielded flag array is set to '**1**' to indicate that this '*source*' has been transposed and as a consequence, cannot be the seed for any future cycles. For each new $j$ and $k$ value, which yields a new '*initial source*', $[kJ+j]$, and '*initial destination*', $[jP+k]$, the algorithm tests the corresponding bit in the bit-fielded flag array and, if FALSE, i.e., the bit is '**0**', which indicates that this element has not been transposed, begins a new transposition cycle. Without this check, the transposition algorithm would automatically take each

| $P\times J=7\times3,\ j=0...J-1,\ k=0...P-1$ | | | |
|---|---|---|---|
| **k** | **j** | **source** [**kJ+j**] | **Operation/Comments** |
| **START** | | | |
| 0 | 0 | 0 | Skip cycle **A**, [**0→0**] |
| 0 | 1 | 1 | Do cycle **B**, [**1→7→9→3→1**] |
| 0 | 2 | 2 | Do cycle **C**, [**2→14→18→6→2**] |
| 1 | 0 | 3 | Skip **3**, transposed in cycle **B** |
| 1 | 1 | 4 | Do cycle **D**, [**4→8→16→12→4**] |
| 1 | 2 | 5 | Do cycle **E**, [**5→15→5**] |
| 2 | 0 | 6 | Skip **6**, transposed in cycle **C** |
| 2 | 1 | 7 | Skip **7**, transposed in cycle **B** |
| 2 | 2 | 8 | Skip **8**, transposed in cycle **D** |
| 3 | 0 | 9 | Skip **9**, transposed in cycle **B** |
| 3 | 1 | 10 | Skip cycle **F**, [**10→10**] |
| 3 | 2 | 11 | Do cycle **G**, [**11→17→19→13→11**] |

| **k** | **j** | **source** [**kJ+j**] | **Operation/Comments** |
|---|---|---|---|
| .....**CONTINUED**..... | | | |
| 4 | 0 | 12 | Skip **12**, transposed in cycle **D** |
| 4 | 1 | 13 | Skip **13**, transposed in cycle **G** |
| 4 | 2 | 14 | Skip **14**, transposed in cycle **C** |
| 5 | 0 | 15 | Skip **15**, transposed in cycle **E** |
| 5 | 1 | 16 | Skip **16**, transposed in cycle **D** |
| 5 | 2 | 17 | Skip **17**, transposed in cycle **G** |
| 6 | 0 | 18 | Skip **18**, transposed in cycle **C** |
| 6 | 1 | 19 | Skip **19**, transposed in cycle **G** |
| 6 | 2 | 20 | Skip cycle **H**, [**20→20**] |
| **FINISH** | | | |

Table 4.2: Logical flow of the complete non-square local forward $xy$-to-$xz$ transpose algorithm for the given $[P\times J=7\times3]$ example depicted in Fig. 4.8b.

element as the start of a new cycle and would incorrectly include/move elements that have already been transposed.

The earlier observation that the starting points for the transposition cycles '**A**' through '**H**' are not sequential is a direct manifestation of the custom algorithm neglecting previously transposed elements which would otherwise act as progenitors for forbidden transposition cycles. Table 4.2 shows the full logical flow of the complete algorithm for the forward $xy$-to-$xz$ transpose as applied to the given example depicted in Fig. 4.8b, whereby trivial, i.e., where the '*source*' equals the '*destination*', and forbidden, where the seed would be a previously transposed element, transposition cycles are omitted.

### 4.3.6 Exploiting the eager protocol

The final hurdle in the design and implementation of the full distributed DFT kernel is the task of devising a distributed-transpose algorithm which can handle completely arbitrary systems to complement the prescribed allocation and decomposition schemes. A critical factor in the overall design and construction is the layout and initial treatment of the data to be transposed. The chosen

starting point leads to two possible development avenues. The system can either be left as is and the development can proceed 'in-situ' or each node can perform a local transpose on its subset in preparation for communication to the appropriate processors. During the development process both approaches were considered, giving rise to two distributed algorithms. The first of two algorithms, which is based around an abstraction of the local theory described in the previous section, derives from the former 'in-situ' model and will be discussed here, whereas the second employs the pre-computation of a local transpose to simplify the host algorithm and will be described in the next section.

Although it is frowned upon and is considered unsafe parallel programming, in the right environment, exploiting the internal message passing protocols can prove beneficial. By designing a transpose algorithm that runs below the 'EAGER_LIMIT', see §4.3.4, one can avail of the inherent low latency of the protocol, however, in doing so, extreme care must be taken as the portability of the algorithm may become restricted. The custom algorithm uses the local theory developed in the previous section to construct a compact and elegant point-to-point distributed transpose that requires no explicit buffering or local precomputation, see Fig. 4.9 for a delineative flowchart. In the local case, the *source* and *destination* for a transaction were given by the transformation $[kJ+j] \rightleftharpoons [jP+k]\,, j=0\ldots J\text{–}1, k=0\ldots P\text{–}1$. For the distributed environment, by accounting for the fact that the data is decomposed across the available nodes, the local concepts can be extended to treat parallel systems. The use of 'cyclic-plane' decomposition means that in $xy$-plane configuration, plane $k$, $k=0\ldots P-1$, is located on processor $k\%np$, where $np$ is the number of processors and that it is plane $(int)\,(k/np)$ on that processor. Likewise, in $xz$-plane configuration, plane $j$, $j=0\ldots J-1$, is located on processor $j\%np$ and is plane $(int)\,(j/np)$ on that processor. This added complication of the data being physically located across the constituent nodes requires the reformulation of the original local transformation relation to that in Eq. 4.7.

$$\left[(int)\left(\frac{k}{np}\right)J+j\right] \text{ on processor } k\%np \;\rightleftharpoons\; \left[(int)\left(\frac{j}{np}\right)P+k\right] \text{ on processor } j\%np \tag{4.7}$$

To implement Eq. 4.7, each processor executes the same for-loop identically and, depending on the value of the control loop variable $i$, $i=0\ldots limit-1$, performs one

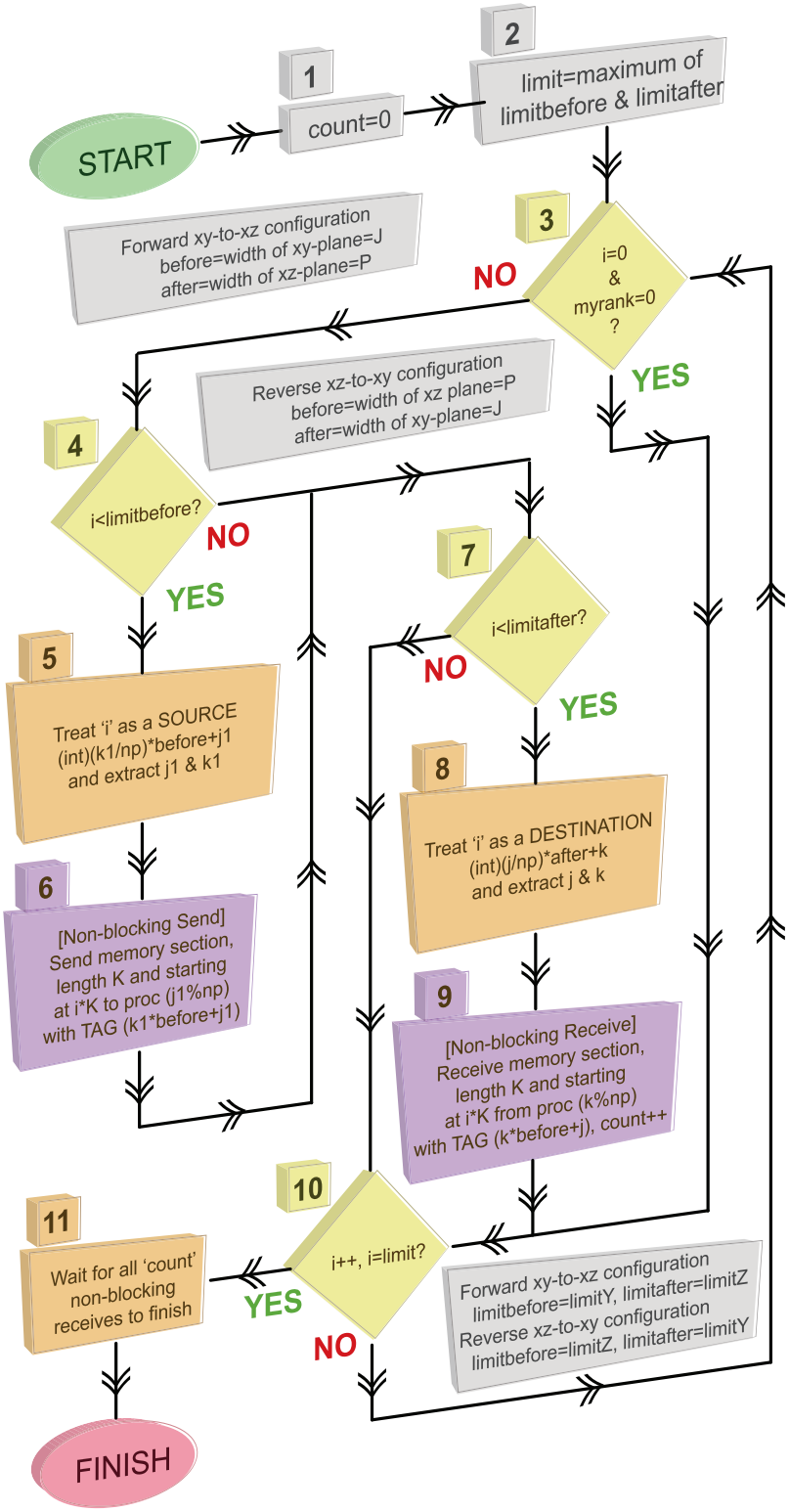



Figure 4.9: Flowchart for the 'in-situ' point-to-point distributed transpose.

*send* and/or *receive* per loop. *limit* is a local variable which stores the maximum number of elements for a specific processor *before* or *after* the distributed transpose. It is the maximum of $limitY$ and $limitZ$, which are the number of local elements on that processor in $xy$-plane and $xz$-plane configurations, respectively. Each node posts a *send* for each element that it has *before* the distributed transpose and a *receive* for each element that it will have *after* the distributed transpose, i.e., $limitY$ *send* and $limitZ$ *receive* operations for the $xy$-to-$xz$ distributed transpose and $limitZ$ *send* & $limitY$ *receive* operations for the $xz$-to-$xy$ distributed transpose.

The particulars of each *send* and *receive* transaction are given by treating each element in the local subset, in turn, as either a *source* or a *destination* and deducing the corresponding *destination* and *source* for that transfer. For the forward $xy$-to-$xz$ distributed transpose, each valid *send*, $i = 0 \ldots limitY - 1$, in the local subset, $i$ is treated as a *source*.

$$i = source = \left[ (int) \left( \frac{k_1}{np} \right) J + j_1 \right] \tag{4.8}$$

The corresponding *destination* for the transfer is then calculated by extracting the $j_1$ & $k_1$ values from Eq. 4.8. Extracting the $j_1$ value using $j_1 = i\%J$ is quite straightforward, however, the $k_1$ value requires a bit more work. The first step is the realisation that the integer cast in Eq. 4.8 can be expressed in terms of the *modulo* function.

$$\begin{aligned} &(int) \left( \frac{k_1}{np} \right) \equiv \frac{k_1 - k_1\%np}{np} \\ &\Rightarrow i = \left[ (int) \left( \frac{k_1}{np} \right) J + j_1 \right] \equiv \left[ \left( \frac{k_1 - k_1\%np}{np} \right) J + j_1 \right] \end{aligned} \tag{4.9}$$

Although, at first glance, it may seem that, since $k_1$ is the unknown and without $k_1$ it is not possible to compute $k_1\%np$, all that has been accomplished is that one indeterminacy has just been traded for another, $k_1\%np$ does not need to be calculated as it is equal to $myrank_{send}$, the rank of the sending processor. With this substitution Eq. 4.9 can be rearranged for $k_1$.

$$k_1 = \frac{(i - j_1)}{J} np + k_1\%np \equiv \frac{(i - j_1)}{J} np + myrank_{send}$$

Using this theory, processor $k_1\%np$ sends element $i$ to processor $j_1\%np$ using a non-blocking *send* with tag $k_1J + j_1$.

Meanwhile, each element $i = 0...limitZ - 1$ is treated as a *destination* and in an analogous manner to that of the prescribed *send*, the corresponding *source* is identified by extracting the $j$ & $k$ values.

$$i = destination = \left[(int)\left(\frac{j}{np}\right)P + k\right] \tag{4.10}$$

In this case $i\%P$ provides the relevant $k$ value and the $j$ value is obtained by replacing the offending integer cast in Eq. 4.10 with the appropriate *modulo*-based equivalent and the term $j\%np$ in the substitution with the communicator rank of the receiving processor, $myrank_{recv}$.

$$\begin{aligned}
&(int)\left(\frac{j}{np}\right) \equiv \frac{j - j\%np}{np} \\
&\Rightarrow i = \left[(int)\left(\frac{j}{np}\right)P + k\right] \equiv \left[(int)\left(\frac{j - j\%np}{np}\right)P + k\right] \\
&\Rightarrow j = \frac{(i-k)}{P}np + k\%np \equiv \frac{(i-k)}{J}np + myrank_{recv}
\end{aligned}$$

Here, processor $j\%np$ posts a non-blocking *receive* for element $i$ from processor $k\%np$ with tag $kJ + j$. This first-principles approach leads to a compact, hassle-free, well-balanced and extremely stable distributed algorithm that is capable of efficiently treating completely arbitrary systems. Each processor performs *send* and *receive* operations based solely on information derived from the loop over the memory locations of the local data, a sort of '*local loop for local data*' scenario whereby restrictive conditional statements for control localisation are not required.

The final step in the transpose algorithm is to ensure that the distributed transpose completes before any access to the pertinent data occurs. The post for the non-blocking *receive* only initiates a *receive* and may return before the operation completes. To safeguard the integrity of the data, once each processor has preformed the prescribed *send* and *receive* transactions for its subset, it must wait for its posted non-blocking receives to complete, at which point the relevant data has been received into the appropriate buffers and the transposed data is ready for use.

The only restriction for the use of the prescribed point-to-point distributed algorithm is that the message size be below the 'EAGER_LIMIT', see §4.3.4. For MPICH1 [208, 2], in its default configuration, the algorithm is as safe as houses as it will never encroach upon the default values of 128000 Bytes. Even for 16 Byte double-complex numbers, the $\hat{\mathbf{x}}$ dimension would have to be larger than 8000, which is highly unlikely for a 3D-based computational domain.

### 4.3.7 Moving beyond the protocols

**Although** the portability of the distributed DFT kernel is only mildly restricted by the requirements of the point-to-point distributed-transpose algorithm described in the previous section, the portability of the finished product as a whole is of paramount importance. Assuming the appropriate conditions exist, the prescribed algorithm is all well and good and its use can be quite advantageous. However, for all-encompassing portability, a truly robust distributed algorithm, that does not rely on the conventions of a specific MPI implementation, is required. The custom algorithm must be oblivious to the underlying protocols and must not be reliant on the immediate return of non-blocking routines to prevent deadlock. This can be easily checked by enforcing synchronous message passing by setting the 'EAGER_LIMIT', see §4.3.4, to zero.

The custom algorithm is based upon the use of local transposes in order to simplify the host algorithm and subsequent data communication. Each node performs a local transpose on its subset using the theory derived in §4.3.5. Here, as with previous sections, only the 2D theory will be discussed, with the extension to 3D being provided by replacing the idea of the transposition of single elements with that of memory sections of length $K$. The custom distributed-transpose algorithm has been designed to be able to treat completely arbitrary systems, and as such, contains a number of variables in order to described the system in full. The most important parameters for the following discussion are listed here.

- $np$ is the number of available processors.
- $myrank$ is the MPI rank which uniquely identifies each processor.
- **Forward xy-to-xz transpose:**
  - $before = J$ is the width of an $xy$-plane.
  - $after = P$ is the number of $xy$-planes globally.

- $planesbefore = planesY$ is the number of $xy$-planes locally in $xy$-plane configuration, see Eq. 4.1.
- $planesafter = planesZ$ is the number of $xz$-planes locally in $xz$-plane configuration, see Eq. 4.2.
- $recv = Znp$ is the base number of $xy$-planes for any node, see Eq. 4.1.

- **Reverse xz-to-xy transpose:**
  - $before = P$ is the width of an $xz$-plane.
  - $after = J$ is the number of $xz$-planes globally.
  - $planesbefore = planesZ$ is the number of $xz$-planes locally in $xz$-plane configuration, see Eq. 4.2.
  - $planesafter = planesY$ is the number of $xy$-planes locally in $xy$-plane configuration, see Eq. 4.1.
  - $recv = Ynp$ is the base number of $xz$-planes for any node, see Eq. 4.2.

To help with the documentation of the custom formalism, illustrative examples are depicted in Figs. 4.10 & 4.11, which show schematics for the forward $xy$-to-$xz$ and reverse $xz$-to-$xy$ distributed transposes of a $[P \times J = 13 \times 5]$ system across $np = 3$ nodes. For each, panel (a) & (b) show the initial pre-transpose and final post-transpose system layouts, respectively, including the memory configurations and data distributions due to the underlying 'cyclic-plane' decomposition. In both cases, for the $np = 3$ participating nodes, panel (c) shows the $[planesbefore \times before] \rightarrow [before \times planesbefore]$ local transposes. These transposes provide contiguous blocks ready for communication to the appropriate processors, and furthermore, permit the requisite in-place functionality through plane-by-plane interchange of the locally transposed planes. From here, the transposition aspect of the operation is complete and there is a one-to-one cyclic correspondence, governed by the *modulo* function & the number of processors, between the position of the now contiguous planes in the local subset and their target processor, i.e., for each processor, plane $i \rightarrow$ processor $[i\%np]$, $i = 0 \ldots before - 1$.

Although the planes are now in the correct post-transpose orientation and can be farmed out to the appropriate nodes using the *modulo* function, as is, the planes do not entirely conform to the required structure. The local transposes merely served to provide contiguous blocks to allow in-place transposition and to streamline the interprocessor communication. Every plane in the post-transpose configuration is composed of a single contribution from each of the active processors, self included.

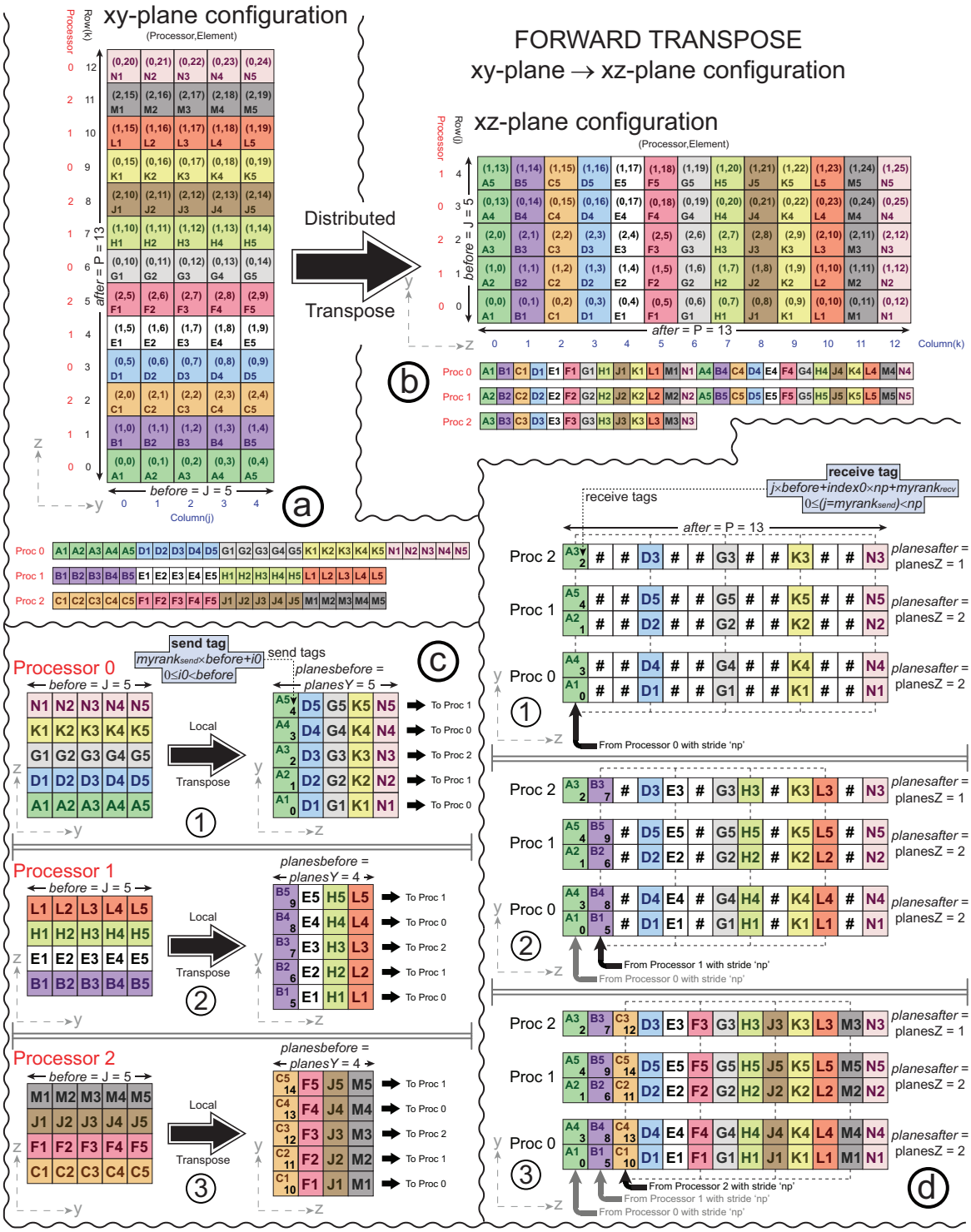



Figure 4.10: Schematic for the forward $xy$-to-$xz$ 'MPI_Type_vector'-based distributed transpose of a $[P \times J = 13 \times 5]$ system. (a) System in $xy$-plane configuration (b) System in $xz$-plane configuration (c) Local transposes (d) Strided receives.

In other words, each plane in the local subsets constitutes only a fraction of the full final plane and although they are sent to the pertinent nodes as contiguous blocks, to comply with the final structural requirements, the data must be received as a vector with $stride = np$, i.e., stride equal to the number of active processors. This is implemented via the '*MPI_Type_vector*' routine which allows the creation of custom derived datatypes, based on a primitive MPI datatype, with an arbitrary number of blocks and with block sizes and strides that are defined as multiples of the chosen building block. On the receiving end, for a specific plane, these strided vectors are received into memory locations beginning at a starting point which is derived from the unique communicator rank of the sender, $myrank_{send}$.

These steps are clearly evident for the chosen examples in Figs. 4.10 & 4.11. For both, the right side of panel (c) shows the [*before*×*planesbefore*] post local transpose layouts of the local subsets. In this state, each subset contains the exact same number of planes, which are destined for communication, via the *modulo* function, to the nodes listed just to the right of each [*before*×*planesbefore*] block in panel (c). Each plane is sent as a contiguous block of width *planesbefore* and is received, starting at the memory location given by the rank of the sending processor, as a vector with $stride = np = 3$, i.e., planes received from processors 0, 1 & 2 start at memory locations 0, 1 & 2 in the relevant plane, respectively, see panels (d,1), (d,2) & (d,3).

This is fine in theory, however, for the practical application of the prescribed algorithm, the devil really is in the detail and several key issues/complications arise which need to be addressed. The first obstacle to the implementation of the algorithm is that the value of *planesbefore*, the width of the contiguous blocks to be communicated, is not a constant across the distributed environment. The use of the local transposes transformed the [*planesbefore*×*before*] subsets to [*before*×*planesbefore*], see panel (c). This produced contiguous blocks of width *planesbefore*, however, the value of *planesbefore*, which gives the number of local planes for the initial pre-transpose configuration, $planesY$ in $xy$-plane configuration and $planesZ$ in $xz$-plane configuration, see Eqs. 4.1 & 4.2, has two distinct rank-dependent values. If the rank of the host processor is below a certain threshold then *planesbefore* is one unit wider than it is otherwise. This can be seen explicitly in either of the illustrative forward $xy$-to-$xz$ or reverse $xz$-to-$xy$ distributed transposes in Figs. 4.10 & 4.11. In the forward case in Fig. 4.10, the value of [*planesbefore*=$planesY$= 5 ], see Fig. 4.10(c,1), for processor 0, differs from the [*planesbefore*=$planesY$= 4 ] for processors 1 & 2, see Fig. 4.10(c,2) & (c,3). The situation is similar in the reverse case in Fig. 4.11, where

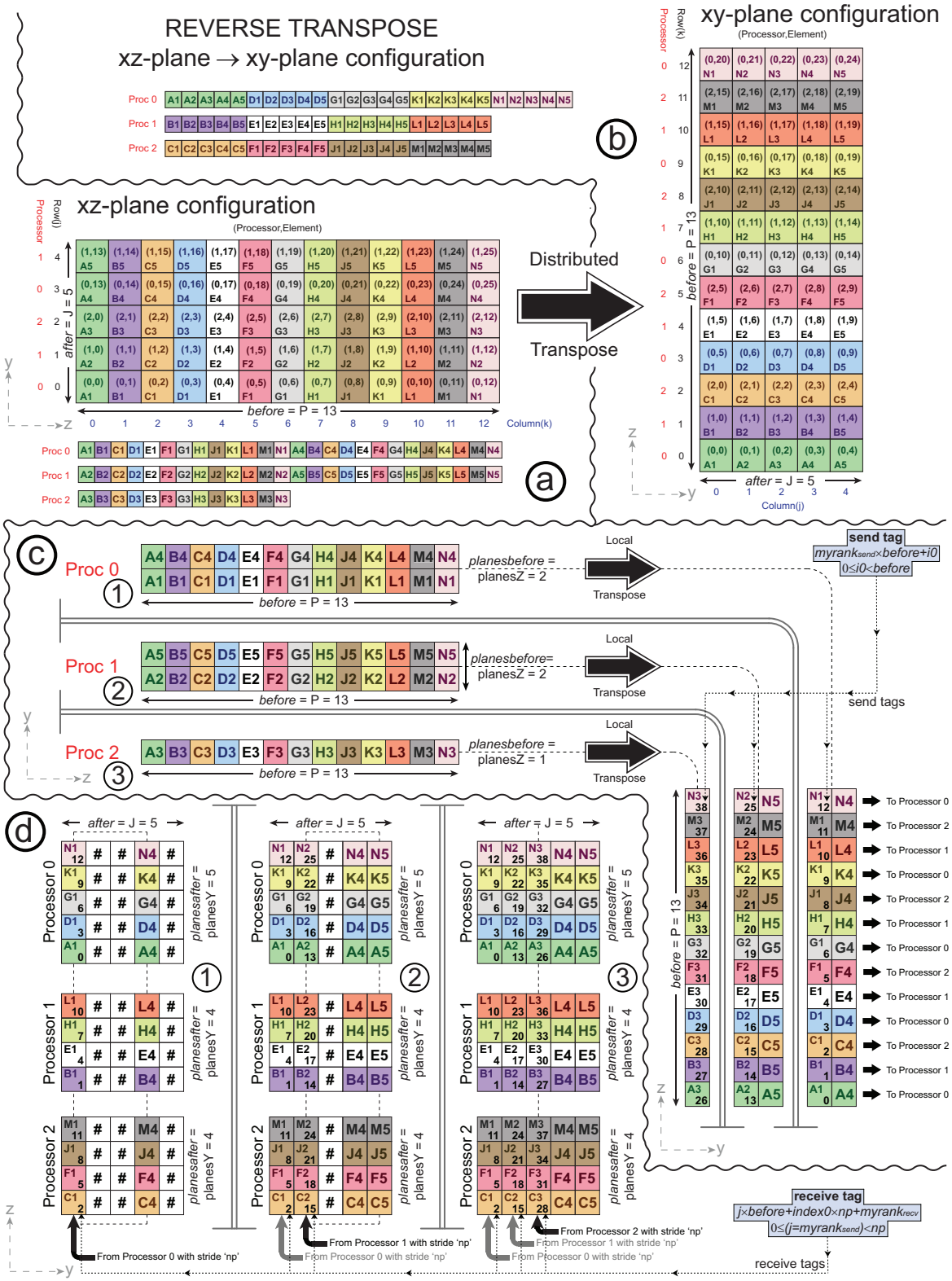



Figure 4.11: Schematic for the reverse $xz$-to-$xy$ 'MPI_Type_vector'-based distributed transpose of a $[J \times P = 5 \times 13]$ system. (a) System in $xz$-plane configuration (b) System in $xy$-plane configuration (c) Local transposes (d) Strided receives.

the value [*planesbefore*=*planesZ*= 1 ], see Fig. 4.11(c,3), for processor 2, differs from the [*planesbefore*=*planesZ*= 2 ] for processors 0 & 1, see Fig. 4.11(c,1) & (c,2).

This disparity in the value of *planesbefore* is a direct manifestation of the 'cyclic-plane' decomposition. In the initial pre-transpose state, if the rank of the host processor is less than $[after\%np]$, $[after = P]$ for an $xy$-to-$xz$ transpose & $[after = J]$ for an $xz$-to-$xy$ transpose, then the processor has [*planesbefore*=$recv$+1 ] planes and [*planesbefore*= $recv$ ] planes otherwise, where $recv$ is the base number of planes for any node, i.e., $recv = Znp$ in $xy$-plane configuration and $recv = Ynp$ in $xz$-plane configuration, see Eqs. 4.1 & 4.2. Of course, after the local transpose of the data subsets, these values become the width of the contiguous blocks, see panel (c). The solution to this is to construct two separate strided vector datatypes, one with $recv$ blocks and the other with $recv$+1 and, depending on the rank of the sending processor, to use the appropriate datatype, i.e.,

$$number\ of\ blocks \quad = \quad \begin{cases} recv + 1, & \text{if } myrank_{send} < after\%np; \\ recv, & \text{if } myrank_{send} \geq after\%np. \end{cases} \tag{4.11}$$

The second and more formidable obstacle is the requirement that the distributed transpose be done in-place. To safeguard against the loss of pertinent data, the in-place functionality requires that the memory locations for the receives must be cleared before the data is received. For the custom point-to-point distributed algorithm derived in the previous section, this was implemented through exploitation of the eager protocol. Using this protocol, messages are sent immediately, in their entirety, as soon as the send is posted and, after a quick check to verify that the data has been copied out of the send buffer, the send buffer is accessible for the requisite receives. Here, there is no such luxury and to circumvent the reliance on specific message passing protocols, the in-place functionality is provided by explicit buffering of the pertinent data, i.e., the sends take place from the external buffer.

Within the transpose algorithm, the main control loop has three independent local variables, [ $i = 0 \ldots before-1$ ], [ $i0 = 0 \ldots before-1$ ] & [ $index0 = 0 \ldots planesafter-1$ ], which control the buffering, sending & receiving of planes on the local processor, respectively. $i$ & $i0$ are concerned with the control of the *before* planes of width *planesbefore* in the pre-communication and post-local transpose configuration of the system, i.e., the right side of panel (c). On the other hand, $index0$ deals with the *planesafter* planes of width *after* in the final post-transpose system layout, see panel

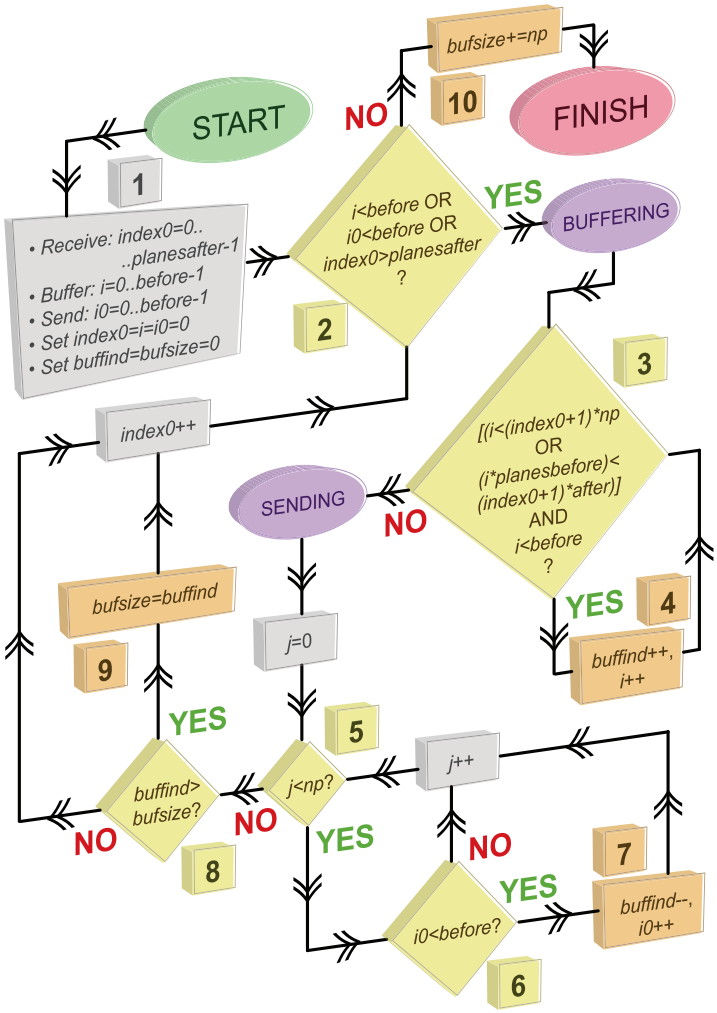



Figure 4.12: Flowchart for the calculation of the buffer requirements for the main 'MPI_Type_vector'-based distributed transpose.

(d). Each of these is made up of a single contribution from each of the active processors, self included, and thus, for each one, there is a communication round involving *np* contiguous sends, one from each processor, and *np* strided receives. As with the value of *planesbefore*, due to the 'cyclic-plane' decomposition, the value of *planesafter* is rank-dependent. Processors with rank [ $myrank < after\%np$ ] will have one more plane that those with [ $myrank \geq after\%np$ ]. As a result, for some communication rounds, processors may only be sending and not receiving data.

To allow the transposition to be carried out in-place, the algorithm dynamically buffers the planes that are to be sent for the current communication round and also buffers enough planes to ensure that the data can be received in-place, i.e., buffers the

current data that is occupying the memory locations and impeding the receives. This buffering is a local concern and each node independently pre-calculates the minimum size of the buffer that it required by performing a dry run of the transpose algorithm without any message passing. A flowchart for the buffer calculation algorithm is shown in Fig. 4.12.

It is often the case that, in order for the operation to be done in-place, for a specific communication round, more planes need to buffered than are to be sent in that round. This gives rise to the possibility of a slight accumulation of planes in the external buffer. Take the example of the forward $xy$-to-$xz$ transpose in Fig. 4.10. Here, for processor 0, [$planesbefore = 5$], whereas for processors 1 & 2, [$planesbefore = 4$]. This means that, for the first communication round, [$index0 = 0$], processor 0 only needs to buffer 3 planes of width [$planesbefore = 5$] in order to receive one planes of width [$after = 13$], whereas to receive the same plane, since [$planesbefore = 4$] on processors 1 & 2, they have to buffer 4 planes. With only 3 sends per processor in this communication round, processors 1 & 2 are each left with one unsent plane, see Table 4.3.

The algorithm cycles through the requisite communication rounds, and for each, counts how many planes of width *planesbefore* need to be buffered and subtracts the number of planes that are to be sent in that round. To implement this, the boolean expression [$i < (index0 + 1) \times np$] ensures that the buffering includes all the requisite planes that are to be sent in the communication round, ($i \times planesbefore$) < [$(index0 + 1) \times after$] ensures that all the planes of width *planesbefore* that are blocking the memory locations and impeding the receives are buffered and [$i < before$] ensures that the algorithm is not trying to buffer planes beyond the *before* local planes of width *planesbefore* that exist in this post-local transpose and pre-communication configuration, see Fig. 4.12(3). The value returned by the buffer algorithm is the buffer accumulation and does not include the memory required to buffer the data that is actually sent in each round, i.e., for the ideal case where there is no accumulation and all data that is buffered in each round is sent in that round, the buffer calculation algorithm returns a buffer requirement of zero, however, the base value for the dynamic sends is still required. However, since each plane, $index0$, is composed of a single contribution from each of the $np$ processors, the maximum number of sends that any processor may have to perform in any communication round is $np$ planes of width *planesbefore*, and thus, for any node, the maximum number of planes that is buffered at any time during the transposition, *bufsize*, is equal to $np$ plus the locally

| | BUFFER CALCULATION | | |
|---|---|---|---|
| | **Processor 0** | **Processor 1** | **Processor 2** |
| | Plane width, *planesbefore*=5 | Plane width, *planesbefore*=4 | Plane width, *planesbefore*=4 |
| **A** | *index0*=0, *i0*=0, *i*=0 | *index0*=0, *i0*=0, *i*=0 | *index0*=0, *i0*=0, *i*=0 |
| | Buffer plane *i*=0 ⇒ *buffind*=1 | Buffer plane *i*=0 ⇒ *buffind*=1 | Buffer plane *i*=0 ⇒ *buffind*=1 |
| | Buffer plane *i*=1 ⇒ *buffind*=2 | Buffer plane *i*=1 ⇒ *buffind*=2 | Buffer plane *i*=1 ⇒ *buffind*=2 |
| | Buffer plane *i*=2 ⇒ *buffind*=3 | Buffer plane *i*=2 ⇒ *buffind*=3 | Buffer plane *i*=2 ⇒ *buffind*=3 |
| | | Buffer plane *i*=3 ⇒ *buffind*=4 | Buffer plane *i*=3 ⇒ *buffind*=4 |
| | Send plane *i0*=0 ⇒ *buffind*=2 | Send plane *i0*=0 ⇒ *buffind*=3 | Send plane *i0*=0 ⇒ *buffind*=3 |
| | Send plane *i0*=1 ⇒ *buffind*=1 | Send plane *i0*=1 ⇒ *buffind*=2 | Send plane *i0*=1 ⇒ *buffind*=2 |
| | Send plane *i0*=2 ⇒ *buffind*=0 | Send plane *i0*=2 ⇒ *buffind*=1 | Send plane *i0*=2 ⇒ *buffind*=1 |
| | **3 in buffer & 3 sent** | **4 in buffer & 3 sent** | **4 in buffer & 3 sent** |
| | *index0*=0 accumulation: 0 | *index0*=0 accumulation: 1 | *index0*=0 accumulation: 1 |
| **B** | *index0*=1, *i0*=3, *i*=3 | *index0*=1, *i0*=3, *i*=4 | *index0*=1, *i0*=3, *i*=4 |
| | Buffer plane *i*=3 ⇒ *buffind*=1 | Buffer plane *i*=4 ⇒ *buffind*=2 | Buffer plane *i*=4 ⇒ *buffind*=2 |
| | Buffer plane *i*=4 ⇒ *buffind*=2 | | |
| | Send plane *i0*=3 ⇒ *buffind*=1 | Send plane *i0*=3 ⇒ *buffind*=3 | Send plane *i0*=3 ⇒ *buffind*=3 |
| | Send plane *i0*=4 ⇒ *buffind*=0 | Send plane *i0*=4 ⇒ *buffind*=2 | Send plane *i0*=4 ⇒ *buffind*=2 |
| | **2 in buffer & 2 sent** | **2 in buffer & 2 sent** | **2 in buffer & 2 sent** |
| | *index0*=1 accumulation: 0 | *index0*=1 accumulation: 0 | *index1*=0 accumulation: 0 |
| | **Total accumulation: 0 planes** | **Total accumulation: 1 plane** | **Total accumulation: 1 plane** |
| | **Total buffer: 0+*np*=3 planes** | **Total buffer: 1+*np*=4 planes** | **Total buffer: 1+*np*=4 planes** |

Table 4.3: Calculation of the buffer requirements for the forward $xy$-to-$xz$ distributed transpose across $np = 3$ processors of the given $P \times J = 13 \times 5$ example in Fig. 4.10. [**A**] Communication round 0, [**B**] Communication round 1.

calculated accumulation.

Once the size of the local buffer has been established, the pertinent data is copied to and extracted from the buffer using the *modulo* function, i.e., for buffering, plane $i \rightarrow buffer[\, i\%bufsize\,]$, and for sending, plane $i0$ will be plane $[\, i0\%bufsize\,]$ in the buffer. This ensures that, since *bufsize* is often little bigger than *np*, the data that is in the buffer, accumulated or not, does not get overwritten.

To summarise the progress thus far, the first task involved each processor performing a local transpose in order to simplify the subsequent algorithm. Two strided MPI datatypes were created, one with *recv* elements and the other with $recv + 1$, to deal with the fact that, although the data is sent as contiguous planes, to comply

with the final data layout, the data is received with $stride = np$, and furthermore, to account for the possibility of two distinct rank dependent message sizes. To enable the transposition to be carried out in-place, each processor then independently calculated the maximum number of planes that it would have to buffer during the operation.

With these prerequisites formalised, the rest of the algorithm can take shape. Fig. 4.13 shows a complete flowchart for the communication phase of the algorithm. For both the buffer calculation and main transpose algorithm, the main control loop boolean expressions $[\, i < before \,]$, $[\, i0 < before \,]$ & $[\, index0 < planesafter \,]$, see Fig. 4.13(2) & 4.12(2), ensure that the algorithm does not overrun its local bounds, and more importantly, that all data that is to be buffered, sent or received gets included. If any one boolean condition is removed, then since the algorithm is designed to operate with systems of completely arbitrary dimensions decomposed across an arbitrary number of processors, although the algorithm might be stable for a certain subset of cases, not all scenarios will be treated correctly, i.e., although a processor might post receives for data, the pertinent data might never have been buffered and sent. For the forward $xy$-to-$xz$ transpose in Fig. 4.10, where, $0 \leq index0 < planesafter$, for processors 0 & 1, $planesafter = 2$, whereas for processor 2, $planesafter = 1$, see panel (d,3). Processor 2 does not have a second plane corresponding to $index0 = 1$, and as a result, without the checks $[\, i < before \,]$ & $[\, i0 < before \,]$, using just the receive condition $[\, index0 < planesafter \,]$, the processor would not buffer or send the data contributions for the second plane, $index0 = 1$, on processor 0 & 1. Deadlock would occur as, although processor 0 would buffer and send [**A4**,**D4**,**G4**,**K4**,**N4**] to processor 0 & [**A5**,**D5**,**G5**,**K5**,**N5**] to processor 1, and processor 1 would buffer and send [**B4**,**E4**,**H4**,**L4**] to processor 0 & [**B5**,**E5**,**H5**,**L5**] to processor 1, processor 2 would never buffer and send [**C4**,**F4**,**J4**,**M4**] to processor 0 & [**C5**,**F5**,**J5**,**M5**] to processor 1.

A key design issue, which has not been mentioned until now, is the send and receive MPI message tags. These tags play a crucial and often understated role. The message tag is an arbitrary non-negative integer value which uniquely identifies a message. Each send that is posted is given such a tag and, in order for the message to be received, a receive with a matching tag must be posted. To ensure that the correct data is received, tags allow the receiver to differentiate between and restrict the receipt of messages from the same source. The theory thus far only describes, the source & destination, the send & receive buffers, and the mes-

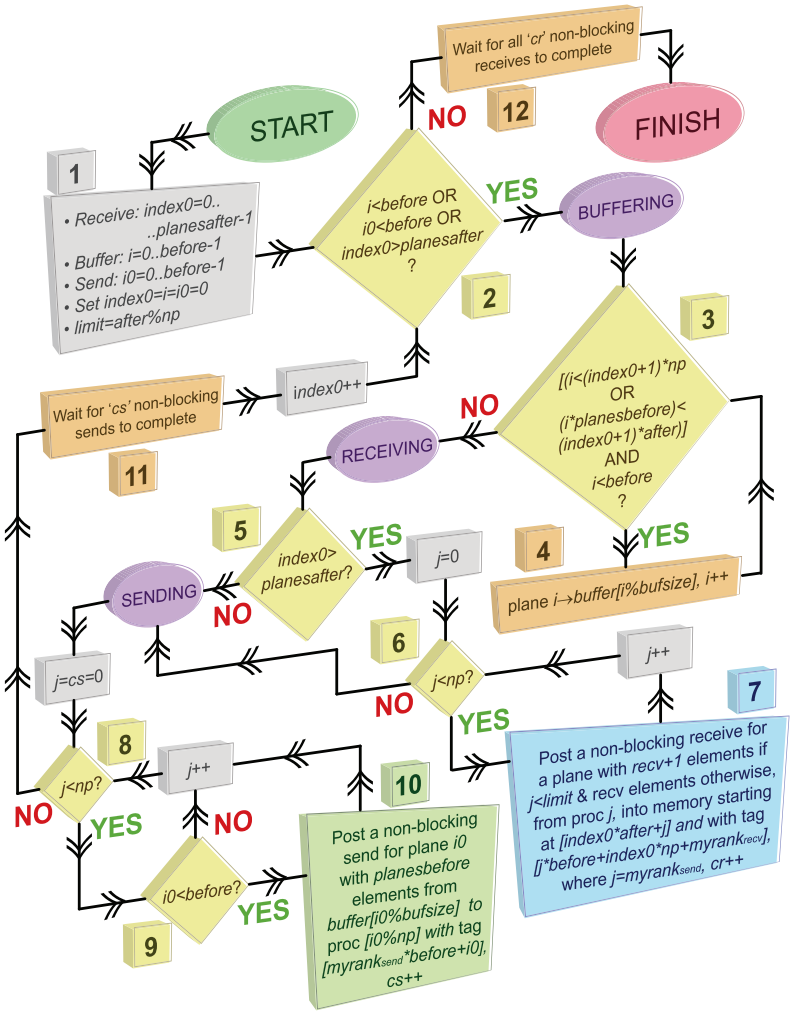



Figure 4.13: Flowchart for the communication phase of the main 'MPI_Type_vector'-based distributed transpose.

sage sizes. It does not specify which message will actually be received. This is facilitated by the use of message tags. For the custom algorithm, the send tag is given by the expression $[\, myrank_{send} \times before + i0 \,]$. This ensures that, since each of the $np$ processors has exactly $before$ planes of width $planesbefore$, each message has a unique tag in the range $[0 \leq tag < (np \times before)]$, i.e., processor 0 has tags $[0 \leq tag < before]$, processor 1 has tags $[before \leq tag < (2 \times before)]$ and processor $[np - 1]$ has tags $[((np - 1) \times before) \leq tag < (np \times before)]$. The receive tags are slightly more complicated and are given by $[\, myrank_{send} \times before + index0 \times np + myrank_{recv} \,]$. The first term relates to the rank of the source processor, i.e., if a message is being received from a processor with rank $myrank_{send}$, then the tag is in

the range $[(myrank_{send} \times before) \leq tag < ((myrank_{send} + 1) \times before)]$. The second term takes account of the fact that, due to the use of the *modulo* function in the sending process, on each processor, there is a difference of $np$ in the tag value for planes destined for the same processor. This can be seen explicitly in Figs. 4.10(c) & 4.11(c), where on each processor, the send tag for planes that are destined for the same processor differ by a factor of $np = 3$. The last term is a direct manifestation of the fact that the planes are sent cyclically using the *modulo* function, $(i0\%np) = destination = myrank_{recv}$. Thus, by adding $myrank_{recv}$ to the receive tag, each processor can filter out only the messages that are relevant to it.

The communication phase involves each processor buffering the planes that are impeding the receives and the planes that are to be sent in the current communication round, see Fig. 4.13(3→4). Plane $i0 = 0 \ldots before - 1$, is sent as a contiguous block of width *planesbefore* to processor $[i0\%np]$ with tag $[\, myrank_{send} \times before + i0 \,]$. The data is received from processor $j$ with $[stride = np = 3]$, into memory starting at location $[\, index0 \times after + j \,]$, with tag $[\, j \times before + index0 \times np + myrank_{recv} \,]$, where, $[0 \leq (j = myrank_{send}) < np]$, and with the number of elements given by Eq. 4.11.

For each plane $[\, index0 < planesafter \,]$, since each is composed of a single contribution from all processors, self included, $np$ receives are posted, see Fig. 4.13(5→7), after which, each processor posts non-blocking sends for the buffered data relevant to communication round, see Fig. 4.13(8→10). The algorithm then waits for just these posted sends to complete, see Fig. 4.13(11), so that the buffer is ready to start accepting planes for the next communication round. Once this has been done for each of the *planesafter* planes on each processor, the algorithm waits for *all* non-blocking receives to complete, see Fig. 4.13(12), at which point the full distributed transpose is complete.

### 4.3.8 Various distributed-transpose algorithms

As was discussed in §4.3.3, the distributed implementation of the OpenDDA framework requires not one, but two 3D DFTs and a 3D iDFT to be performed on the data that is plane-wise decomposed across the active nodes in the parallel environment. Since, by definition, each node has local knowledge of only two of the three requisite Cartesian directions, *full*-system 3D distributed transposes are needed to facilitate these transforms.

The system is initially in $xy$-plane configuration, i.e., decomposed across the available nodes along the $\hat{\mathbf{z}}$ direction. In this configuration, each node performs the $\hat{\mathbf{x}}$ & $\hat{\mathbf{y}}$ DFTs of the six independent tensor components and just the $\hat{\mathbf{y}}$ DFTs of the three zero-padded vector components. To permit the completion of the 3D DFTs, the *whole* system is then transposed from $xy$-to-$xz$ configuration. With the system in $xz$-plane configuration, i.e., decomposed across the available nodes along the $\hat{\mathbf{y}}$ direction, the $\hat{\mathbf{z}}$ DFTs of the tensor components and the $\hat{\mathbf{x}}$ & $\hat{\mathbf{z}}$ DFTs of the vector components are performed. After the FD element-wise multiplication of the transforms of the tensor and vector components and the $\hat{\mathbf{z}}$ & $\hat{\mathbf{x}}$ iDFTs of the resultant, an $xz$-to-$xy$ transpose returns the system to $xy$-plane configuration to permit the $\hat{\mathbf{y}}$ iDFTs of the resultant.

This algorithm requires three distributed transposes, two $xy$-to-$xz$, one for the DFT of the six independent tensor components and one for the DFT of the three zero-padded vector components, and one $xz$-to-$xy$, for the iDFT of the resultant of the FD element-wise multiplication of the transforms of the tensor and vector components. Based on the prescribed custom-designed distributed transpose theory in §4.3.5 → §4.3.7, the new OpenDDA framework incorporates three different transpose algorithms, and for each one, the user can choose whether to transpose the three vector or six tensor components separately or in a single conjoint operation. All three algorithms work in-place and are capable of transposing completely arbitrary systems sizes decomposed across an arbitrary number of nodes. The following lists the different options that are available for the distributed-transpose algorithm (set by choosing $0, 1, \ldots, 5$ in the '*control.input*' file, see the section on selecting the distributed-transpose algorithm at the end of §5.3):

- COMPONENTS TRANSPOSED SEPARATELY
  - **0:** Transposes a 3D array line-by-line
    This algorithm is based on the exploitation of the 'EAGER_LIMIT' as prescribed in §4.3.6, and consequently, is only guaranteed to work with MPICH1, (not MPICH2 & Open MPI), in their default configurations, although as described in §4.3.4, it is quite straightforward to alter the defaults via command line arguments, environment variables or implementation specific control parameters.
  - **1:** Transposes a 3D array plane-by-plane (local transpose & implicit MPI buffering)
    This is a legacy algorithm that arose as an intermediate stepping stone during the development phase of algorithm **2**. It is basically algorithm **2** without explicit buffering. This means that it is reliant on the 'EAGER_LIMIT'. The only reason that algorithm **0** gets away with exploiting the 'EAGER_LIMIT' is that it transposes the 3D system line-by-line, and thus, the message sizes are guaranteed to be small compared to the 'EAGER_LIMIT'. However, this algorithm is based on the plane-by-plane theory prescribes in §4.3.7, and as such, the individual message sizes cannot be guaranteed to be

small compared to the 'EAGER_LIMIT'. Therefore, algorithms **1** & **4** will only work for very small system sizes and should not be considered for practical use.

- **2:** Transposes a 3D array plane-by-plane (local transpose & explicit buffering)
  This algorithm is based on the plane-by-plane transposition with explicit buffering as prescribed in §4.3.7, and as such, is fully compliant with the MPI standard, unlike algorithms **0:** & **1:**, is not reliant on the specifics of the internal message passing protocols and is guaranteed to work in all scenarios, i.e., the safe bet.

- COMPONENTS TRANSPOSED SIMULTANEOUSLY
  - **3:** Variant of **0** in which all components are transposed in a conjoint operation
  - **4:** Variant of **1** in which all components are transposed in a conjoint operation
  - **5:** Variant of **2** in which all components are transposed in a conjoint operation

As will be seen later in §6, it turns out that algorithms **0** & **3** tend to be much faster than algorithms **2** & **5**. This seems counter-intuitive as one would expect that with fewer large messages, the plane-by-plane transposition of algorithms **2** & **5** would have the edge over the line-by-line transposition of algorithms **0** & **3**. However, from empirical evidence, this turns out not to be the case. Although algorithms **0** & **3** have the specific restriction that either MPICH1 must be available in its default configuration or the 'EAGER_LIMIT' in whichever implementation is being employed must be set to an appropriately high value, where MPICH1's default value of 128000 Bytes seems to be more than adequate, the associated performance benefits outweigh any and all associated inconvenience. Anyway, this is not considered to be an issue for concern as MPICH1 is trivial to obtain & install and any self-respecting computing facility worth its salt should have a default MPICH1 install as a matter of course.

*"I've come up with a set of rules that describe our reactions to technologies:*
*1. Anything that is in the world when you're born is normal and ordinary and is just a natural part of the way the world works.*
*2. Anything that's invented between when you're fifteen and thirty-five is new and exciting and revolutionary and you can probably get a career in it.*
*3. Anything invented after you're thirty-five is against the natural order of things."*

**Douglas Adams**

# 5 Code description

## 5.1 Introduction

**OpenDDA** is written completely from scratch in the C language. Apart from the fact that it is the author's favourite language, C was chosen as the ANSI standard is supported by all modern C compilers, C is highly portable, extremely efficient, easy to program and is the language of choice for many of the state-of-the-art high performance scientific libraries, i.e., FFTW[13] [177, 178], GSL[14] [180], Mersenne Twister[15] [204, 203], PETSc[16] [209], ATLAS[17] [210], etc.

During the initial design and blueprinting phase, several prime directives were prescribed to drive and guide the rest development process. **Directive I** was fairly standard and involved making a concerted effort, at every stage of development, to maximise both the performance and portability of each of the constituent components and hopefully as a result, the finished code as a whole. **Directive II** was to minimise the reliance on external libraries; which meant custom building as much of the requisite functionality as possible, as long as it did not conflict with the first directive. **Directive III** was to maximise the readability and maintainability of the code without conflicting with the first or second directives. This involved the minimisation

[13]Fastest Fourier Transform in the West (FFTW), http://www.fftw.org
[14]GNU Scientific library (GSL), http://www.gnu.org/software/gsl
[15]Mersenne Twister, http://www.math.sci.hiroshima-u.ac.jp/∼m-mat/MT/emt.html
[16]Portable, Extensible Toolkit for Scientific computation (PETSc), http://www.mcs.anl.gov/petsc
[17]Automatically Tuned Linear Algebra Software (ATLAS), http://math-atlas.sourceforge.net

of the levels of abstraction within the code, i.e., limiting or completely eliminating the use macro definitions, aliases and function tunnels, i.e., a function call with a function that is in turn, called by another function, etc. Although this may not seem like a very important consideration, it derives from the author's disdain for unnecessary levels of abstraction that sacrifice these virtues in the name of 'clever' syntax. It is also founded upon the notion that codes are generally modified, extended and upgraded on a more or less continual basis, and often not by the original developer. Minimising the level of abstraction during the original development process produces code that is both intuitive and readable. Not only does this help with day-to-day maintenance and the possible implementation of subsequent optimisations, but it also improves the chances that the application will be adopted by subsequent developers, preventing the application from fading into obscurity once the original author ceases active development. For everyday rough and ready programs, this is not a key issue, however, for large scale software projects that are aimed at an user base that extends beyond the original developers, readability and maintainability should be a cornerstone of the design and development process.

As a quick example of how seemingly harmless abstraction can be hazardous, consider the following scenario. The '**extern**' keyword tells the compiler that the variable in question is defined outside the current scope, which may or may not even be in the same file, and although it plays a vital role in global functionality, it requires that the variable actually be defined elsewhere. If, for some software project, the keyword '**extern**' was aliased to the name '**GLOBAL**', subsequent developers could misinterpret the abstraction and assume that the keyword '**GLOBAL**' completely defines a global variable, and proceed to modify, replace, add and develop, without declaring these '**GLOBAL**' variables explicitly. This would lead to hard to debug, faulty, and messy code that would produce unpredictable results, prolonging development.

## 5.2 Code structure, versions & variants

As previously mentioned, the new OpenDDA framework has been custom-built from scratch in the C language. At present, the only external dependencies are FFTW 3 [178], which provides the DFT functionality for the highly optimised DFT kernel that calculates the matrix-vector products required

by the iterative solution schemes, and for the parallel implementations, the relevant parallel APIs.

During the development process the new framework was constructed in an extremely modular fashion allowing any requisite alterations, and of course, any future development and/or modifications, to be performed in an almost 'plug-and-play' fashion. Furthermore, to maximise maintainability and readability, all files, functions, variables and arrays have been given human-readable pertinent names, e.g., '*control.input.template*', '*read_parameters.c*', '*dcomplex_bicgstab_MPI.c*', '*size_parameter*', '*wavenumber*', '*zero_padded_vector*', etc. On top of the intuitive naming conventions, to further aid the maintainability and readability of the new framework, data has been grouped by association into conveniently named structures. For example, to store the data pertaining to the iterative solution process, a specific globally visible structure, entitled '*iterative*', was constructed to hold all the relevant data, see Excerpt 5.1. This allows the constituent parameters to be set, stored and accessed via an extremely developer-friendly interface, i.e., '*iterative.maximum=10000*', '*iterative.scheme=bicgstab*', etc. A whole host of similar structures exist for other control data groups, including target description, output file names, timing information, effective radii, incident wavelengths, target rotations, incident polarisation, refractive indices, etc.

Code Excerpt 5.1: Example of the grouped structure-based storage of associated control data employed within the OpenDDA framework

```
/* Iterative control type definition */
#ifndef _OPENDDA_ITERATIVE_
#define _OPENDDA_ITERATIVE_
typedef struct {
/* Type definition for a structure to hold the control information for
   the iterative solution scheme */

   int maximum; /* Maximum number of iterations */
   int precond; /* Preconditioning flag */
   int initial_guess; /* Initial guess flag */
   int vec; /* Number of starting vectors */
   double tolerance; /* Convergence tolerance */
   double breakdown; /* Breakdown tolerance */
   char scheme[15]; /* Iterative solution scheme */
} attributes_iterative;
#endif
attributes_iterative iterative={0};
```

In terms of the variable scope, within the OpenDDA framework, it is simple to ascertain the scope of a specific variable or array as the declaration of all global data is localised in the main header files '*opendda_S.h*', '*opendda_OMP.h*' & '*opendda_MPI.h*', for the serial, OpenMP-based shared-memory and MPI-based distributed-memory implementations, respectively. To facilitate this global nature, the corresponding

external declarations of these variables is provided in a specific header file, entitled '*definitions.h*', which is included in all relevant source files.

During the development process, a fundamental concept that shaped the custom framework was that, in order to maximise performance and flexibility, the new framework should, by the inclusion of serial, shared-memory and distributed functionality, be able to take proper advantage of the various different architectures that exist. However, it was decided early on that these components should be manifested in separate architecture specific implementations of the new framework. The reasoning behind this decision is that an all-in-one distribution would be less maintainable and the associated development process would be much more complicated. Due to the added complexity that would be inherently required to incorporate the requisite support for the disparate architectures, including thread safety for the shared-memory implementation and parallel memory allocation, domain decomposition & interprocess communication for the distributed-memory implementation, the new framework would end up being a veritable rat's nest of '*if*', '*else*', '*and*' & '*or*' statements. This would essentially cloud the logical progression through the development process, producing a framework that would probably be suboptimal and would be much more prone to bugs and harder to debug. Furthermore, by creating separate versions, architecture specific support can be tailored into the framework and fine-tuned in such a way that, for a conjoint distribution, due to incompatibilities in the data layout, storage and component scalability, would not be conducive with computational efficiency and performance across all platforms.

In terms of the support for the different platforms, as is probably quite evident by now, there are at present, three different architecture specific versions of the new framework. **OpenDDA_S** is the base serial implementation, **OpenDDA_OMP** is the OpenMP-based shared-memory implementation and **OpenDDA_MPI** is the MPI-based distributed-memory implementation. Furthermore, each of these versions has three associated precision specific variants, i.e., single, double and long double. Please note that although the framework exists in all three precisions, the single-precision version was principally created for accessing the relative performance of the new framework, allowing direct and meaningful performance comparison with that of the DDSCAT code; which only exists in single precision.

## 5.3 Program control, inputs & outputs

A pivotal component of the new framework and something to which quite a lot of attention was given was the design and development of an intuitive, user-friendly and, in as much as is possible, foolproof method for parameter specification. This involved the design and creation of two distinct but intimately related components; the templates for the parameter input file(s) and a routine to parse the input file(s).

The template for the main input file, entitled '*control.input.template*', has been designed to make the specification of the various parameters as simple as possible. All a prospective user has to do is to copy this template file to '*control.input*' and alter the control parameters as necessary. As a quick aside, it is worth noting that during the compilation of any of the variants of the OpenDDA framework, to prevent the loss of any pertinent data, a backup of the previous control file is made, if one exists, '*control.input*'→'*control.input.copy*'. As will be seen below, to allow for intuitive and straightforward manipulation of the constituent control parameters, the layout of the control file is such that each control parameter has a specific human-readable descriptive string associated with it. The format is of the form: **descriptive string='user-defined value'**. The fundamental caveat in relation to the modification of the control file is that these descriptive strings, up to and including the '=' symbol, which the parser relies on to extract the relevant data, *must not* be altered in any way. Only the actual parameter data between the single quotation marks may be altered. For editing the file, GVIM, the GUI version of VIM text editor, is the recommended choice as it enhances the readability by employing different colours for the comments, descriptive strings and parameter values by default. Apart from the restrictions associated with these descriptive strings, the parser is extremely forgiving. During the development process, the parser was painstakingly constructed so as to possess quite a bit of leeway in the requisite format. It automatically ignores all spaces, tabs and punctuation except '+', '−', '.' & ',' and permits complex numbers to be supplied in a variety of formats, '$\mathbf{a}+\mathbf{bi}$', '$\mathbf{a}+\mathbf{bI}$', '$\mathbf{a}+\mathbf{b}$', '$\mathbf{a},\mathbf{bi}$', '$\mathbf{a},\mathbf{bI}$' & '$\mathbf{a},\mathbf{b}$'.

Due to the importance of the control file and the fundamental role that it and the constituent control parameters play in the OpenDDA framework as a whole, it is worth explaining the input format for each parameter and any associated issues, i.e., for many of the control parameters, rather than specifying a range within the

control file, OpenDDA allows the provision of a data file instead. Each description will be accompanied by the associated extract from the control file template, '*control.input.template*', itself. The first entry in the control file is for the incident wavelength.

```
# ******************************************************************************
# Incident wavelengths=minimum,increment,maximum - [micrometres]
Wavelengths='3.175,0.0,3.175'
#
```

The wavelengths are given in micrometres and the range is specified by giving the minimum, the increment between adjacent values and the maximum. A single wavelength value, as here, can be specified by setting the minimum and maximum to the same value and setting the increment to zero. If the file '*INPUT_wavelength.input*' exists, the wavelength values are read in from that file to an automatically allocated array and these values are ignored. The only restriction placed on the data in this file is that it must be positive and monotonically increasing. A template file, '*INPUT_wavelength.input.template*', is contained in the '*input_templates*' sub-directory. The second entry in the control file is for the specification of the incident polarisation via the normalised Jones vector [211]; an idea borrowed from the mainstay that is DDSCAT [5].

**Normalised Jones vector:** $\mathbf{E} = \begin{bmatrix} E_x \\ E_y \end{bmatrix}$, $|\mathbf{E}| = 1$

**Linear x-polarised:** $E_x = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$ **Linear y-polarised:** $E_y = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$

**Linear polarised at $+45^\circ$ [no phase difference]:** $E_{+45} = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 \\ 1 \end{bmatrix}$

**Linear polarised at $-45^\circ$ [$\pi$ phase difference]:**

$$E_{-45} = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 \\ -1 \end{bmatrix} \equiv \frac{1}{\sqrt{2}} \begin{bmatrix} -1 \\ 1 \end{bmatrix} \qquad \begin{array}{l} \text{(y leading x by } \pi) \equiv \text{(x leading y by } \pi) \equiv \\ \text{(y lagging x by } \pi) \equiv \text{(y lagging x by } \pi) \end{array}$$

**Right-circularly and left-circularly polarised [$\frac{\pi}{2}$ phase difference]:**

$$E_{rcp} = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 \\ -i \end{bmatrix} \equiv \frac{1}{\sqrt{2}} \begin{bmatrix} i \\ 1 \end{bmatrix} \qquad \left(\text{y leading x by } \frac{\pi}{2}\right) \equiv \left(\text{x lagging y by } \frac{\pi}{2}\right)$$

$$E_{lcp} = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 \\ i \end{bmatrix} \equiv \frac{1}{\sqrt{2}} \begin{bmatrix} -i \\ 1 \end{bmatrix} \qquad \left(\text{y lagging x by } \frac{\pi}{2}\right) \equiv \left(\text{x leading y by } \frac{\pi}{2}\right)$$

```
# ******************************************************************************
# Incident polarisation - described via the normalised Jones vector [E_{x},E_{y},0]
#
# e_{0}=[x_{real}+x_{imag}i,y_{real}+y_{imag}i,0.0+0.0i],yes/no
# yes/no: whether to include calculations for the orthonormal polarisation state
#
Incident polarisation e0='[1.0+0.0i,0.0+0.0i,0.0+0.0i],no'
#
```

When specifying the polarisation state, the normalisation factor is not required as normalisation is performed automatically by the parser. The '*yes/no*' option specifies whether or not to include calculations for the orthonormal polarisation state, which if required, is automatically generated by the parser. The next entry is the so-called effective radius which is the radius of a sphere of equal volume.

```
# ******************************************************************************
# Effective radii=minimum,increment,maximum - [micrometres]
Effective radii='0.5,0.0,0.5'
#
```

As with the incident wavelength, the values are given in micrometres and the range is specified by giving the minimum, the increment between adjacent values and the maximum. A single effective radius, can be specified by setting the minimum and maximum to the same value and setting the increment to zero. If the file '*INPUT_target_effective_radius.input*' exists, the radius values are read in from that file to an automatically allocated array and these values are ignored. The only restriction placed on the data in this file is that it must be positive and monotonically increasing. A template file, '*INPUT_target_effective_radius.input.template*', is contained in the '*input_templates*' sub-directory. Next up is the definition of the target shape.

```
# ******************************************************************************
# Target shape and construction parameters where [K,J,P] are the integer
# [x,y,z]-dimensions of the dipole array
#
# Shapes: (a) ellipsoid:          Shape='ellipsoid,K,J,P,NULL,NULL,NULL'
# Shapes: (b) cuboid:             Shape='cuboid,K,J,P,NULL,NULL,NULL'
# Shapes: (c) custom from file: Shape='valid_target_file_name,K,J,P,NULL,NULL,NULL'
Shape='ellipsoid,10,10,10,NULL,NULL,NULL'
#
```

At present, the only target options that are available within the OpenDDA framework are an ellipsoidal target, a cuboidal or rectangular target and the ability to read in a custom target from a user-supplied file. Ideally, the available options would encompass a broad representative and physically useful selection that would include spheres, ellipsoids, cuboids, cylinders, tetrahedra, triangular & hexagonal prisms, tetrahedra and multilayered slabs, with options for possible anisotropy or inhomogeneity, and perhaps even some form of a user-friendly flexible graphical interface for specifying irregular layouts. Unfortunately, the creation of this functionality was beyond the temporal contraints associated with the project. In terms of the feasibility of the extension of the target generation algorithms, the more rudimentary geometries should not pose any significant problems, however, in terms of a GUI for custom target specification, the development of such an interface is definitely not a

trivial concern and its viability would require further investigation. The ellipsoidal and rectangular targets are specified by the keywords '*ellipsoid*' and '*cuboid*', respectively, and must be accompanied by the associated dimensions of the computational domain, $K$, $J$ & $P$. Unlike the data files for the incident wavelengths and the effective radii, there is no specific filename associated with the custom target functionality. The user simply supplies an arbitrary filename, the associated $K$, $J$ & $P$ dimensions and the complex refractive index information, which will be discussed next, in the '*control.input*' file. The custom target file must contain six comma-separated values for each occupied site in the computational domain, $x, y, z, c_x, c_y, c_z$, where, $0 \leq x < K$, $0 \leq y < J$ & $0 \leq z < P$, are the coordinates of the occupied sites and, $c_x$, $c_y$ & $c_z$, are the associated material flags indicating the associated complex refractive indices, $0 \ldots (number_of_materials - 1)$. N.B. Note that the associated dimensions $K$, $J$ & $P$, the number of materials and the actual complex refractive index values *must* still be specified in the '*control.input*' file. A template file, '*INPUT_target_custom.input.template*', is contained in the '*input_templates*' sub-directory. The complex refractive index information is specified by giving the number of different materials and a value for each one.

```
# ******************************************************************************
# The number of dielectric materials for the target [MAX=unlimited] and the corresponding
# complex refractive indices, m=n+ki, for the target,0..(number_of_dielectric_materials-1)
#
Number of dielectric materials for the target='1'
Material 0='1.63631+0.372i'
Material 1='NULL'
...
Material 15='NULL'
#
```

Please note that there is no limitation on the number of distinct materials that can be employed within the OpenDDA framework. Furthermore, rather than specifying the actual specific refractive index value for a material, the OpenDDA framework allows the provision of a user-defined table from which, if not known explicitly, the refractive index value corresponding to the current incident wavelength can be interpolated, see §2.3. If a valid file exists for a particular material, $0 \ldots (number_of_materials - 1)$, the corresponding entry in the '*control.input*' file is ignored and the data is read in to an automatically allocated array for subsequent interpolation. To facilitate interpolation of the data, the data in this file must be such that the wavelength data, which is given in micrometres, is positive and monotonically increasing, and furthermore, the wavelength & refractive index values must be comma-separated. As discussed at the start of this section, all irrelevant punctuation is ignored and

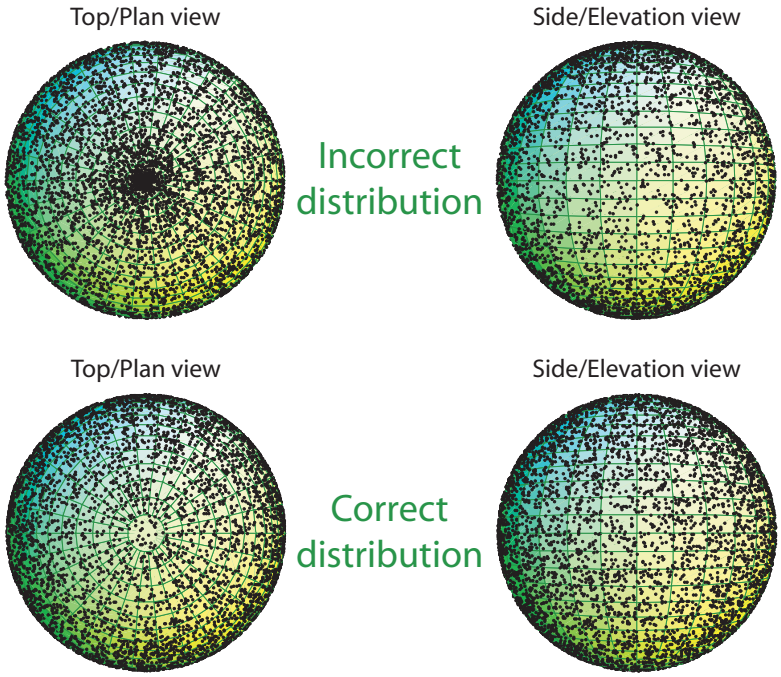



Figure 5.1: A plot of the distribution of 12000 points on a unit sphere[18]. The top section shows the incorrect distribution obtained by distributing the spherical coordinates evenly in $\phi \in [0, 2\pi)$ & $\theta \in [0, \pi]$, i.e. bunching near the poles, whereas the bottom section shows the correct even distribution that is by distributing evenly in $\cos(\theta) \in [1, -1]$.

the complex values can be specified in a variety of formats. The filename *must* be '*INPUT_refractive_index_vs_wavelength_material_'#'.input*, where '#' is the material number to which the file relates. As with other parameters, a relevant template file, '*INPUT_refractive_index_vs_wavelength_material_0.input.template*', has been included in the '*input_templates*' sub-directory. The next entry is the specification of the target orientations via the three Euler angles, $\phi$, $\theta$ & $\psi$, see §1.3.

```
# ******************************************************************************
# Target orientations(phi,theta,psi)=minimum,increment,maximum
#
Euler phi='0.0,30.0,330.0'
Euler theta='-1.0,0.2,1.0'
Euler psi='0.0,30.0,330.0'
#
```

As with the incident wavelength and effective radius parameters, the ranges are specified by giving the minimum, the increment between adjacent values and the maximum. Similarly, a single angle can be specified by setting the minimum and maximum to the same value and setting the increment to zero. As with previous param-

eters, if a relevant user-provided input file exists, where the naming conventions are '*INPUT_target_orientation_euler_phi.input*', '*INPUT_target_orientation_euler_psi.input*' & '*INPUT_target_orientation_euler_theta.input*', for $\phi$, $\theta$ & $\psi$, respectively, then the ranges are ignored and the data for the angles in question are read in from those files to automatically allocated arrays. As usual, templates for these input files are in the '*input_templates*' sub-directory. The only restrictions placed on the data for the angles $\phi$ & $\psi$ is that the values should be in the range $[0, 2\pi)$ and should be positive and monotonically increasing. However, $\theta$, which should only be monotonically increasing, is a little bit different. Whereas, the two azimuthal Euler angles define longitudes, i.e., slices of equal area, the polar or zenith Euler angle $\theta$ defines latitudes. If these are distributed evenly in the range $[0, \pi]$, then the latitudes are equally spaced but the resulting area elements are not of equal size; getting smaller towards the poles. With foresight to future orientational averaging support, as with DDSCAT [5] and ADDA [188], for integration purposes, since the integration element is $d\Omega = \sin(\theta)d\phi d\theta d\psi = d\phi d\cos(\theta)d\psi$, by distributing $\theta$ evenly in $\cos(\theta) \in [1, -1]$, area elements are of equal size and the total area in $(\cos(\theta), \phi)$ is $4\pi$. To see this explicitly, Fig. 5.1 has been included which shows the distribution of 12000 points on a unit sphere. As shown in the top half, by choosing a uniform distribution of the spherical coordinates in $\phi \in [0, 2\pi)$ & $\theta \in [0, \pi]$, since the area gets smaller toward the poles, points end up bunched near the poles, whereas in the bottom half, by distributing evenly in $\cos(\theta) \in [1, -1]$, a uniform distribution is obtained.

Due to the associated time restraints and the fact that the remit of the project was solely to build a new highly-optimised computational infrastructure, only token post-processing, exclusively for operational verification of the new framework, is performed. However, although the associated post-processing may not be in place, the fundamental infrastructural support for most of the post-processing has been incorporated. One such example is the user-defined scattering directions, i.e., a set of directions in which scattering results are requested.

```
# ******************************************************************************
# Scattering directions using the spherical coordinates phi and theta
# (phi,theta)=minimum,increment,maximum
#
Scattering angle phi='0.0,30.0,330.0'
Scattering angle theta='0.0,30.0,90.0'
#
```

[18] Adapted from the "Sphere Point Picking" entry from MathWorld - A Wolfram Web Resource, Weisstein, E. W., http://mathworld.wolfram.com/SpherePointPicking.html

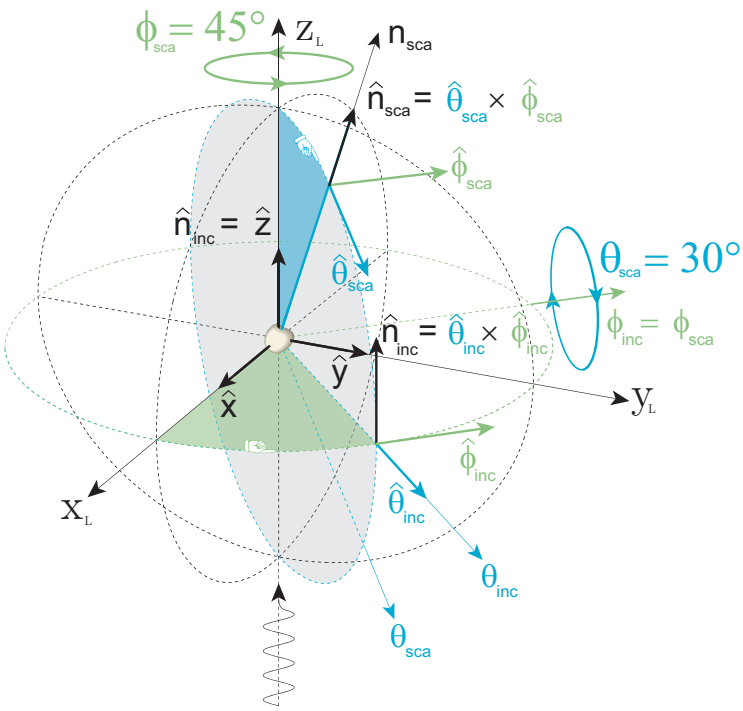



Figure 5.2: A schematic of the scattering conventions for specification of the scattering directions using the two spherical coordinates angles, the azimuth angle $\phi_{sca} \in [0, 2\pi)$ about $+\hat{\mathbf{z}}$ and the polar/zenith angle $\theta_{sca} \in [0, \pi]$ measured from $+\hat{\mathbf{z}}$. $\theta$ is the angle between the direction of propagation of the incident wave, $\hat{\mathbf{n}}_{inc} = +\hat{\mathbf{z}}$ and the scattering direction $\hat{\mathbf{n}}_{sca}$. $\phi$ is the angle between the $xz$-plane and the scattering plane, i.e., the plane passing through both $\hat{\mathbf{n}}_{inc}$ and $\hat{\mathbf{n}}_{sca}$.

The scattering directions are specified using the spherical coordinates $\phi_{sca}$ & $\theta_{sca}$. Conceptually, these angles are identical to the Euler angles $\phi$ & $\theta$. $\phi_{sca}$ is an azimuthal angle, i.e., a clockwise angle $[0, 2\pi)$ in the $xy$-plane about the $+\hat{\mathbf{z}}$ axis looking in the $+\hat{\mathbf{z}}$ direction, and defines longitudes, and $\theta_{sca}$ is a polar or zenith angle, i.e., the clockwise angle $[0, \pi]$ in the $xz$-plane about the $+\hat{\mathbf{y}}$ axis looking in the $+\hat{\mathbf{y}}$ direction, and defines latitudes. $\theta_{sca}$ is the angle between the direction of propagation of the incident wave $\hat{\mathbf{n}}_{inc} = +\hat{\mathbf{z}}$ and the scattering direction $\hat{\mathbf{n}}_{sca}$. $\phi_{sca}$ is the angle between the $xz$-plane and the scattering plane, i.e., the plane passing through both $\hat{\mathbf{n}}_{inc}$ and $\hat{\mathbf{n}}_{sca}$, see Fig. 5.2. Again, the ranges are specified by giving the minimum, the increment between adjacent values & the maximum, and a single angle can be specified by setting the minimum and maximum to the same value and setting the increment to zero. Also, if either of the user-defined input files '*INPUT_scattering_direction_phi.input*' or '*INPUT_scattering_direction_theta.input*' exists, where as usual, templates are included

in the '*input_templates*' sub-directory, the given ranges are ignored and the data, which should be positive and monotonically increasing, is read in from that file to an automatically allocated array. This marks the end of the control parameters associated with the physical description of the system. The rest of the control parameters are related to the computational aspects of the framework. The first is the choice of iterative solution scheme and the associated control parameters.

```
# ******************************************************************************
# Iterative scheme and number of starting vectors
# (1) bicg,NULL            BiConjugate-Gradients
# (2) bicg_sym,NULL        BiConjugate-Gradients for symmetric systems
# (3) bicgstab,NULL        Stabilised version of the BiConjugate-Gradients
# (4) cg,NULL              Conjugate-Gradients
# (5) cgs,NULL             Conjugate-Gradients Squared
# (6) mlbicgstab_orig,X    BiCGSTAB variant based on multiple Lanczos starting vectors
#                          (Author's original algorithm)
# (7) mlbicgstab,X         BiCGSTAB variant based on multiple Lanczos starting vectors
#                          (Author's reformulation)
# (8) mlbicgstab_ss,X      BiCGSTAB variant based on multiple Lanczos starting vectors
#                          (Author's space saving algorithm)
# (9) qmr,NULL             Quasi-minimal residual with coupled two-term recurrences
# (10) qmr_sym,NULL        Quasi-minimal residual for symmetric systems
# (11) rbicgstab,X         Restarted, stabilised version of the BiConjugate-Gradients
# (12) tfqmr,NULL          Transpose-free quasi-minimal residual
```

```
Iterative scheme='bicgstab,NULL'
# Iterative scheme - Convergence tolerance
Convergence tolerance='1e-10'
# Iterative scheme - Breakdown tolerance
Breakdown tolerance='1e-50'
# Iterative scheme - Maximum number of iterations
Maximum number of iterations='10000'
# Iterative scheme - Initial guess
Initial guess='0'
# Iterative scheme - Preconditioning
Preconditioning='1'
#
```

This simply involves setting the iterative scheme and, if relevant, the number of starting vectors, the stopping criteria, including the maximum allowable number of iterations & the convergence tolerance, i.e., the value that the relative norm must be smaller than, the breakdown tolerance, which protects against division by zero, the initial guess for the polarisations and whether or not to employ Point-Jacobi preconditioning.

A useful feature that has been incorporated into the OpenDDA framework is the creation of a basic output file that contains the spatial description of the target in so-called VTK[19] format. The Visualization ToolKit is an open source and freely

[19]The Visualization ToolKit (VTK), http://www.vtk.org

available software system for 3D computer graphics, image processing, and visualisation. This output file allows the simulation target to be viewed dynamically in 3D. Although the VTK interface and syntax are quite involved and have a rather steep learning curve, front-end GUIs such as Paraview[20] and Mayavi[21] provide an extremely user-friendly point-and-click interface that hides the associated complexity of the VTK language and rendering engine. Paraview, which is the author's choice, is an open-source, multi-platform application developed as a collaborative effort between Kitware (the developers of VTK), Sandia National Laboratories, CSimSoft, Los Alamos National Laboratory, Army Research Laboratory and others. It is freely available and is capable of visualising extremely large data sets. By default, this option is disabled as, depending on the size of the simulation, the output files can be quite large and can take a while to write to disk.

```
# ******************************************************************************************
# VTK data output. Print out the target xyz coordinate data to a vtk file for 3D
# visualisation of the target using a GUI like Paraview
VTK target xyz coordinate data='0'
#
```

An important feature in the new framework is the associated timing functionality.

```
# ******************************************************************************************
# Timing. Output the overall program runtime and timings for the iterative kernel & the
# individual components of the DFT engine to the file OUPUT_program_execution.info
Timing='1'
#
```

By default, timing is left enabled as the associated overheads are negligible. The last entry that is global to all versions and variants of the OpenDDA framework is the output precision. OpenDDA allows users to explicitly set the output precision. Users can either choose the number of significant digits for all outputs or the parameter can be set equal to zero, in which case, OpenDDA automatically uses the maximum number of significant digits for the specific precision for the current compute environment, i.e., the header file '*float.h*' includes the constants 'FLT_DIG', 'DBL_DIG' & 'LDBL_DIG'. These define the maximum precision for the float, double and long double datatypes, respectively.

```
# ******************************************************************************************
# Number of significant digits for the program output. If this is set to '0' then the code uses the
# predefined value DBL_DIG which is the number of significant digits for the double data type
Output precision [DOUBLE]='[0]'
#
```

[20]Paraview, http://www.paraview.org
[21]Mayavi, http://mayavi.sourceforge.net

Apart from these parameters which are common to all versions and variants of the OpenDDA framework, each of the parallel implementations has a specific computational control parameter associated with it. For the OpenMP-based shared-memory implementation, the user is able to specify the number of threads to use for the execution of parallel regions within the framework.

```
# ******************************************************************************
# Number of OpenMP threads. The number of threads for the parallel regions. If set to 0
# here then the number of threads available is determined automatically by OpenMP.
OpenMP threads='4'
#
```

However, if the control parameter is set equal to zero, then the framework, via the OpenMP library routines, automatically determines the number of available threads. Although, at first glance, this may seem unnecessary as, performance-wise, it may seem illogical not to take advantage of every available thread, the reason for this functionality is that parallel environments are often subject to queue restrictions in which only a fraction of the available threads are allocated to any one user. Thus, an internal limit must be set to prevent the framework from attempting to commandeer threads from outside this allocation. If the user-supplied value turns out to be invalid, i.e., less that zero or greater than the maximum number of available threads, OpenDDA defaults to the automatic determination of the number of threads available.

For the MPI-based distributed-memory implementation, the number of processes to use is specified as a command-line argument to the MPI execution script, however, the distributed-transpose algorithm that supports the parallel DFT kernel must be selected, see §4.3.8 for a discussion of the possible options.

```
# ******************************************************************************
# Distributed-transpose algorithm.
Distributed-transpose algorithm='[0,0]'
#
```

Although it would probably never be done in practise, the user is allowed to specify different algorithms for the tensor and vector components, i.e., one for both the forward transpose of the six independent tensor components and another for the forward transpose of the three components of the zero-padded vector and the reverse transpose of the resultant of the FD element-wise multiplication of the tensor & vector components. The first entry specifies the transpose algorithm for the tensor components and the second entry relates to the vector components.

As a final remark, it is worth very briefly explaining the various output files that the OpenDDA framework produces.

- '*OUTPUT_incident_wave.info*' provides information about the incident wave including the wavelengths that were used, incident polarisation state, and whether or not the calculation for the orthonormal polarisation state was included.
- '*OUTPUT_iterative.info*' provides information about the setup and performance of the iterative solution process, including a summary of the user-defined control parameters and a detailed log of the convergence behaviour.
- '*OUTPUT_program_execution.info*' provides the timing information for the framework, including the overall runtime, the time taken by the iterative scheme to reach convergence, the number of calls that were made to the DFT kernel and a detailed breakdown of the performance of each component of the DFT kernel.
- '*OUTPUT_target_description.info*' provides a detailed summary of the simulated target, including the shape, the dimensions of the dipole array, the number of sites in the computational domain, the number of occupied sites in the representation of the target, the number of different materials in the target, the effective radius, size parameter, dipole spacing and the number of dipoles per wavelength.
- '*OUTPUT_refractive_index_vs_incident_wavelength_'#'.info*', provides a list of the refractive index for each incident wavelength for material '#'.
- '*OUTPUT_target_orientations.info*' & '*OUTPUT_scattering_angles.info*' provide a list of the target orientations and scattering angles that were used for the simulation.

## 5.4 Memory estimation & performance timing

One of the most crucial aspects of high-performance computing is the pre-execution knowledge of the associated memory requirements. With the aim of achieving the best possible results, the ability to estimate the memory that a particular problem will require allows users to tailor the attributes of the problem to the available resources, and thus, get the most out of the application.

In terms of the allocated memory that a particular program will require, there are two flavours to be considered; that of stack and heap allocation, see Excerpt 5.2 for an illustrative example. The stack is an area of memory that is used for data whose size can be determined when the program is compiled. It is used for storing variables that will not grow in size and will not be moved around in memory, i.e., static data objects. It is a '*LIFO: Last In, First Out*' based orderly structure in which memory management is performed automatically, i.e., it is set up at compile time and memory

is released as soon as it goes out of scope. It is extremely efficient due to the fact that it requires no system calls and the process of enlarging or shrinking the stack simply involves adding to or subtracting from the memory address in the 'stack pointer', i.e., a pointer to the most recent item placed on the stack; the current topmost item on the stack.

On the other hand, the heap is an area of memory that is reserved for memory that is allocated at runtime. It is used for storing variables that are likely to grow in size and may move around in memory, i.e., dynamic data objects. It is a disorderly structure in which memory management is the responsibility of the programmer, i.e., memory blocks can be requested in any size and remain allocated until they are deallocated explicitly. Memory that is allocated but not subsequently deallocated is referred to as a memory leak, i.e., unintentional memory consumption by a computer program which can diminish the performance by reducing the amount of available memory. Although heap allocation provides greater control and flexibility, it requires explicit intervention from the programmer/operating system and has additional associated overheads as the allocation and deallocation involves system calls.

Code Excerpt 5.2: Example demonstrating the declaration of 'static' variables on the stack and 'dynamic' variables on the heap.

```
1  void function()
   {
3     int a; /* a is on the stack */
      int array[15]; /* 15 integers on the stack */
5     size_t size = BUFSIZE; /* size is on the stack */

7     int *ptr;
      ptr=(int*)malloc(size); /* BUFSIZE integers allocated on the heap
9                                BUT the actual ptr is on the stack */
      ...
11
      free(ptr);ptr=NULL;
13 }
```

Each of the OpenDDA variants is accompanied by a dedicated function, '*guessti-mate*', that estimates the memory requirements for a given parameter set, chosen precision, single, double, or long double, and architecture, serial, OpenMP, or MPI. Each is an independent executable, separate from the main OpenDDA program and only takes account of non-negligible dynamic arrays, i.e., the heap allocation; the memory estimation routines ignore the relatively insignificant contribution from the stack allocation. Please note that this estimation of the approximate memory requirements is *pre-execution*. No real time monitoring of the stack and heap allocation is performed during runtime within the OpenDDA framework.

The functions are completely interactive and parse all inputs to ensure validity. They return the estimated memory requirements for all the iterative schemes. For the

MPI version, the function also returns the maximum memory required per node, for each scheme. All variants read in the dimensions of the dipole array, the target shape, (Note: at present only ellipsoid, cuboid and custom target from file are supported), whether or not to use preconditioning and the number of starting vectors for the relevant iterative schemes. On top of these basic parameters, the OpenMP estimator also prompts for the number of threads that will be active. This allows the estimation to include the contribution from the extra scratch spaces that are required to ensure thread safety.

For the MPI version, apart from the additional parameter to specify the distributed-transpose algorithm to be used, the only other difference is that the estimator must be run within an MPI environment. This is extremely important for the distributed environment, where each autonomous node has only a limited amount of local memory, as it allows proper estimation of the memory requirements per node. The fact that it must be run with the '*mpirun*' or '*mpiexec*' execution script should not be considered as a reason for concern and does not require that the estimator be actually queued on a large-scale distributed system. This is a crucial design issue. The MPI estimator is meant to be a dynamic and interactive tool to optimise the parameters of a given problem for a specific distributed environment. It would be nonsensical to have the situation where the MPI estimator itself, would have to be submitted to a batch queue and executed on the distributed system. This would negate the dynamic and interactive nature of the estimator and furthermore, depending on the state of the queue, for a task with negligible runtime, may take an extended period of time to return results. If the memory estimations are not satisfactory or within the specific thresholds imposed by the computational resources, the estimation parameters would have to be altered and the job resubmitted. However, MPI will just as easily spawn the required processes on a local host. To compound this fact and show a sample estimation, the following shows the execution of the MPI estimator for a sphere based on a $K = J = P = 400$ computational domain, decomposed across 64 nodes, using distributed-transpose algorithm 0, preconditioning and 3 starting vectors for the relevant iterative schemes. Although the estimation is for a distributed system, it is run using the command '*mpiexec -n 64 guesstimate_MPI*' on just a single serial Pentium 4 host.

```
james@abydos:~/double/opendda_MPI_double/source>
Please enter K [the integer x dimension of the dipole array]
400
Please enter J [the integer y dimension of the dipole array]
400
Please enter P [the integer z dimension of the dipole array]
400
Please select the required shape using the appropriate integer
.    0. ellipsoid
.    1. cuboid
.    2. custom from file
0
Transpose algorithm? 0..5
0
Preconditioning enabled OR disabled? 0 OR 1
1
Number of starting vectors for the iterative schemes:
RBiCGSTAB & ML(n)BiCGSTAB*? 1..50
3

. OOOOOOOOOOOOOOOOOOOOOOOOOOOOO
. O   Maximum allocated memory per node   O
. OOOOOOOOOOOOOOOOOOOOOOOOOOOOO

Number of occupied sites in the 400x400x400 array = 33511376

  Iterative Scheme          Total (GB)    Maximum per node (GB)

  BiCG                      32.889GB      0.542GB
  BiCG_sym                  28.395GB      0.471GB
  BiCGSTAB                  32.889GB      0.542GB
  CG                        28.395GB      0.471GB
  CGS                       32.889GB      0.542GB
  ML(3)BiCGSTAB_orig        52.364GB      0.850GB
  ML(3)BiCGSTAB             52.364GB      0.850GB
  ML(3)BiCGSTAB_ss          43.376GB      0.708GB
  QMR                       35.885GB      0.590GB
  QMR_sym                   32.889GB      0.542GB
  RBiCGSTAB(3)              38.881GB      0.637GB
  TFQMR                     35.885GB      0.590GB
```

In terms of the performance timing, for both the serial and OpenMP implementations of OpenDDA, microsecond accurate timing functionality is provided by the '*gettimeofday*' function as defined in '*sys/time.h*'. Although one of the internal structures associated with this function is now considered obsolete, this portability issue is easily circumvented, either by passing 'NULL' in its stead or by simply hard-coding the structure definition into the timing file. For the MPI versions of OpenDDA, the timing is provided by the in-built MPI function '*MPI_Wtime*'.

Although performance timing and benchmarking play a much more crucial role in the development and optimisation process, due to the negligible overheads incurred, the execution times are still calculated by default. The times are stored in the file 'OUTPUT_program_execution.info' and include the overall runtime, the runtime of the iterative scheme and a thorough breakdown of the performance of the DFT matrix-vector kernel.

## 5.5 Compiling & running OpenDDA

In order to compile and run the OpenDDA framework, the first task is to satisfy any and all external dependencies. The main prerequisite and the only external package required by all variants of the new framework, is the FFTW package, version 3; the FFTW 2 interface is not supported and as pointed out by the authors, the interface for FFTW 3 is not backwards-compatible [178].

The FFTW source code is freely available from the website and is trivial to install, see below. The default compile is in double-precision, however, this can be altered simply by using *--enable-float* or *--enable-long-double*. Once the library has been installed, each precision specific variant of OpenDDA automatically links to the relevant library using the appropriate linker flag, i.e., *-lfftw3* for double precision, *-lfftw3f* for single precision or *-lfftw3l* for long double precision.

- **INSTALLING FFTW 3**
- ./configure --prefix='*install path*' **CC**='*C compiler*' **CFLAGS**='*C flags*' **F77**='*Fortran 77 compiler*' **FFLAGS**='*Fortran 77 flags*' **CXX**='*C++ compiler*' **CXXFLAGS**='*C++ flags*'
- make && make install
  - **DOUBLE-PRECISION**: default
  - **SINGLE-PRECISION**: add *--enable-float* to the configure line above
  - **LONG DOUBLE-PRECISION**: add *--enable-long-double* to the configure line above

Apart from FFTW 3, the only other prerequisites are the parallel APIs required by the parallel implementations of the OpenDDA framework. For the OpenMP-based shared-memory implementation, an OpenMP-compliant C compiler that supports the various library routines and compiler directives is essential. There are many such compilers available, of which a by no means exhaustive list includes, the Intel Compilers[22], the PathScale Compilers[23], the Portland Group PGI Compilers[24], the Sun Studio[25], and even, as of version 4.2, the GNU Compiler Collection[26] (GCC).

For the MPI-based distributed-memory implementation, OpenDDA requires an MPI API. This can be any of the freely available & portable implementations of MPI, including MPICH1 [208, 2], MPICH2 [3] or Open MPI [4]. However, MPICH1 is the

[22]Intel Compilers, http://www.intel.com
[23]PathScale Compilers, http://www.pathscale.com
[24]Portland Group PGI Compilers, http://www.pgroup.com
[25]Sun Studio, http://developers.sun.com/sunstudio
[26]GCC, the GNU Compiler Collection, http://gcc.gnu.org

recommended choice as it the only one that, in its default configuration, allows the use of the most efficient distributed-transpose algorithms that support the DFT kernel, i.e., algorithms **0** & **3**, see §4.3.8. Without explicitly tinkering with the thresholds of the internal message passing protocols, see §4.3.4, the other implementations can be used but are restricted to using distributed-transpose algorithms **2** & **5**, see §4.3.8.

Once these prerequisites have been satisfied, the chosen variant of the OpenDDA framework can be compiled by setting the command for the C compiler using the tag **CC**, the desired compiler flags using the tag **CFLAGS** and the header and library paths in the relevant Makefile. For the serial and OpenMP-based shared-memory implementations, setting the header and library paths just involves specifying the include & library paths for FFTW 3 using the tags, **INCFFTW** & **LIBFFTW**, respectively, whereas for the MPI-based distributed-memory implementations, the include & library for the MPI API must also be set using the tags, **INCMPI** & **LIBMPI**, respectively.

As an aside, it is worth mentioned that, as discussed in §2.5, due to the lack of native complex support in the ANSI C89 standard, for guaranteed portability, the OpenDDA framework incorporates custom-built complex functionality. This means that each complex mathematical function is provided by a call to a small external function, i.e., '*dcomplex dcomplex_add(dcomplex in0,dcomplex in1)*'. With this in mind, it is recommended that '*interprocedural optimisation*' be employed when compiling OpenDDA. This enables so-called '*function inlining*' or '*inline function expansion*' which involves the compiler integrating functions that are frequently executed into their callers; actually replacing the function call instructions with the function code itself. These days compilers have got to the stage where they will carry out these optimisations between multiple files, performing '*inline function expansion*' for calls to functions defined in separate files. This '*interprocedural optimisation*' can be enabled via the appropriate compiler flags, i.e., for the Intel and PathScale compilers, just add '*-ipo*' and '*-ipa*' to **CFLAGS**, respectively.

The serial, OpenMP-based and MPI-based variants of the OpenDDA framework can then be compiled using the commands '**make clean&&make opendda_S**', '**make clean&&make opendda_OMP**' and '**make clean&&make opendda_MPI**', respectively. With the relevant variant compiled, all that is left to do is to copy the template control file '*control.input.template*' to the filename '*control.input*' and edit the contained input parameters as appropriate, see §5.3. The OpenDDA framework can then be executed using either '**./opendda_S**', '**./opendda_OMP**' and '**mpirun -np**

**# opendda_MPI**' or '**mpiexec -n # opendda_MPI**', as the case may be, where '#' is the number of processors, or, if available, the MPI variants can be submitted to a batch queue using an appropriate PBS script. For example, the following shows a sample PBS submission script.

**SAMPLE PBS JOB SUBMISSION SCRIPT**

```
#PBS -N job_name
#PBS -A user_account
## Tell pbs to use XX nodes and YY processors per node
## Note: Cluster uses nodes=XX:ppn=YY but shared memory
## machine uses ncpus=XX to specify resources
#PBS -l nodes=64:ppn=2
## Maximum time required in hh:mm:ss
#PBS -l walltime=36:00:00
## Files to contain standard error and standard output
#PBS -e job_name.err
#PBS -o job_name.out
## Mail user only if job aborts (use -m ae if you want mail
## notification of job completion)
#PBS -m ae
#PBS -M email_address
### Queue name
#PBS -q queue_name
## Declare job non-rerunable
#PBS -r n
## Have PBS inherit your current environment
#PBS -V

# Set the working directory
WORKDIR=$PBS_O_WORKDIR
## handy tip: $PBS_O_WORKDIR is set by PBS to the directory you
## submitted the job from
#cd $PBS_O_WORKDIR
cd $WORKDIR
echo Working directory is $WORKDIR

echo Time is: `date`

# Run the executable
mpiexec -n 64 -pernode ./opendda_MPI;
```

*"We are at the very beginning of time for the human race. It is not unreasonable that we grapple with problems. But there are tens of thousands of years in the future. Our responsibility is to do what we can, learn what we can, improve the solutions, and pass them on."*

**Richard Feynman**

*"The most exciting phrase to hear in science, the one that heralds the most discoveries, is not 'Eureka!' but 'That's funny...' "*

**Isaac Asimov**

# 6 Benchmarking & Analysis

Please note that due to time restraints and in an attempt to standardise the results, all benchmarks and comparisons were run using the same set of essentially arbitrarily chosen physical parameters.

- **Shape**: SPHERE
- **Incident wavelength**: $\lambda = 3.175\,\mu\text{m}$
- **Effective radius**: $a = 0.5\,\mu\text{m}$
- **Incident polarisation**: $\hat{\text{e}}_0 = \hat{\text{x}}$
- **Complex refractive index**: $m = 1.63631 + 0.372i$[27]
- **Target orientation**: $\phi = 25^\circ, \theta = 55^\circ, \psi = 80^\circ$

## 6.1 Benchmarking architectures

Several architectures were employed to test the performance and ascertain the scalability of the new framework. The benchmarking process involved four separate systems, two shared-memory and two distributed-memory. The former allowed both the serial and OpenMP versions to be put through their paces, whereas the latter provided the necessary benchmarks for the distributed MPI-based implementation.

Since the OpenDDA framework has been custom-built from scratch and due to the sheer number of benchmarks that had to be performed in order to cover all

[27] Complex refractive index of ice for $\lambda = 3.175\,\mu\text{m}$ from the REFICE routine, originally developed by Warren in 1984 [176], and subsequently improved by Warren, Gao and Wiscombe in 1995 (Unpublished, source code available in the PyARTS package, a python package developed to compliment the Atmospheric Radiative Transfer Simulator - ARTS. Available from the Satellite Atmospheric Science Group, http://www.sat.ltu.se/arts/tools).

aspects of the new components of this framework, to simplify the description of the relevant architecture on which a specific benchmark took place, the following list details both the particular hardware and software configurations of each system. For each benchmark, the relevant architecture will be referenced simply by the associated tag, NEMO, FRANKLIN, HOBBES or WALTON.

- **SHARED-MEMORY SYSTEMS**
  - **NEMO:** *nemo.scg.nuigalway.ie*
    - SYSTEM: HP xw9300 Workstation
    - OS: openSUSE 10.2, x86_64
    - CPU: 2×AMD Dual-Core Opteron 275, 2.2 GHz, 2 MB(2 × 1 MB) cache
    - RAM: 16 GB
    - COMPILERS: Intel Compilers 9.1, icc 9.1.045, icpc 9.1.045, ifort 9.1.040
    - SOFTWARE: FFTW 3.1.2, MPICH1 1.2.7p1, MPICH2 1.0.5p4
  - **FRANKLIN:** *franklin.scg.nuigalway.ie*
    - SYSTEM: SGI Altix 3700
    - OS: Red Hat Enterprise Linux AS release 3, IA-64
    - CPU: 24×Intel Itanium 2, 1.5 GHz, 6 MB cache
    - RAM: 96 GB
    - COMPILERS: Intel Compilers 9.1, icc 9.1.045, icpc 9.1.045, ifort 9.1.040
    - SOFTWARE: FFTW 3.1.2, Intel MKL 8.1.014
- **DISTRIBUTED-MEMORY SYSTEMS**
  - **HOBBES:** *hobbes.scg.nuigalway.ie*
    - SYSTEM: Dell Poweredge 1750
    - OS: Red Hat Linux release 9, x86
    - CPU: 32 compute nodes, 64×Intel Xeon, 2.4 GHz, 512 KB cache
    - RAM: 2 GB per node
    - COMPILERS: Intel Compilers 9.1, icc 9.1.045, icpc 9.1.045, ifort 9.1.040
    - SOFTWARE: FFTW 3.1.2
  - **WALTON:** *walton.ichec.ie*
    - SYSTEM: IBM cluster 1350
    - OS: SUSE LINUX Enterprise Server 9, x86_64
    - CPU: 476 compute nodes, 952×AMD Opteron Single-Core Opteron 250, 2.4 GHz, 1 MB cache
    - RAM: 412 have 4 GB per node, 64 have 8 GB per node
    - COMPILERS: QLogic PathScale Compiler Suite, Version 3.0
    - SOFTWARE: FFTW 3.1.2, MPICH1 1.2.7, MPICH2 1.0.2

NEMO is the author's desktop workstation on which OpenDDA was developed, FRANKLIN and HOBBES are housed in the basement of the I.T. Department in NUI, Galway, and WALTON, which is located in an IBM hosting facility in the outskirts of Dublin, is the ICHEC[28] distributed cluster. The author also had access to

[28]Irish Centre for High-End Computing, http://www.ichec.ie

ICHEC's large-scale Bull Novascale 6320 shared-memory system, known as HAMILTON, *hamilton.ichec.ie*, which boasts a $\frac{1}{4}$ TB of physical memory and 32 1.5GHz Itanium 2 processors with 4MB caches. However, it was decided to exclude this system as, due to high demand, it was not possible to get sole access; a requirement that is imperative for shared-memory benchmarks to avoid corruption from memory contention issues with other jobs. Furthermore, for optimal throughput, HAMILTON employs daemons that dynamically interfere with the execution of the job to ensure that it is on the most efficient cpu set, and if as other jobs finish, a more efficient configuration becomes available, the job is shifted to that set. Although this is sensible from an administrative point of view, from a benchmarking perspective, it is by no means ideal and negates the possibility of meaningful and trustworthy results.

## 6.2 DFT kernel performance

The following two subsections deal with the performance of just the new highly-optimised custom-built DFT matrix-vector kernel itself. §6.2.1 discusses the performance of the serial implementation described in §3.6 and §6.2.2 looks at the performance and scalability of both the shared-memory OpenMP and distributed-memory MPI implementations of the kernel that were detailed in §4.2 and §4.3, respectively.

### 6.2.1 Serial performance

To put the prescribed optimisations from §3.6 in perspective, Table 6.1 & 6.2 tabulate the improvements quantitatively for the case where $K = J = P = 255$, and consequently, $K' = J' = P' = 509$ & $K'' = J'' = P'' = 512$. These values have not been chosen for any specific reason, bar the fact that they produce a linear system which is towards the high-end of applicability for the original algorithm, even for medium-sized supercomputers. The tabulated results are merely indicative, with equivalent improvements achieved across the entire range of the algorithms. In comparison to the original algorithm, the new algorithm requires over 62%, 75% and 37% less transforms in the $\hat{\mathbf{x}}$, $\hat{\mathbf{y}}$ and $\hat{\mathbf{z}}$ directions, respectively, which equates to a reduction of almost 60% in the total number of transforms that need to be performed in order to carry out the matrix-vector product, see Table 6.1. Furthermore, the memory required to store both the six independent tensor components and three

| | COMPUTATION | | | |
|---|---|---|---|---|
| | **Original No. of Transforms** | **New No. of Transforms** | **Difference Actual** | **Difference %** |
| Y0→Y5:$\hat{\mathbf{x}}$ | $6J''P''$(**1,572,864**) | $6JP$(**390,150**) | **1,182,714** | **75.20%** |
| X0→X2:$\hat{\mathbf{x}}$ | $3J''P''$(**786,432**) | $3J''P$(**391,680**) | **394,752** | **50.20%** |
| Q0→Q2:$\hat{\mathbf{x}}$ | $3J''P''$(**786,432**) | $3J''P$(**391,680**) | **394,752** | **50.20%** |
| **Total : $\hat{\mathbf{x}}$** | **3,145,728** | **1,173,510** | **1,972,218** | **62.70%** |
| Y0→Y5:$\hat{\mathbf{y}}$ | $6K''P''$(**1,572,864**) | $6(K+kf)P$(**394,740**) | **1,178,124** | **74.90%** |
| X0→X2:$\hat{\mathbf{y}}$ | $3K''P''$(**786,432**) | $3KP$(**195,075**) | **591,357** | **75.20%** |
| Q0→Q2:$\hat{\mathbf{y}}$ | $3K''P''$(**786,432**) | $3KP$(**195,075**) | **591,357** | **75.20%** |
| **Total : $\hat{\mathbf{y}}$** | **3,145,728** | **784,890** | **2,360,838** | **75.05%** |
| Y0→Y5:$\hat{\mathbf{z}}$ | $6K''J''$(**1,572,864**) | $6(K+kf)(J+jf)$(**399,384**) | **1,173,480** | **74.61%** |
| X0→X2:$\hat{\mathbf{z}}$ | $3K''J''$(**786,432**) | $3K''J''$(**786,432**) | 0 | **0%** |
| Q0→Q2:$\hat{\mathbf{z}}$ | $3K''J''$(**786,432**) | $3K''J''$(**786,432**) | 0 | **0%** |
| **Total : $\hat{\mathbf{z}}$** | **3,145,728** | **1,972,248** | **1,173,480** | **37.30%** |

Table 6.1: Comparison of the number of 1D transforms that are required in order to carry out the matrix-vector product for the original and new algorithms.

zero-padded vector components has been reduced significantly. The fact that the T-to-c conversion of the six tensor components was never explicitly performed means that the storage for the associated arrays has been reduced by over 87%. For the three vector components, the use of an external $xz$-scratch plane provides a reduction of over 75%. Together, these optimisations reduced the memory required to store the tensor and vector components by over 83%, see Table 6.2.

To affirm the benefits ascribed to the new serial framework, several corroborative benchmarks were performed on FRANKLIN. In order to exemplify the various improvements detailed in §3.6, the advancements have been applied sequentially to show their respective contribution to the cumulative effect. The results are the average of three separate runs. Figs. 6.1→6.2 encompass the disparate ranges over which the algorithms can be applied. Fig. 6.1 shows the original, various hybrid and new timings in a regime that allows comparison with the explicit matrix-vector multiplication and Fig. 6.2 shows long-range timings, where the algorithms have been applied, within their aptitude, to much larger systems. The following list describes the interpretation of the legend entries.

- **3D (Orig):** Original algorithm where the 'out-of-the-box' 3D DFT is used to calculate all forward and inverse transforms.
- **1D (Orig):** Original algorithm performed using 1D DFTS.
- **1D (T New):** Hybrid system, where the tensor components are treated with the new algorithm and the vector components and iDFT are as in **1D (Orig)**.

| | STORAGE | | | |
|---|---|---|---|---|
| | **Original Storage** | **New Storage** | **Difference Actual** | **Difference %** |
| $^{[P']}[^{J']}[^{K']}_{\text{c03D}} \rightarrow$ $^{[P']}[^{J']}[^{K']}_{\text{c53D}}$ | $6K''J''P''$ | $6(K+kf)(J+jf)(P+pf)$ $+K''+J''+P''$ | | |
| | **805,306,368** | **103,042,608** | **702,263,760** | **87.21%** |
| Single | **6GB** | **0.77GB** | **5.23GB** | **87.21%** |
| Double | **12GB** | **1.54GB** | **10.46GB** | **87.21%** |
| $^{[N']}_{\text{x03D}} \rightarrow {}^{[N']}_{\text{x23D}}$ | $3K''J''P''$ | $3KJ''P$ | | |
| | **402,653,184** | **99,878,400** | **302,774,784** | **75.20%** |
| Single | **3GB** | **0.74GB** | **2.26GB** | **75.20%** |
| Double | **6GB** | **1.49GB** | **4.51GB** | **75.20%** |
| **Total** | **1,207,959,552** | **202,921,008** | **1,005,038,544** | **83.20%** |
| Single | **9GB** | **1.51GB** | **7.49GB** | **83.20%** |
| Double | **18GB** | **3.03GB** | **14.97GB** | **83.20%** |

Table 6.2: Comparison of the complex number storage for the kernel of the matrix-vector product for the original and new algorithms.

- **1D (T & V New):** Next level of hybrid system, where only the iDFT is performed as in '**1D DFT (Orig)**' and all forward DFTs are performed as prescribed by the new algorithm.
- **1D (T, V & I New):** Complete new algorithm where all DFTs & iDFTs are performed as prescribed by the new algorithm.
- **1D OMP (T, V & I New): 1D (T, V & I New)** using OpenMP over 4 threads.
- **Intel MKL CBLAS zgemv:** Explicit matrix-vector product using the Intel MKL[29] [179]. Note: Only possible for very small systems.

**N.B. In each plot, the label 'Tensor Matrix Order N' refers to a matrix of order N, (the number of sites in the full computational domain), whose elements are tensors, i.e., the matrix is actually of order 3N.**

Before discussing the results in these figures, it is worth a quick look at the ranges over which the various algorithms were applied and their associated memory requirements. Since the original, various hybrid and new algorithms have different memory requirements, the ranges over which they were applied also differ. Table 6.3 lists the maximum range values chosen for the various algorithms. For each stage of optimisation, the columns give the number of dipoles $N_d$ in the representation of the spherical target and the associated memory usage. The final column lists the

[29] Intel Math Kernel Library (MKL), http://developer.intel.com/software/products/mkl

| | Maximum K=J=P | SPHERE $N_d$ | Memory | Memory required for K=J=P=610 |
|---|---|---|---|---|
| **Intel MKL CBLAS zgemv** | 35 | 22,575 | 68.35GB | 1894.41PB |
| **3D & 1D (Orig)** | 310 | 15,599,512 | 33.98GB | 257.16GB |
| **1D (T New) & (T&V New)** | 510 | 69,460,944 | 66.42GB | 113.77GB |
| **1D & 1D OMP (T,V&I New)** | **610** | **118,851,616** | **51.97GB** | **51.97GB** |

Table 6.3: Comparison of the number of dipoles, $N_d$, in the representation of the spherical target and the associated memory usage for the chosen upper limits of the benchmarking range for the various algorithms. Note that the chosen upper limit for the full new algorithm is $K = J = P = 610$ and the final column shows the memory that would be needed to be able to use the size for the various algorithms.

memory that would be required by each of the various stages if it were to be applied to the maximum for the new algorithm, i.e., $K = J = P = 610$. The main point to note is that although the original 'out-of-the-box' 3D DFT algorithm would require over 257 GB for $K = J = P = 610$, the new algorithm and its OpenMP counterpart, only require 52 GB to perform the same calculation.

With regards to the actual benchmarking results themselves, for the short-range comparison in Fig. 6.1, the first noteworthy point is that even though the explicit matrix-vector multiplication was performed with the highly optimised Intel MKL [179], it is obvious that the prodigious memory requirements associated with the sheer size of the systems involved negate its use almost immediately.

Another important point is the for both the short-range and long-range cases, the timings for the **3D (Orig)** and **1D (Orig)** algorithms are essentially identical. Since the optimisation of the kernel is predicated on the concept of removing superfluous transforms by performing the requisite 3D DFTs as ensembles of 1D transforms, this congruency is crucial as it shows that there is no overhead associated with the migration.

Furthermore, for both plots, the sequential application of the components of the new algorithm is clearly discernible. Each of the three constituent optimisations, i.e., the optimisation of the DFT of the six independent tensor components, the DFT of the three zero-padded vector components and the subsequent iDFT of the resultant of the element-wise FD multiplication of the tensor and vector components, provides an improvement, which culminates with the aforementioned 60% reduction in the number of requisite transforms having a comparably profound influence on the timings.

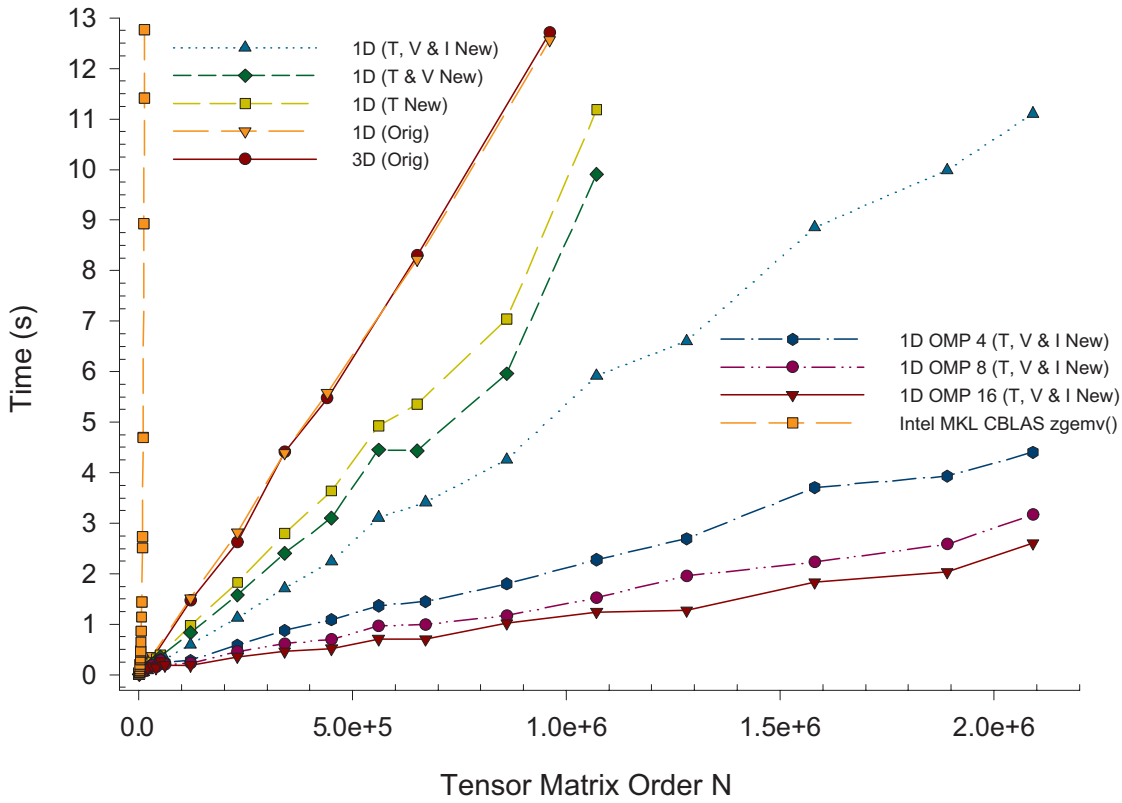



Figure 6.1: A short-range comparison of the different algorithms that can be applied to calculate the matrix-vector product to allow the inclusion of the explicit calculation using the CBLAS zgemv function from the Intel MKL [179].

In long-range case, the comparison of the original and new algorithms is limited to the nethermost region of the plot. This is due to the prohibitive memory requirements associated with the use of the original algorithm. Note that even for extremely large systems, the new algorithm is notably faster than the original algorithm is for a system that is just a fraction of the size.

As a final remark on these two plots, although the scalability and performance of the shared-memory implementation will be treated more formally in the next section, for direct comparison with the serial improvements, each plot includes timings for the OpenMP-based kernel across 4, 8 & 16 threads, respectively. These timings unequivocally show the substantial performance gains that can be obtained from shared-memory parallelisation using the OpenMP API. To compound this fact, Fig. 6.3 displays the corresponding speedup achieved. As one would expect, the scalability of the algorithm improves as the system size is increased. This makes sense as to make

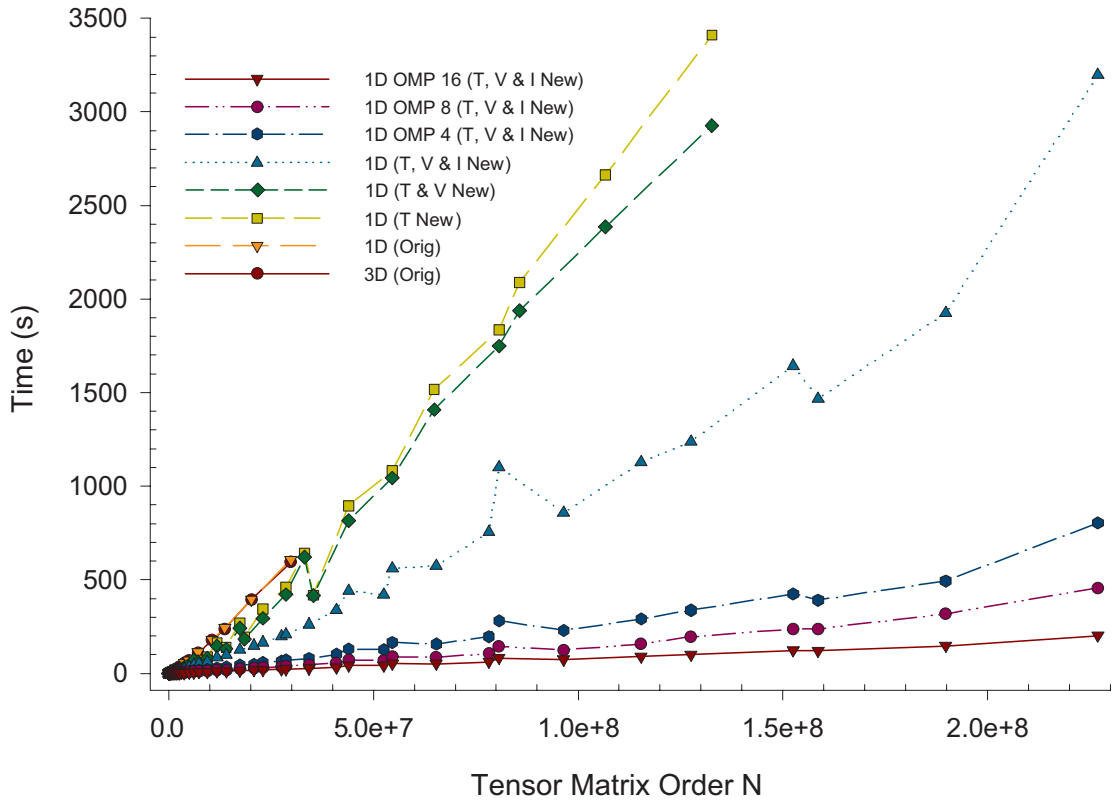



Figure 6.2: A long-range comparison of the different algorithms that can be applied to calculate the matrix-vector product to illustrate the high end behaviour of the algorithm and to show that the new algorithm can be applied to systems long after the original algorithm has become impracticable.

the parallelisation beneficial, there must be enough work for each thread to ensure that the associated overheads become negligible. Across 4 threads, the speedup is essentially ideal across most of the range, however, as the number of threads goes up, the work delegated to each thread becomes less and less and there is a noticeable drop-off in the scalability towards the lower end of the range.

On a separate note, as mentioned previously, DFT plans must be created in order to implement the requisite transforms. The fact that the new algorithm only requires the creation of a limited number of 1D plans, rather than the full 3D plans required by the original algorithm, means that the associated initialisation becomes negligible. This is especially true for FFTW with the FFTW_MEASURE option as used by OpenDDA, which instructs FFTW to run and measure the execution time of several DFTs in order to find the best way to compute the transform. This can be

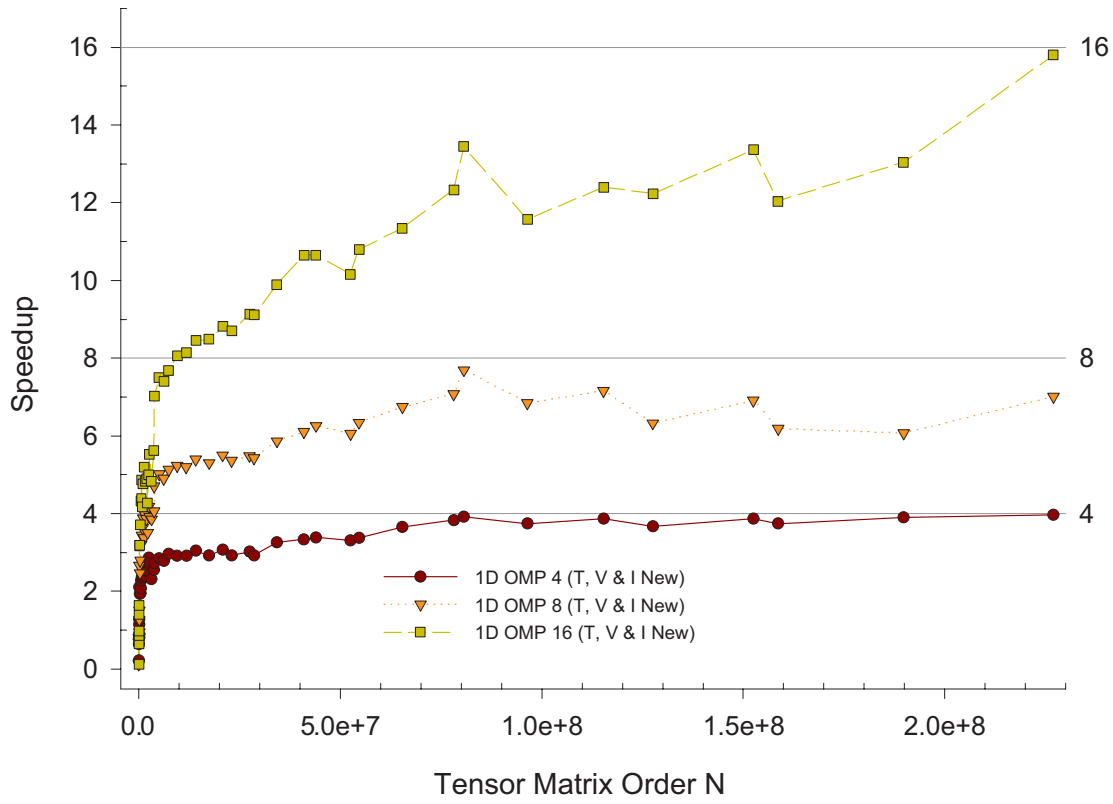



Figure 6.3: OpenMP speedup achieved across 4, 8 & 16 threads, respectively, for the long-range case shown in Fig. 6.2.

clearly seen by the unambiguous disparity in Fig. 6.4, which shows the timings for the FFTW plan creation using FFTW_MEASURE for the various algorithms, where for all intents and purposes, the initialisation times for the serial and OpenMP variants of the new algorithm are essentially flatline.

Finally, although the subject of the associated memory requirements has already been touched on in Table 6.2 & 6.3, Fig. 6.5 shows a log plot contrasting the memory required to perform the matrix-vector product using the various serial and parallel algorithms across the full benchmarking range. Apart from the serial and OpenMP variants, the figure also incorporates the corresponding memory requirements for the distributed MPI-based implementation, showing the decrease in memory per node that is directly proportional to the number of active nodes. The results speak for themselves, with the new algorithm requiring less than a fifth of the memory of the original algorithm. The serial decrease in the transition from the original to the new algorithm is augmented by the fact that, although, as mentioned at the end of §3.6,

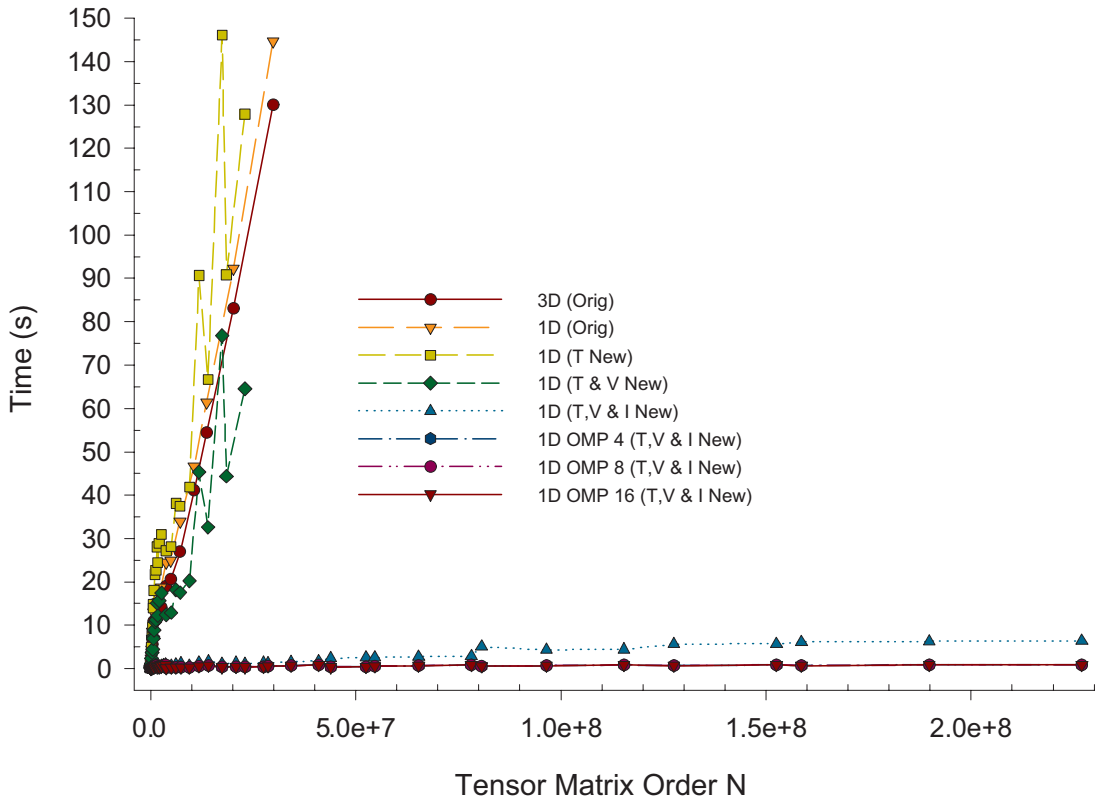



Figure 6.4: long-range comparison of the time taken to create the FFTW plan using FFTW_MEASURE for the different algorithms.

the MPI version requires more memory than the serial algorithm when run on a single node, the parallel memory allocation means that the memory requirement per node decreases linearly as the number of nodes is increased. This distributed memory model allows the benefits of the new algorithm to be capitalised upon whereby the reduction in memory per node is only limited by the number of nodes available.

### 6.2.2 Parallel performance

The very fact that, in order to enable the removal of the superfluous and counter productive components of the requisite 3D DFTs and iDFT, the 3D transforms have been reduced from a single conjoint operation to a collection of individual 1D transforms, means that parallel techniques can be employed to further enhance the performance and the range of applicability of the kernel. As was discussed in §4, there are two separate custom-built parallel variants, an OpenMP-

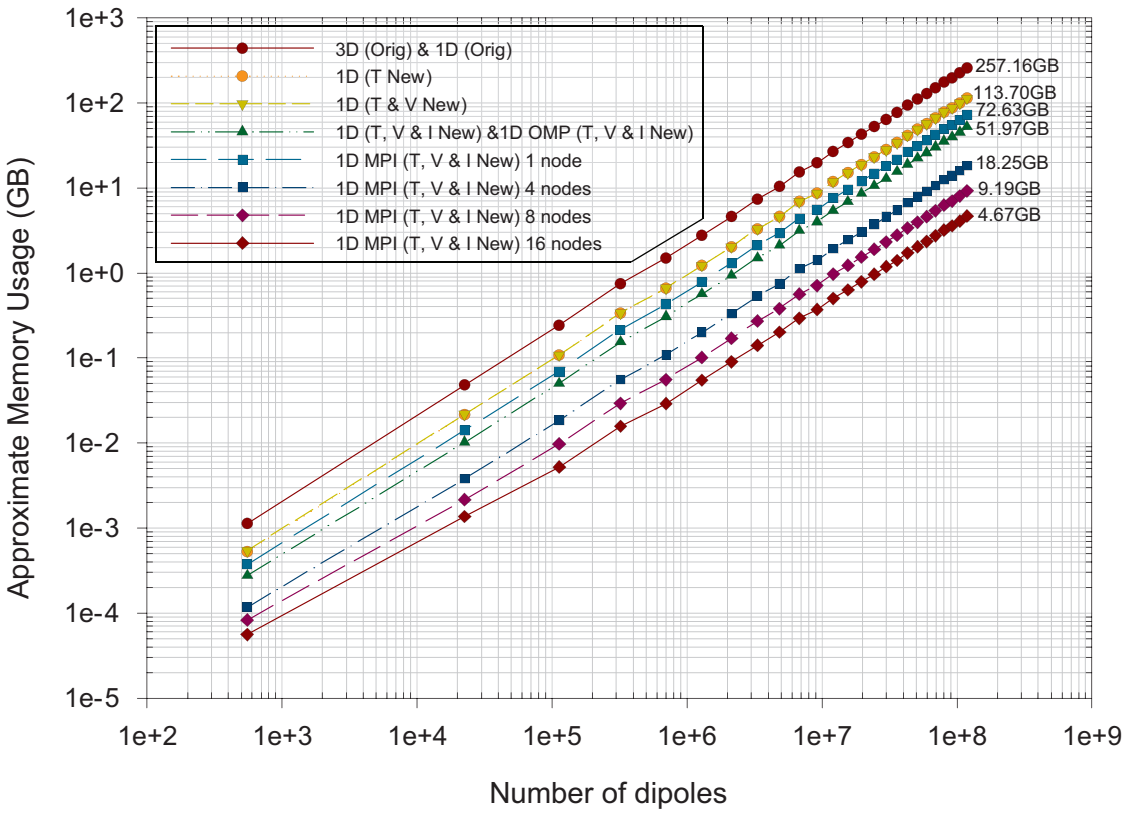



Figure 6.5: Approximate memory required to perform the matrix-vector product with double-complex accuracy.

based implementation designed for shared-memory architectures and a MPI-based implementation designed for distributed-memory systems.

To ascertain the performance and scalability of the OpenMP-based shared-memory implementation, a large-scale benchmark was performed across $1, 2, \ldots, 20$ threads on FRANKLIN, the results of which, are the average of three separate runs. The system size was chosen to be $K = J = P = 600$, which produced a computational domain with $N = 216 \times 10^6$ sites in total and a representation of the spherical target with $N_d = 113,098,912$ occupied sites. In terms of memory usage, this system required 48.897 GB for 1 thread with a linear increase of 0.064 GB per thread to 50.121 GB for 20 threads, where the extra memory is for the thread private scratch spaces that are required to ensure thread safety. Although this may sound like large amount of memory, in comparison, for the same system, the original 'out-of-the' box 3D transforms, which are limited to the serial environment, would require 241.855 GB. This is well in excess of $4\frac{1}{2}$ times the maximum that is required by the OpenMP implementation

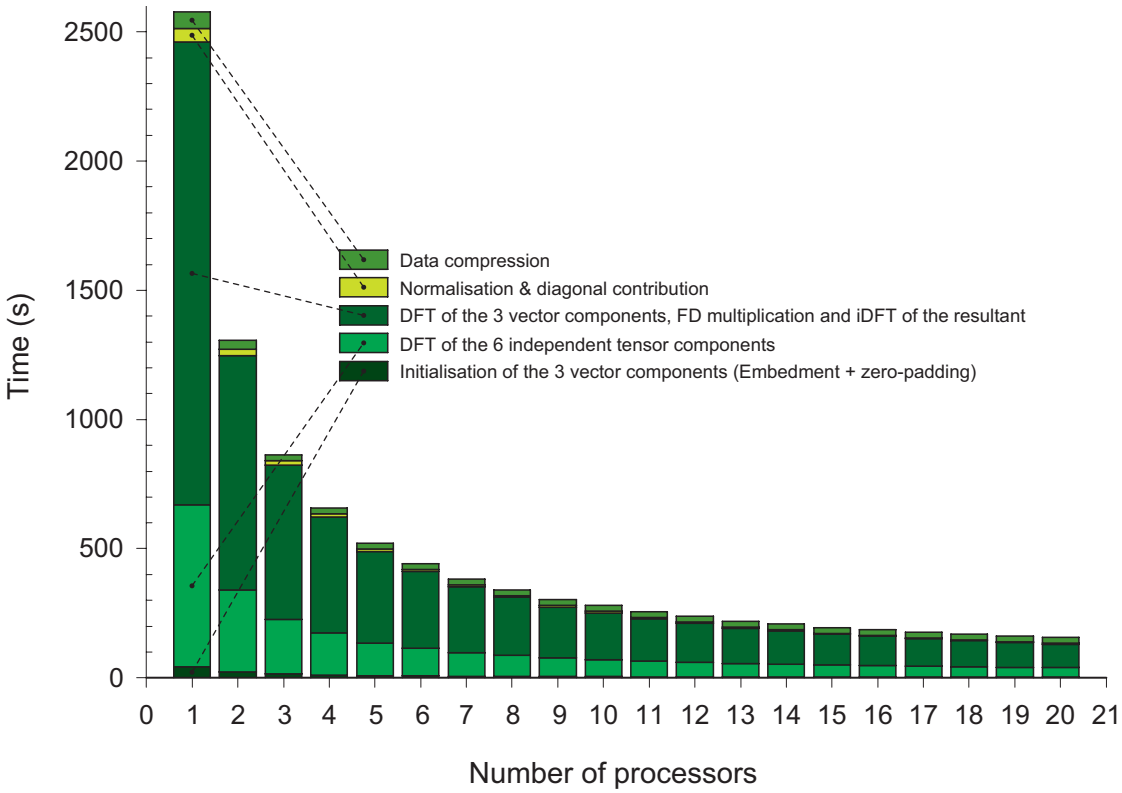



Figure 6.6: OpenMP scalability and timing run on FRANKLIN for a spherical target with $N_d = 113,089,912$ occupied sites. (Scalability view)

of the new algorithm and a factor of over $2\frac{1}{2}$ times the full 96 GB of total available physical memory on FRANKLIN.

Figs. 6.6 & 6.7 show the timing results for the benchmark and although they essentially display the same information, they have both been included, as in Fig. 6.6, it is much easier to see the cumulative scalability of the new algorithm as a whole, whereas for Fig. 6.7, the actual contribution of each internal component and its specific scalability are more discernible.

The first component is the time for the initialisation of the target vector. As mention in §2.6.2, for computational efficiency, OpenDDA only stores vector entries corresponding to occupied sites. However, for the requisite structure to enable DFTs to be employed to streamline the matrix-vector products required by the iterative schemes, the matrix-vector kernel must use the structure associated with the entire rectangular computational domain. Thus, the computationally compact vector arrays must be extended out during the embedding procedure. The second component

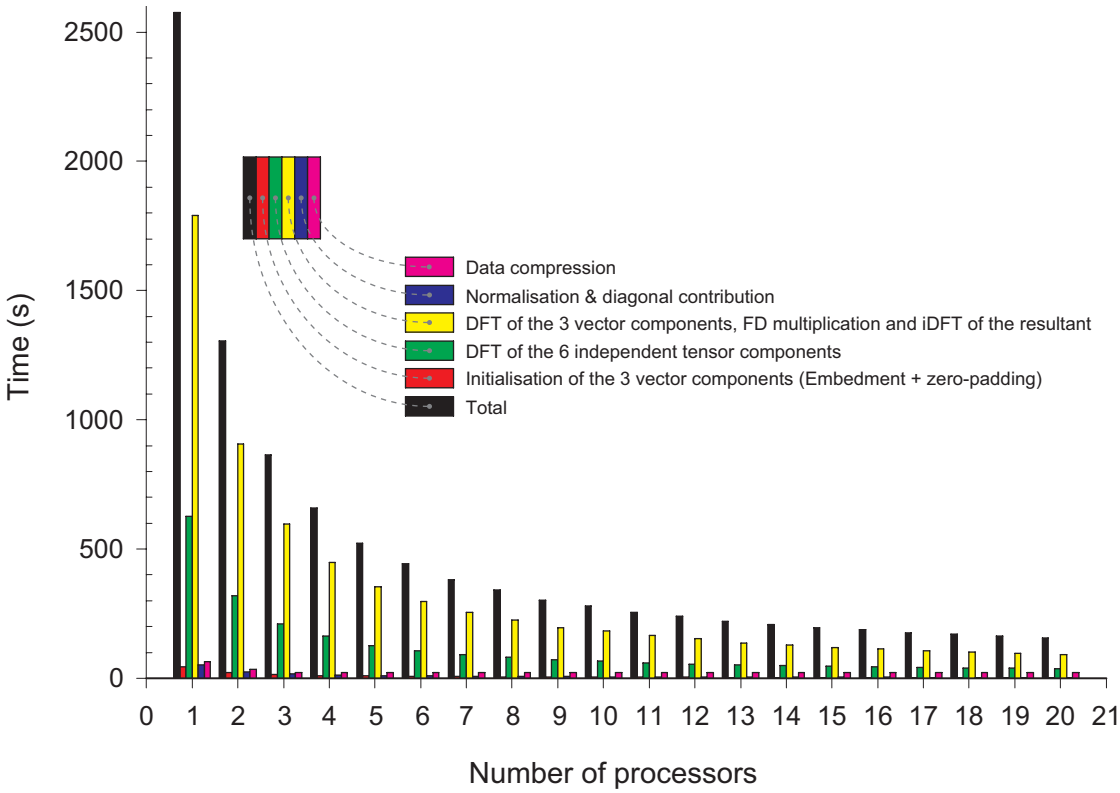



Figure 6.7: OpenMP scalability and timing run on FRANKLIN for a spherical target with $N_d = 113,089,912$ occupied sites. (Component view)

is just the time for DFT of the six independent tensor components. The third component accounts for the DFTs of the three zero-padded vector component arrays, the subsequent element-wise FD multiplication of the tensor & vector components and the iDFTs of the resultant, which as mentioned in §3.6, have been amalgamated into a conjoint operation for computational efficiency. This component constitutes the core of the DFT kernel, and as such, is the dominant computational burden, as reflected by the associated magnitude in both figures. The fourth component is comprised of the normalisation of the resultant and the addition of the previously excluded diagonal contribution that had to be ignored for DFT applicability. The final component is the data compression component which compresses the resultant back into the computationally efficient form with just the entries corresponding to occupied sites.

The actual timings results are self-explanatory, with a reduction from 2575.95s for 1 thread to just 156.05s using 20 threads. In terms of the associated speedup, Fig. 6.8

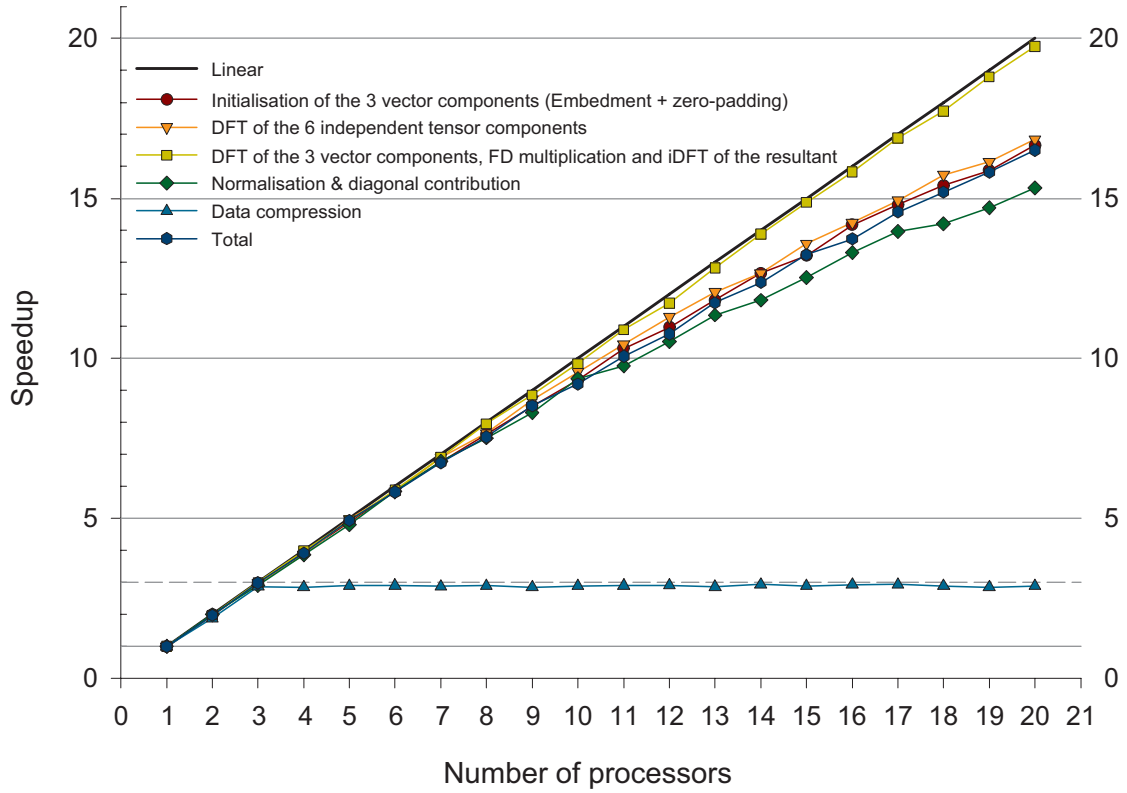



Figure 6.8: Speedups associated with the OpenMP scalability and timing run on FRANKLIN for a spherical target with $N_d = 113,089,912$ occupied sites.

displays the speedups for both the DFT kernel as a whole and for each of the five distinct constituent components separately. For the data compression component, the scalability can be seen to be limited to a factor of three. As was discussed in §2.6.2, this is because the data compression is performed in-place in the matrix-vector product result buffer. In the shared-memory environment, all active processors are working within a single address space and, to avoid the loss of pertinent data, the data compression algorithm is restricted to one thread for each of the three vector components. However, it can be clearly seen from either Fig. 6.6 or Fig. 6.7 that data compression only constitutes a minute fraction of the overall runtime, and as a result, this limitation has little or no effect on the overall scalability of the algorithm as a whole.

For OpenMP-based applications, the more work that there is to be subdivided, the less significant the associated OpenMP overheads become and the better the scalability of the component will be. Fig. 6.8 exemplifies this fact, as the core of the

kernel, i.e., the DFTs of the three zero-padded vector component arrays, the subsequent element-wise FD multiplication of the tensor & vector components and the iDFTs of the resultant, which constitutes, on average, between 60% and 70% of the entire runtime, has practically ideal scalability, whereas for the other smaller components, the scalability, although not far from ideal, does decrease slightly. Although this benchmark was performed under essentially ideal conditions, where the author had sole access to FRANKLIN for the duration of the benchmark, excluding any extraneous circumstances like severe memory contention issues with other large jobs on a particular system, for everyday use, the scalability should not be far removed from these values.

To test the performance and scalability of the MPI-based distributed-memory implementation, the same $K = J = P = 600$ system as for the OpenMP shared-memory benchmark, i.e., with $N = 216 \times 10^6$ sites in the computational domain, of which $N_d = 113,098,912$ are occupied in the spherical representation of the target, was run on WALTON. For this distributed case, the results are the average of 48 separate runs. The reason for so many separate runs is that, WALTON is the 952 processor main computational resource at the ICHEC[30] supercomputing facility and is always in high demand from the extremely large user base. Thus, unlike FRANKLIN, it was not sensible or feasible to try obtain sole access. Although, users are assigned distinct node sets, since distributed programs generically require interprocessor communication, all active users are sharing the same finite amount of network bandwidth, and as a result, there is still the performance issue of the dependency of the communication time on the network load. Furthermore, during the benchmarking process, WALTON was going through an awkward spate of what were termed 'retarded nodes', where processors would unexpectedly throttle down to well below their operating norm. Since the performance of the distributed implementation as a whole is reliant on each node doing its part, the benchmarks were often being corrupted by these rogue nodes.

Unlike the OpenMP-based shared-memory implementation which began with a single thread and was subsequently scaled up to 20 threads, the distributed benchmark was initialised across 32 nodes and was subsequently scaled up to 64 nodes. The reason for this is that WALTON is a distributed cluster, and as such, each of the nodes that were utilised for the benchmark has only 4 GB of physical memory. The shared-memory implementation required 48.897 GB for 1 thread, however, as

[30] Irish Centre for High-End Computing, http://www.ichec.ie

| Nodes | Total (GB) | Per node (GB) | Nodes | Total (GB) | Per node (GB) | Nodes | Total (GB) | Per node (GB) |
|---|---|---|---|---|---|---|---|---|
| 32 | 70.638 | **2.219** | 43 | 71.071 | **1.655** | 54 | 71.968 | **1.412** |
| 33 | 70.610 | **2.208** | 44 | 71.791 | **1.649** | 55 | 71.986 | **1.314** |
| 34 | 70.954 | **2.104** | 45 | 71.809 | **1.643** | 56 | 72.570 | **1.310** |
| 35 | 70.693 | **2.094** | 46 | 71.264 | **1.638** | 57 | 73.153 | **1.307** |
| 36 | 71.084 | **1.991** | 47 | 71.751 | **1.538** | 58 | 72.887 | **1.303** |
| 37 | 70.692 | **1.981** | 48 | 72.427 | **1.532** | 59 | 72.480 | **1.300** |
| 38 | 71.028 | **1.879** | 49 | 71.927 | **1.528** | 60 | 72.104 | **1.265** |
| 39 | 71.419 | **1.871** | 50 | 71.460 | **1.492** | 61 | 72.610 | **1.199** |
| 40 | 70.816 | **1.832** | 51 | 72.057 | **1.425** | 62 | 73.147 | **1.196** |
| 41 | 71.549 | **1.763** | 52 | 72.687 | **1.420** | 63 | 73.685 | **1.193** |
| 42 | 71.474 | **1.756** | 53 | 72.422 | **1.416** | 64 | 73.465 | **1.190** |

Table 6.4: Total memory required and associated maximum memory requirement per node for the performance and scalability benchmark of the main MPI-based distributed-memory implementation of the new algorithm.

mentioned previously, to facilitate the distributed FD multiplication of the tensor & vector components, the MPI-based implementation stores an extra section of the tensor components, see §4.3.2, and thus for 1 node, requires 68.241 GB. It is obvious that neither of these will fit on a single node on WALTON. However, due to the inherent parallel decomposition that is part of the new distributed algorithm, the maximum memory requirement per node decreases linearly as the number of nodes is increased. Thus, allowing for some extra scratch spaces for each of the additional nodes, across 32 nodes, the system requires 70.638GB in total, but *only* **2.219GB** per node. As the system is scaled from 32 to 64 nodes, again, apart from some extra scratch spaces for each of the additional nodes, the memory requirements fluctuate slightly due to the fact that the number of planes both before and after the distributed transposes has decreased. The extra scratch spaces increase the total memory requirement marginally, however, due to the parallel memory allocation, the maximum requirement per node drops steadily as the number of active nodes is increased, to just **1.19 GB** per node across 64 nodes, see Table 6.4.

The actual timing results and the associated speedups for this benchmark are shown in Fig. 6.9. Apart from the communication aspect, the only pivotal difference between this and the shared-memory implementation is that the framework infrastructure is such that all functions are now performed in parallel by autonomous nodes.

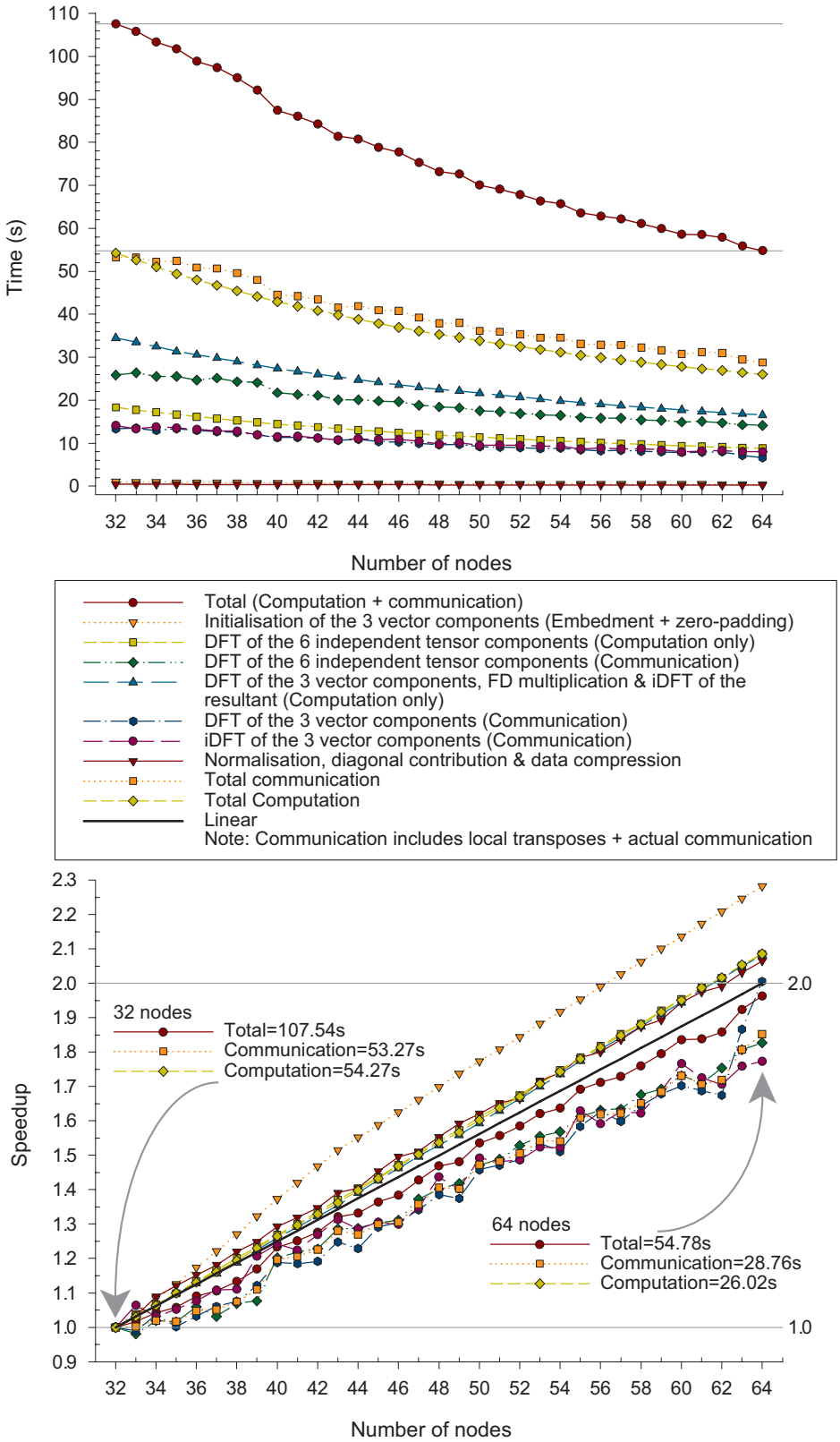



Figure 6.9: MPI scalability and timing run on WALTON with associated speedups for a spherical target with $N_d = 113,089,912$ occupied sites.

For the legend entries, there are a few small differences. First of all, the communication phases for the two forward DFTs and single iDFT have been included. Secondly, the normalisation of the resultant, the addition of the previously ignored diagonal contribution and the data compression are now lumped together. This is because, in the distributed-memory environment, by definition, there are no issues with memory contention, and thus, no possibility of pertinent data being unintentionally overwritten during the data compression process. Each node simply compacts the local vector data.

In terms of performance, the calculation took 107.54 s across 32 nodes; 53.27 s for communication and 54.27 s for computation. Across 64 nodes, the same calculation took 54.78 s; 28.76 s for communication and 26.02 s for computation. To put these in perspective, for the OpenMP-based shared implementation, the same system required 2575.95 s using 1 thread and 156.05 s using 20 threads. Incidentally, these distributed benchmarks were computed using distributed-transpose algorithm **5**, see §4.3.8. This is a plane-by-plane distributed transpose that transposes all components simultaneously and is based on the theory prescribed in §4.3.7. At the time, due to looming deadlines and associated time restraints, algorithm **5** was chosen on a purely intuitive basis. Not only is it guaranteed to work in all scenarios but since it is based on the notion of sending fewer large messages rather than a lot of small messages, it was expected that it would also be one of the most efficient transpose algorithms. However, algorithms **0** & **3** are line-by-line transposes based on the theory prescribed in §4.3.6 and although their portability comes with a caveat, see §4.3.8, based on empirical evidence, they tend to be significantly faster than their plane-by-plane all-weather counterparts. Unfortunately, this fact was not realised until most of the way through this main distributed benchmark. Each node on WALTON has 2 processors and 4 GB of physical memory. Thus, in order to perform this benchmark, the '*-pernode*' flag, which tells MPICH that the user wants to only spawn one process per node, i.e., 1 processor and 4 GB of physical memory per node, even though only 64 were actually used during the benchmarking process, 128 processors had to be commandeered for each run. Due to the fact that this is a large subset of the total number of processors on WALTON, the high demand for these resources, the large number of runs that were averaged for the benchmark and the time restraints associated with the project, it was not feasible to repeat this benchmark with other transpose algorithms. Thus, although the quoted timing values are more than satisfactory, it is expected that the associated communication values, which are comparable to the computation times,

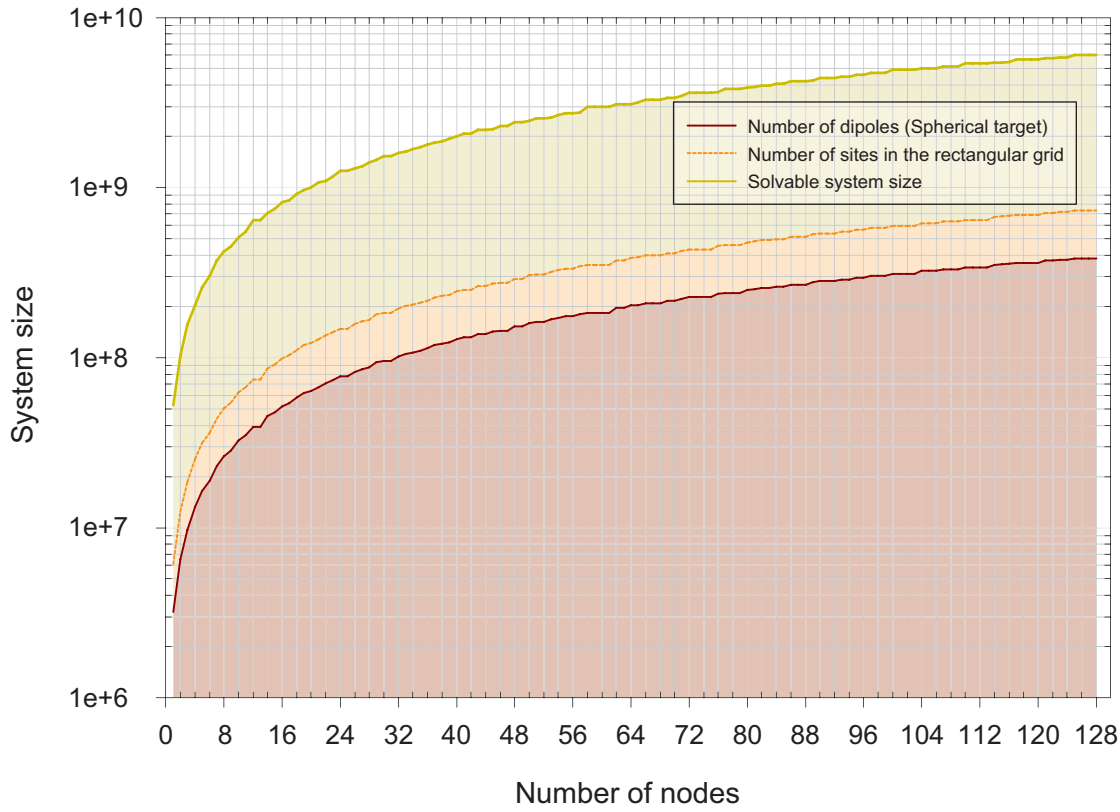



Figure 6.10: A heuristic plot for the maximum number of dipoles that can be used within the standalone double-precision DFT matrix-vector kernel for a spherical target as a function of the number of processors (assuming 2 GB per node for data storage).

could be improved upon by employing either of the line-by-line based transpose algorithms, see algorithms **0** & **3**, §4.3.8.

The use of parallel memory allocation and distributed computing in general has profound implications in terms of the scalability and applicability of the host algorithm which it supports. It extends the range far beyond that of the serial case and can be epitomised by the proverb "*Many hands make light work*". For the DFT matrix-vector product kernel, the allocation in parallel means that the range of the algorithm has been extended by several orders of magnitude. Fig. 6.10 gives an approximate indication of the number dipoles that can be used within the standalone double-precision DFT matrix-vector kernel as a function of the number of nodes (assuming 2GB per node for data storage). The bottom line (dark red) is the upper limit for the number of dipoles for a spherical target, the middle line (dashed orange)

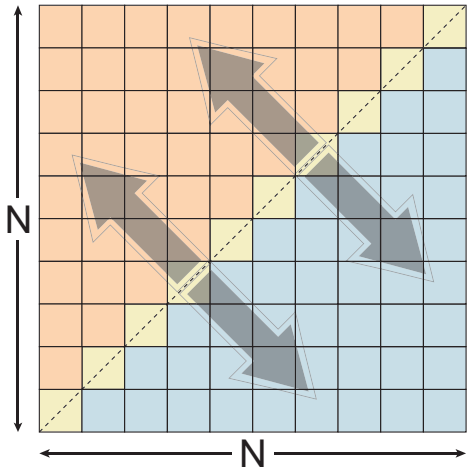



Figure 6.11: Simple in-place local transpose for a square system, i.e., basic flip across the diagonal.

Code Excerpt 6.1: Simple in-place local transpose for a square system, i.e., simple flip across the diagonal.

```
int j,k,source,dest,N,temp,matrix[N*N];

/* Swap elements kN+j & jN+k */
for(k=1;k<N;k++){ /* Row index */
    for(j=0;j<k;j++){ /* Column index */
        source=k*N+j;
        dest=j*N+k;
        temp=matrix[source];
        matrix[source]=matrix[dest];
        matrix[dest]=temp;
    }
}
```

is the size of the encompassing computational domain, which is also the upper limit for the number of dipoles for a rectangular target, and the top line (off yellow) is the size of the system to be solved, i.e., the size after the T-to-c conversion has been performed.

All the distributed-transpose algorithms, except the line-by-line based transpose algorithms, see algorithms **0** & **3**, §4.3.8, are supported by custom-built local in-place transposes which simplify the host algorithm, see §4.3.5. To access the performance of these local transposes, two separate benchmarks were performed on NEMO.

The first involved comparing the performance of the new custom non-square algorithm to a simple in-place local transpose for a square system, i.e., simple flip across the diagonal, see Fig. 6.11 & Excerpt 6.1. The benchmark was run with $K = 100, 200 \ldots 900$, and to create 'square' systems, $J = P = 100, 200..900$, with a maximum memory requirement of 10.86 GB for $K = J = P = 900$. For completeness, both forward and reverse directions were timed, the results of which are shown in Fig. 6.12. The benchmark shows that for 'square' systems, the results for the custom non-square algorithm are practically identical to those of the simple square algorithm in both directions, which is theoretically, the fastest way to carry out the in-place local transpose.

The second benchmark involved testing the performance of the new custom non-square in-place algorithm against a simple point-to-point out-of-place algorithm for a range of non-square systems. The benchmark was run with $K = 100, 200 \ldots 700$,

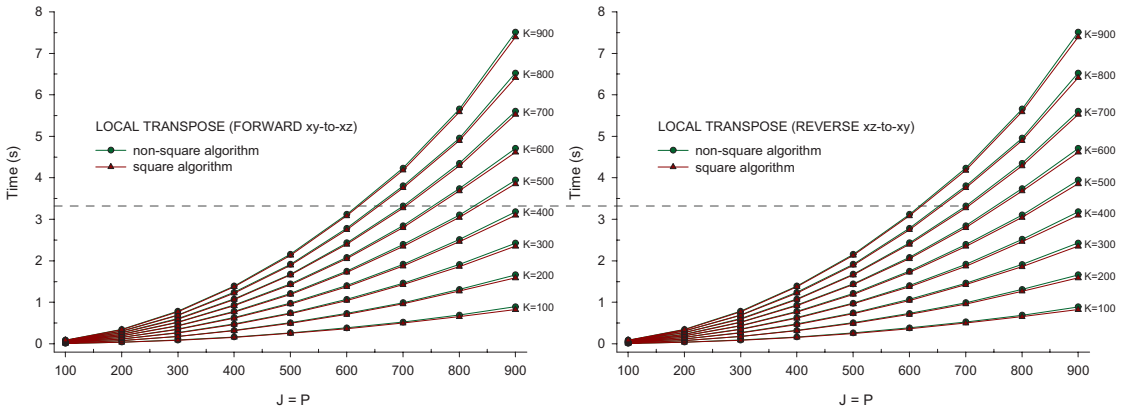



Figure 6.12: A comparison of the custom non-square and basic square in-place local transpose algorithms on NEMO.

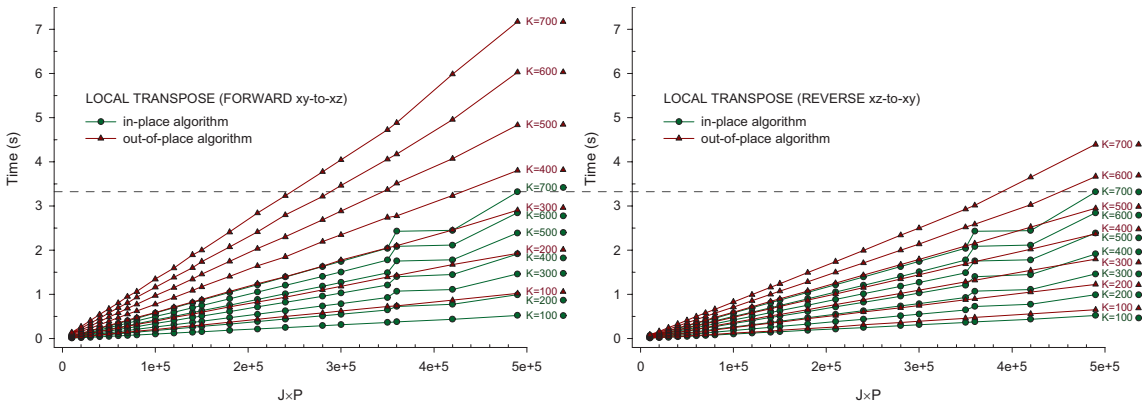



Figure 6.13: A comparison of the custom in-place and simple out-of-place non-square local transpose algorithms on NEMO.

$J = 100, 200 \ldots 700$ & $P = 100, 200 \ldots 700$, with a maximum memory requirement of 10.22 GB for $K = J = P = 700$. The results are shown in Fig. 6.13 and as with the previous benchmark, both forward and reverse directions have been included. The results show that the new custom non-square in-place algorithm is noticeably faster than the simple point-to-point out-of-place algorithm in both directions. This is most likely due to the fact that the new custom non-square in-place algorithm is moving data locally within the same array, whereas the simple point-to-point out-of-place algorithm is moving data over a larger distance from one array to another. To show consistency between the four separate plots in Figs. 6.12 & 6.13, a horizontal line has been drawn at $K = J = P = 700$ which gives a value of 3.319 s in Fig. 6.12 and 3.323 s in Fig. 6.13.

As a final note on the performance of the distributed-memory implementation, Fig. 6.14 shows a comparison of the serial, OpenMP-based shared-memory and MPI-

based distributed-memory implementations of the DFT matrix-vector product kernel using 4 processors on NEMO. The comparison was run over a range up to and including $K = J = P = 310$, which gives a maximum of $N = 29,791,000$ sites in the computational domain and $N_d = 15,599,512$ occupied sites in the representation of the spherical scatterer, with a maximum memory requirement of 6.887 GB for the serial implementation, 6.940 GB for the shared-memory implementation and 9.703 GB, with 2.429 GB per process, for the distributed-memory implementation. The results are the average of three separate runs and as with the benchmark for the main distributed-memory implementation in Fig. 6.9, algorithm **5**, see §4.3.8, was used. Although it is duly noted that due to the necessity of distributed transposes and all the inter-processor communication that that entails, the performance of the MPI version will always lag behind that of the OpenMP version, even for these small systems, both versions provide noticeable speedups that improve as the system size is increased. The fact that the MPI version lags behind that of the OpenMP version is inconsequential as the MPI version was never meant to be used in a shared-memory environment and furthermore, the lag is offset by the fact that the distributed-memory implementation employs parallel memory allocation which provides a decrease in memory that is directly proportional to the number of active nodes, enabling the treatment of hitherto impracticable systems.

It might be argued that OpenMP-based shared-memory implementations provide the same level of benefits, however, in the author's opinion, this is not the case. Although, it is duly accepted that the concept of shared-memory systems, without the need for any interprocess communication, is by far the most appealing, in reality, pound-for-pound, shared-memory machines are far more expensive than their distributed counterparts. Furthermore, the quality of the architectural infrastructure limits the number of processors. The two shared-memory systems that the author has access to, FRANKLIN & HAMILTON, exemplify this point. FRANKLIN has 24 processors and 96 GB of physical memory, whereas HAMILTON, which has $\frac{1}{4}$ TB of physical memory, has only 8 processors more. Due to memory contention issues, it is often not feasible to increase the number of processors arbitrarily as the amount of physical memory is increased. However, distributed-memory systems, which are often said to be just a side effect of economic necessity, are not subject to these issues and by definition, as the total available memory goes up, so does the number of processors.

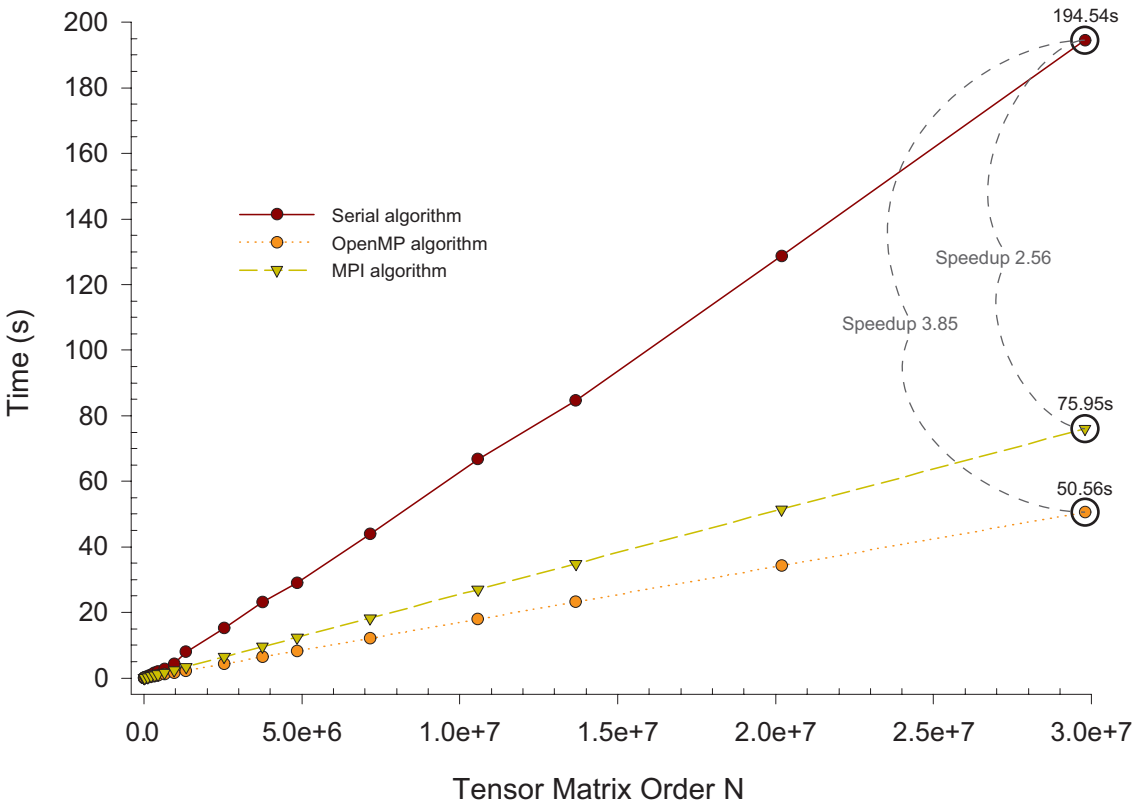



Figure 6.14: A comparison of the serial, OpenMP-based shared-memory and MPI-based distributed-memory implementations on NEMO.

## 6.3 Full system performance

Apart from accessing the performance of the custom-built serial, OpenMP-based shared-memory and MPI-based distributed-memory implementations of the DFT kernel, several benchmarks were also performed to ascertain the performance of the iterative methods that are driven by these new DFT kernels. The following set of benchmarks test the performance of these custom-built iterative methods across the various supported architectures. As with the DFT kernel, each of the iterative schemes has serial, OpenMP-based shared-memory and MPI-based distributed-memory implementations. This is to ensure that the methods integrate flawlessly with the host algorithm and with the new highly-optimised kernel that performs the requisite matrix-vector products.

For each of the architecture specific implementations, the benchmarks were performed using the double-precision version. For all benchmarks, the convergence tol-

erance was taken as the default value of $1\times10^{-10}$, the maximum number of iterations was set to $1\times10^{4}$, the initial guess was set to zero and Point-Jacobi preconditioning was used.

To test the full serial implementation of OpenDDA, a benchmark was run on NEMO with $K = J = P = 200$. This produced a computational domain with $N = 8\times10^{6}$ sites in total and a representation of the spherical target with $N_d = 4,188,896$ occupied sites. The results, which are the average of three runs, are shown in Fig. 6.15. The results provide, not only the performance of the iterative schemes as a whole but, in order to give a feel for the contribution of the underlying matrix-vector products to the overall runtime, the results also give the number of required matrix-vector products and a complete breakdown of their performance.

Due to that amount of timing data that has to be displayed in order to achieve this transparency, each of the constituent components has been assigned a short identifying tag, as listed below.

- **SV**: Number of staring vectors
- **NI**: Number of iterations to converge to the prescribed tolerance
- **CT**: Time to converge to the prescribed tolerance
- **MVPs**: Number of matrix-vector products required by the iterative scheme during convergence
- **MVP**: Average time taken to perform a matrix-vector product
- **MVP BREAKDOWN**: A breakdown of the average time taken by each component of the matrix-vector product
- **VI**: Average time taken to initialise the zero-padded vector components
- **TDFT**: Average time taken to perform the DFT of the six independent tensor components
- **CORE**: Average time taken to perform the DFT of the three zero-padded vector components, the element-wise FD multiplication of the tensor & vector components and the iDFT of the resultant
- **NDDC**: Average time taken to normalise the resultant, add the diagonal contribution and compress the resultant to only include entries corresponding to occupied sites
- **Memory**: Memory required to perform the benchmark

Since efficiency is the name of the game, for each benchmark, the three fastest iterative schemes, i.e., with the shortest time to reach convergence, denoted by **CT**, have been highlighted in each table.

For the OpenMP-based shared-memory implementation, the full system benchmark was run using 4 threads on NEMO, with the same parameter set as for the serial case, and the results, which are shown in Fig. 6.16, are also the average of three separate runs. Note that slightly more memory is required than for the serial case. This is due to the extra DFT scratch arrays that are needed to ensure thread safety. There are two differences between the presentation of the serial results in Fig. 6.15 and these results. First of all, as mentioned previously, to avoid the loss of pertinent data, the data compression algorithm at the end of the DFT matrix-vector product

| Scheme | SV | NI | CT (s) | MVPs | MVP (s) | MVP BREAKDOWN VI (s) | TDFT (s) | CORE (s) | NDDC (s) | Memory (GB) |
|---|---|---|---|---|---|---|---|---|---|---|
| bicg | - | 21 | 1439.701 | 42 | 33.461 | 0.533 | 11.445 | 32.163 | 0.491 | 3.134 |
| $bicg_{sym}$ | - | 21 | 741.503 | 21 | 34.333 | 0.548 | 11.359 | 32.753 | 0.491 | 2.572 |
| bicgstab | - | 12 | 846.555 | 24 | 34.236 | 0.545 | 11.416 | 32.724 | 0.492 | 3.134 |
| cg | - | 21 | 746.576 | 21 | 34.569 | 0.548 | 11.549 | 32.980 | 0.491 | 2.572 |
| cgs | - | 13 | 911.289 | 26 | 34.118 | 0.543 | 11.504 | 32.640 | 0.491 | 3.134 |
| mlbicgstab | 1 | 12 | 868.546 | 24 | 34.469 | 0.543 | 11.564 | 32.979 | 0.491 | 4.070 |
| mlbicgstab | 2 | 8 | 831.932 | 23 | 34.219 | 0.546 | 11.436 | 32.680 | 0.495 | 4.819 |
| mlbicgstab | 3 | 6 | 817.754 | 22 | 34.367 | 0.542 | 11.480 | 32.883 | 0.499 | 5.568 |
| $mlbicgstab_{orig}$ | 1 | 12 | 862.585 | 24 | 34.254 | 0.545 | 11.568 | 32.736 | 0.491 | 4.070 |
| $mlbicgstab_{orig}$ | 2 | 8 | 848.450 | 23 | 34.578 | 0.542 | 11.449 | 33.038 | 0.495 | 4.819 |
| $mlbicgstab_{orig}$ | 3 | 6 | 813.385 | 22 | 34.076 | 0.543 | 11.429 | 32.509 | 0.499 | 5.568 |
| $mlbicgstab_{ss}$ | 1 | 12 | 836.006 | 23 | 34.421 | 0.543 | 11.405 | 32.922 | 0.491 | 3.321 |
| $mlbicgstab_{ss}$ | 2 | 8 | 811.673 | 23 | 33.455 | 0.545 | 11.376 | 31.893 | 0.517 | 3.883 |
| $mlbicgstab_{ss}$ | 3 | 6 | 842.932 | 23 | 34.339 | 0.545 | 11.426 | 32.779 | 0.517 | 4.445 |
| qmr | - | 22 | 3704.595 | 109 | 33.423 | 0.523 | 11.468 | 32.296 | 0.495 | 3.508 |
| $qmr_{sym}$ | - | 21 | 759.644 | 21 | 34.659 | 0.548 | 11.421 | 33.076 | 0.491 | 3.134 |
| rbicgstab | 1 | 14 | 996.250 | 28 | 34.491 | 0.540 | 11.509 | 33.044 | 0.491 | 3.134 |
| rbicgstab | 2 | 7 | 990.463 | 28 | 34.173 | 0.540 | 11.490 | 32.727 | 0.493 | 3.508 |
| rbicgstab | 3 | 4 | 851.089 | 24 | 34.172 | 0.544 | 11.440 | 32.654 | 0.493 | 3.883 |
| tfqmr | - | 13 | 1850.298 | 53 | 34.157 | 0.529 | 11.405 | 32.912 | 0.497 | 3.508 |

Figure 6.15: The performance, on NEMO, of the full serial system as a function of the iterative scheme and, where applicable, the number of starting vectors.

kernel has limited scalability in the shared-memory environment. Thus, unlike the serial case, the normalisation, addition of the diagonal contribution and subsequent compression of the resultant to only include entries corresponding to occupied sites, cannot be amalgamated into a single operation in the OpenMP implementation. The data compression must be kept separate, and as a result, Fig. 6.16 has two columns, denoted by **ND** & **DC**, instead of the single **NDDC** column in Figure. 6.15. **ND** denotes just the normalisation of the resultant of the iDFT and addition of the diagonal contribution, whereas **DC** stands for the now distinct data compression operation. The second difference is that the OpenMP results provide an associated OpenMP speedup, obtained with respect to the serial results in Fig. 6.15, using the 4 threads on NEMO, and is denoted by **SU** in the fifth column.

In terms of the speedups provided by the OpenMP-based shared-memory implementation, the results are extremely encouraging. As was seen with respect to the performance of the OpenMP-based DFT kernel, see §6.2.2, the more work that OpenMP has to divide up among the active processors, the less significant the associated overheads will be and the better the scalability of the host algorithm will be. The problem with the parallelisation of iterative methods is that, apart from

| Scheme | SV | NI | CT (s) | SU | MVPs | MVP (s) | MVP BREAKDOWN VI (s) | TDFT (s) | CORE (s) | ND (s) | DC (s) | Memory (GB) |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| bicg | - | 21 | 404.291 | 3.561 | 42 | 9.036 | 0.219 | 3.235 | 8.429 | 0.155 | 0.153 | 3.155 |
| $bicg_{sym}$ | - | 21 | 212.647 | 3.487 | 21 | 9.476 | 0.216 | 3.294 | 8.790 | 0.156 | 0.153 | 2.594 |
| bicgstab | - | 12 | 234.722 | 3.607 | 24 | 9.143 | 0.213 | 3.159 | 8.486 | 0.151 | 0.154 | 3.155 |
| cg | - | 21 | 209.774 | 3.559 | 21 | 9.340 | 0.218 | 3.167 | 8.658 | 0.156 | 0.153 | 2.594 |
| cgs | - | 13 | 257.424 | 3.540 | 26 | 9.245 | 0.218 | 3.165 | 8.593 | 0.157 | 0.154 | 3.155 |
| mlbicgstab | 1 | 11 | 225.023 | 3.860 | 22 | 9.099 | 0.205 | 3.167 | 8.449 | 0.145 | 0.154 | 4.092 |
| mlbicgstab | 2 | 8 | 246.616 | 3.373 | 23 | 9.416 | 0.204 | 3.169 | 8.773 | 0.147 | 0.155 | 4.841 |
| mlbicgstab | 3 | 6 | 241.627 | 3.384 | 22 | 9.124 | 0.207 | 3.166 | 8.467 | 0.146 | 0.153 | 5.590 |
| $mlbicgstab_{orig}$ | 1 | 11 | 236.089 | 3.654 | 22 | 9.700 | 0.208 | 3.158 | 9.049 | 0.147 | 0.154 | 4.092 |
| $mlbicgstab_{orig}$ | 2 | 8 | 256.019 | 3.314 | 23 | 9.662 | 0.208 | 3.146 | 9.013 | 0.147 | 0.153 | 4.841 |
| $mlbicgstab_{orig}$ | 3 | 7 | 295.570 | 2.752 | 26 | 9.452 | 0.203 | 3.188 | 8.819 | 0.148 | 0.153 | 5.590 |
| $mlbicgstab_{ss}$ | 1 | 12 | 238.865 | 3.500 | 23 | 9.528 | 0.204 | 3.291 | 8.877 | 0.148 | 0.153 | 3.343 |
| $mlbicgstab_{ss}$ | 2 | 8 | 246.211 | 3.297 | 23 | 9.336 | 0.203 | 3.149 | 8.692 | 0.148 | 0.152 | 3.904 |
| $mlbicgstab_{ss}$ | 3 | 6 | 266.241 | 3.166 | 23 | 9.581 | 0.207 | 3.173 | 8.946 | 0.150 | 0.154 | 4.466 |
| qmr | - | 22 | 1055.049 | 3.511 | 109 | 9.318 | 0.206 | 3.216 | 8.774 | 0.149 | 0.153 | 3.530 |
| $qmr_{sym}$ | - | 21 | 216.662 | 3.506 | 21 | 9.545 | 0.210 | 3.257 | 8.876 | 0.147 | 0.154 | 3.155 |
| rbicgstab | 1 | 14 | 284.542 | 3.501 | 28 | 9.498 | 0.212 | 3.279 | 8.869 | 0.151 | 0.154 | 3.155 |
| rbicgstab | 2 | 7 | 285.188 | 3.473 | 28 | 9.498 | 0.209 | 3.199 | 8.867 | 0.151 | 0.153 | 3.530 |
| rbicgstab | 3 | 4 | 242.207 | 3.514 | 24 | 9.355 | 0.207 | 3.181 | 8.704 | 0.149 | 0.154 | 3.904 |
| tfqmr | - | 13 | 524.656 | 3.527 | 53 | 9.433 | 0.209 | 3.217 | 8.857 | 0.152 | 0.154 | 3.530 |

Figure 6.16: The performance, using 4 threads on NEMO, of the full OpenMP-based shared-memory system as a function of the iterative scheme and, where applicable, the number of starting vectors.

the requisite matrix-vector products, by comparison, the rest of the algorithms are composed of computationally lightweight vector operations, i.e., dot products, vector addition, subtraction, scaling, etc. As a result, during the development of the shared-memory implementations of these iterative schemes, the author was concerned that the scalability of the vector operations might not be satisfactory and might subvert the advancements that were made during the development of the DFT kernel. However, the scalability figures in Fig. 6.16, with an average of almost $3\frac{1}{2}$ using 4 threads, have assuaged those fears.

For the MPI-based distributed-memory implementation, the benchmarks have the extra dimension of which of the four possible distributed-transpose algorithms, algorithms **0**, **2**, **3** & **4**, see §4.3.8, to employ to support the DFT kernel. Since this quadruples the amount of results for each benchmark, the results have been placed in Appendix D.

Unlike the serial and OpenMP-based shared-memory implementations, the distributed implementation requires interprocess communication, and as a result, the timing breakdown for the requisite matrix-vector products also includes the average times for the distributed transposes that support the DFT kernel. These are identified

| | SYSTEM SIZES FOR THE DISTRIBUTED BENCHMARK | | | |
|---|---|---|---|---|
| **Nodes** | **K=J=P** | **N** | $\mathbf{N_d}$ | **Figure in Appendix D** |
| 4 | 240 | 13,824,000 | 7,238,592 | **D.2** |
| 8 | 310 | 29,791,000 | 15,599,512 | **D.3** |
| 16 | 380 | 54,872,000 | 28,732,192 | **D.4** |
| 32 | 480 | 110,592,000 | 57,904,928 | **D.5** |
| 64 | 600 | 216,000,000 | 113,098,912 | **D.6** |

Table 6.5: The relevant system sizes for the distributed benchmarks of the MPI-based variant of the OpenDDA framework on WALTON, including the number of sites in the computational domain, $N$, and the number of occupied sites in the representation of the spherical target, $N_d$.

by the tags, **TFTR**, **VFTR** & **VRTR**, and denote the average time taken to perform the forward distributed transpose required by the DFT of the six independent tensor components, the average time taken to perform the forward distributed transpose required by the DFT of the three zero-padded vector components and the average time taken to perform the reverse distributed transpose required by the iDFT of the resultant of the FD element-wise multiplication of the tensor & vector components, respectively.

Although, as mentioned previously, the MPI-based distributed-memory implementation was never meant to be used in a shared-memory environment, to give a feel for the relative performance, the full system benchmark was run using 4 processors on NEMO, with the same parameter set as for the serial and shared-memory cases, and the results, which are shown in Fig. D.1, are also the average of three separate runs. As one would expect from the similar comparison of the DFT kernel, see Fig. 6.14, even though the MPI version is being run in a shared-memory environment, which probably means that the interprocess communication is being implemented by some '*memcpy*' type operation via the '*get protocol*', see §4.3.4, the associated overheads effect the timings enough to produce speedups that, although not negligible by any stretch of the imagination, are shy of the OpenMP-based values, see Fig. 6.16.

For a proper distributed test of the performance of the MPI-based implementation, several benchmarks were run on WALTON across 4, 8, 16, 32 & 64 nodes. The results of each are the average of ten separate runs and the system sizes were chosen so as to produce memory requirements that would allow all incorporated iterative schemes to be compared, see Table 6.5.

Although, it is duly conceded that it is difficult to draw any definitive conclusions

from benchmarks based on such a narrow parameter set, all serial, OpenMP-based shared-memory and MPI-based distributed-memory results seem to be in good agreement. For the systems tested they all indicate that the same three iterative methods, which are three of the most basic schemes, consistently outperform the more involved schemes. However, to enable more informed verdicts, benchmarks over a wider parameter space, which were unfortunately precluded due to the time restraints associated with the project, may show that, since convergence tends to be slower for larger refractive index values [6], as the number of requisite iterations increases, the enhanced stability of the more involved iterative methods may prove beneficial and ultimately, may provide superior convergence. Furthermore, apart for the performance of the iterative methods as a whole, the distributed benchmarks support the earlier claim that the line-by-line distributed transposes, based on the theory described in §4.3.6, i.e., algorithms **0** & **3**, tend to be significantly faster than their plane-by-plane counterparts based on the theory described in §4.3.7, i.e., algorithms **2** & **5**, see §4.3.8.

As a final remark, in an analogous manner to the heuristic plot for the maximum number of dipoles that can be used within the standalone double-precision DFT matrix-vector kernel, see Fig. 6.10 in §6.2.2, Fig. 6.17 shows a plot for the maximum number of dipoles that can be used within the full system as a function of the number of processors and the iterative scheme (assuming 2 GB per node for data storage and, where appropriate, three starting vectors). The data with continuous lines represents the upper limits for the number of dipoles for a spherical target, the data with no lines represents the sizes of the encompassing computational domain, which are also the upper limits for the number of dipoles for a rectangular target, and the dotted lines represent the size of the systems to be solved, i.e., the size after the T-to-c conversion has been performed.

## 6.4 Code comparison

To put the new custom-built OpenDDA framework in context, several comparative benchmarks were performed between the various architecture specific implementations of OpenDDA 0.41 and two freely available DDA codes, DDSCAT [5], version 6.1, and ADDA [188, 212], version 0.76. DDSCAT is a Fortran-based implementation developed by Bruce T. Draine from the Princeton University Observatory and Piotr J. Flatau from the Scripps Institution of Oceanography, Uni-

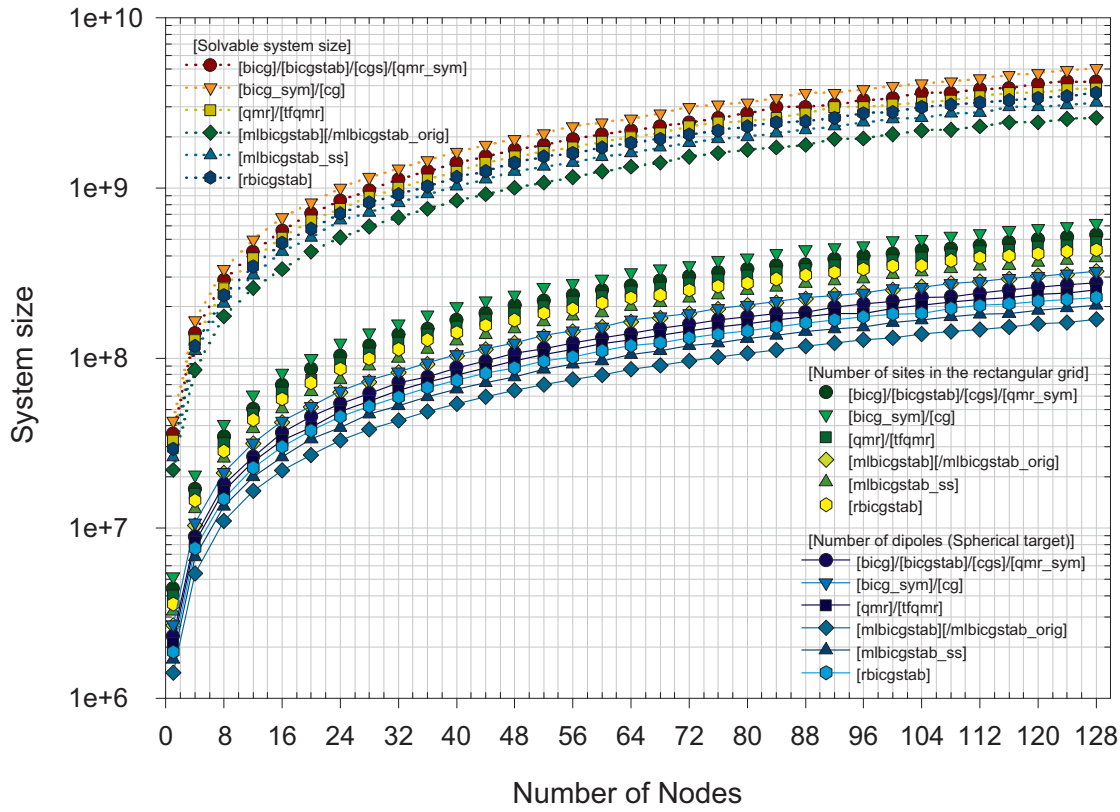



Figure 6.17: A heuristic plot for the maximum number of dipoles that can be used within the full system as a function of the number of processors and the iterative scheme (assuming 2 GB per node for data storage and, where appropriate, three starting vectors).

versity of California, San Diego, whereas ADDA is a C-based implementation developed by Maxim A. Yurkin and Alfons G. Hoekstra at the University of Amsterdam.

Although it is not strictly fair to compare the new OpenDDA framework with DDSCAT, which was originally developed before such convenient access to user-friendly parallel APIs, it serves to set the scene and provides a good indication of the advancements that have been achieved in relation to the memory requirement associated with the new framework. For a meaningful comparison, the comparative benchmark between OpenDDA 0.41 and DDSCAT 6.1 was performed on FRANKLIN using BiCGSTAB and for both the serial and OpenMP-based shared-memory variants of OpenDDA, the single-precision versions were employed. The results are shown in Table 6.6. For the OpenMP-based shared-memory implementation of OpenDDA, the benchmark was run using 4, 8 & 16 threads. The system size was chosen as

| BiCGSTAB | Threads | Iteration (s) | Product (s) | Tensor components Creation (s) | DFT (s) | Total (s) | Memory (GB) |
|---|---|---|---|---|---|---|---|
| DDSCAT | 1 | 145.277 | 55.768 | | | 53.714 | 3.224 |
| openDDA_S | 1 | 43.538 | 18.668 | 1.171 | 14.967 | 16.138 | 1.567 |
| openDDA_OMP | 4 | 16.231 | 6.792 | 0.339 | 4.341 | 4.680 | 1.578 |
| openDDA_OMP | 8 | 9.297 | 3.854 | 0.177 | 2.279 | 2.456 | 1.592 |
| openDDA_OMP | 16 | 5.853 | 2.376 | 0.089 | 1.203 | 1.292 | 1.621 |

Table 6.6: Comparative benchmark run on FRANKLIN between DDSCAT 6.1 and the single-precision serial and OpenMP-based shared-memory variants of OpenDDA 0.41, using BiCGSTAB, for $K = J = P = 200$, which produced a computational domain with $N = 8 \times 10^6$ sites in total and a representation of the spherical target with $N_d = 4,188,896$ occupied sites.

$K = J = P = 200$, which produced a computational domain with $N = 8 \times 10^6$ sites in total and a representation of the spherical target with $N_d = 4,188,896$ occupied sites.

Since the number of iterations to reach convergence is dependent on the user-defined convergence tolerance and on the way in which the stopping criteria are defined within the specific implementation, the comparison focusses, not on the number of iterations, but the time per iteration, which is the fundamental performance factor. In terms of the quoted memory requirements, for the OpenDDA variants, the approximate memory requirements are obtained from the in-built '*guesstimation*' routines, see §5.4. For DDSCAT, to get an estimate for approximate memory usage, it was decided not to take the full memory usage of the code as a whole as this would include contributions that were not affiliated to the task at hand. Instead, due to the inherent unfamiliarity with the inner subtleties of DDSCAT, to get a rough approximation, only the 'large' arrays pertaining to the iterative solution were included, i.e., [*CXADIA*, *CXAOFF*, *CXZC*, *CXZW*, *CXALPH*, *CXALOF*, *CXE* & *CXPOL*].

Table 6.6 lists the number of threads that were employed for each benchmark, the average time per iteration, the average time per matrix-vector product, the average time to initialise the six independent tensor components and finally, an estimate of the memory required by the solution process. For the initialisation of tensor components by the OpenDDA variants, the table also includes a breakdown of the contributions from the two distinct components, i.e., the average time to compute the components and the average time to perform the DFTs of these components. For DDSCAT, the reason that only the total time for the initialisation of the tensor components is

given is that the initialisation is embedded/integrated into the matrix-vector product kernel. Since the initialisation of the tensor components only needs to be recalculated if the wavenumber changes, for this benchmark, the initialisation was only performed once per run. Thus, for simplicity, the total time for the initialisation of the tensor components was taken as the difference between the first call to the matrix-vector product, which included this initialisation, and the average of the subsequent calls, which did not include any tensor component initialisation. Due to the way in which the benchmarks were conducted, for both DDSCAT and OpenDDA, the time per iteration and time for the initialisation of the tensor components are the average of three separate runs, however, the time per matrix-vector product is the average of 24 and 72 separate runs, respectively. The results indicate that for the tested system, the serial and OpenMP-based shared-memory implementation, using 4, 8 & 16 threads, show significant performance improvements. The average time per iteration is between [$\mathbf{3.4} \rightarrow \mathbf{24.8}$] times faster, the average time per matrix-vector product is between [$\mathbf{3.0} \rightarrow \mathbf{23.5}$] times faster, the average time to initialise tensor components is between [$\mathbf{3.3} \rightarrow \mathbf{41.6}$] times faster and the OpenDDA variants also require approximately [$\mathbf{49.7}\% \rightarrow \mathbf{51.4}\%$] less memory for the same calculation.

Next, ADDA 0.76 underwent the same treatment on FRANKLIN. For this comparative benchmark, BiCGSTAB, $\text{BICG}_{\text{sym}}$ & $\text{QMR}_{\text{sym}}$ were used and for both the serial and OpenMP-based shared-memory variants of OpenDDA, the double-precision versions were employed. The system size for the benchmarks are identical to those for the comparison with DDSCAT and, as before, for the OpenMP-based shared-memory implementation of OpenDDA, the benchmarks were run using 4, 8 & 16 threads. The results of the comparisons for BiCGSTAB, $\text{BICG}_{\text{sym}}$ & $\text{QMR}_{\text{sym}}$ are shown in Table 6.7.

For ADDA, the reason that only the total time for the initialisation of the tensor components is given is that ADDA performs the initialisation of the tensor components as a single conjoint operation, i.e., the computation of the components and the DFTs of these components are perform together. Due to the way in which the benchmarks were conducted, for both ADDA and OpenDDA, the time per iteration and time for the initialisation of the tensor components are the average of three separate runs, however, for BiCGSTAB, the time per matrix-vector product is the average of 36 and 72 separate runs, respectively, and for $\text{BICG}_{\text{sym}}$ & $\text{QMR}_{\text{sym}}$, the time per matrix-vector product is the average of 57 and 63 separate runs, respectively.

| | Threads | Iteration | Product | Tensor components | | | Memory |
|---|---|---|---|---|---|---|---|
| | | | | Creation | DFT | Total | |
| **BiCGSTAB** | | (s) | (s) | (s) | (s) | (s) | (GB) |
| ADDA | 1 | 176.779 | 87.050 | | | 86.589 | 3.673 |
| openDDA_S | 1 | 68.576 | 31.412 | 2.044 | 16.657 | 18.702 | 3.134 |
| openDDA_OMP | 4 | 27.914 | 12.185 | 0.499 | 5.179 | 5.678 | 3.155 |
| openDDA_OMP | 8 | 16.527 | 7.090 | 0.249 | 2.778 | 3.028 | 3.184 |
| openDDA_OMP | 16 | 10.056 | 4.203 | 0.134 | 1.505 | 1.639 | 3.241 |
| **$BICG_{sym}$** | | (s) | (s) | (s) | (s) | (s) | (GB) |
| ADDA | 1 | 92.414 | 87.055 | | | 86.415 | 3.111 |
| openDDA_S | 1 | 34.280 | 31.393 | 2.044 | 16.668 | 18.713 | 2.572 |
| openDDA_OMP | 4 | 13.676 | 12.010 | 0.499 | 5.186 | 5.685 | 2.594 |
| openDDA_OMP | 8 | 8.025 | 6.912 | 0.249 | 2.768 | 3.018 | 2.622 |
| openDDA_OMP | 16 | 5.141 | 4.209 | 0.134 | 1.505 | 1.639 | 2.680 |
| **$QMR_{sym}$** | | (s) | (s) | (s) | (s) | (s) | (GB) |
| ADDA | 1 | 92.960 | 86.675 | | | 86.633 | 3.673 |
| openDDA_S | 1 | 35.525 | 31.390 | 2.044 | 16.645 | 18.689 | 3.134 |
| openDDA_OMP | 4 | 14.731 | 12.291 | 0.499 | 5.264 | 5.763 | 3.155 |
| openDDA_OMP | 8 | 8.303 | 6.843 | 0.249 | 2.762 | 3.011 | 3.184 |
| openDDA_OMP | 16 | 5.095 | 4.138 | 0.134 | 1.501 | 1.635 | 3.241 |

Table 6.7: Comparative benchmark run on FRANKLIN between ADDA 0.76 and the double-precision serial and OpenMP-based shared-memory variants of OpenDDA 0.41, using BiCGSTAB, $BICG_{sym}$ & $QMR_{sym}$, for $K = J = P = 200$, which produced a computational domain with $N = 8 \times 10^6$ sites in total and a representation of the spherical target with $N_d = 4,188,896$ occupied sites.

For the associated memory requirements, as with the previous comparative benchmark with DDSCAT, for the OpenDDA variants, the approximate memory requirements are obtained from the in-built '*guesstimation*' routines, see §5.4. To obtain an estimate for the approximate memory requirements for ADDA, as with DDSCAT, it was deemed nonsensical to to take the full memory usage of the code as a whole as this would include contributions that were not relevant to the iterative solution. For this reason, only 'large' arrays pertaining to the iterative solution were included. Furthermore, some of the 'large' arrays were excluded from consideration, as in ADDA, not all relevant 'large' arrays coexist, and thus, some do not contribute to the cumulative memory requirements. The quoted memory requirements for ADDA account

| | cpus | Iteration | Product | Tensor components | | | Memory |
|---|---|---|---|---|---|---|---|
| | | | | Creation | DFT | Total | |
| $\mathbf{QMR_{sym}}$ | | (s) | (s) | (s) | (s) | (s) | (GB) |
| **ADDA** | **1** | **1017.801** | **969.218** | | | **839.165** | **29.265** |
| **openDDA_S** | **1** | **458.938** | **407.502** | **15.795** | **156.366** | **172.161** | **24.998** |
| **openDDA_OMP** | **4** | **129.109** | **109.051** | **3.979** | **44.141** | **48.120** | **25.084** |
| **openDDA_OMP** | **8** | **71.211** | **59.688** | **1.999** | **22.910** | **24.908** | **25.199** |
| **openDDA_OMP** | **16** | **49.283** | **41.923** | **1.059** | **12.009** | **13.067** | **25.428** |

Table 6.8: Comparative benchmark run on FRANKLIN between ADDA 0.76 and the double-precision serial and OpenMP-based shared-memory variants of OpenDDA 0.41, using $\mathrm{QMR_{sym}}$, for $K = J = P = 400$, which produced a computational domain with $N = 64 \times 10^6$ sites in total and a representation of the spherical target with $N_d = 33,511,376$ occupied sites.

for the following arrays. For $\mathrm{BICG_{sym}}$, the memory estimation includes, [*xvec*, *rvec*, *pvec*, *Einc*, *Avecbuffer*, *Dmatrix*, *Xmatrix*, *slices* & *slices_tr*], and for BiCGSTAB & $\mathrm{QMR_{sym}}$, the estimation also includes [*vec1*, *vec2* & *vec3*].

The results in Table 6.7, for the tested system, are quite compelling and are all in good agreement. As with the comparison with DDSCAT, the results for the serial and OpenMP-based shared-memory implementation, using 4, 8 & 16 threads, show significant performance improvements. Overall, the average time per iteration is between [**2.6** → **18.3**] times faster, the average time per matrix-vector product is between [**2.8** → **21.0**] times faster, the average time to initialise tensor components is between [**4.6** → **53.0**] times faster and the OpenDDA variants also require approximately [**11.8**% → **17.3**%] less memory for the same calculation.

To take this comparison one step further, it was decided to raise the stakes to see if a change of scale affected the relative performance. This time the system size was chosen as $K = J = P = 400$, which produced a computational domain with $N = 64 \times 10^6$ sites in total and a representation of the spherical target with $N_d = 33,511,376$ occupied sites. This benchmark was run just before general access to FRANKLIN was restored and due to the associated time restraints, the comparison was only performed with $\mathrm{QMR_{sym}}$, which is ADDA's default iterative scheme. Here, for both ADDA and OpenDDA, the time per iteration and time for the initialisation of the tensor components are the average of three separate runs and the time per matrix-vector product is the average of 87 and 60 separate runs, respectively.

| | cpus | Iteration | Product | Tensor components | | | Memory |
|---|---|---|---|---|---|---|---|
| | | | | Creation | DFT | Total | |
| **BiCGSTAB** | | (s) | (s) | (s) | (s) | (s) | (GB) |
| ADDA | 1 | 77.342 | 37.587 | | | 65.445 | 3.673 |
| openDDA_S | 1 | 68.323 | 32.645 | 1.770 | 11.588 | 13.358 | 3.134 |
| openDDA_OMP | 4 | 19.834 | 9.150 | 0.477 | 3.296 | 3.772 | 3.155 |
| **$BICG_{sym}$** | | (s) | (s) | (s) | (s) | (s) | (GB) |
| ADDA | 1 | 40.184 | 37.584 | | | 65.283 | 3.111 |
| openDDA_S | 1 | 33.894 | 32.385 | 1.770 | 11.562 | 13.332 | 2.572 |
| openDDA_OMP | 4 | 10.446 | 9.640 | 0.477 | 3.280 | 3.757 | 2.594 |
| **$QMR_{sym}$** | | (s) | (s) | (s) | (s) | (s) | (GB) |
| ADDA | 1 | 40.679 | 38.371 | | | 65.534 | 3.673 |
| openDDA_S | 1 | 35.139 | 33.087 | 1.770 | 11.642 | 13.412 | 3.134 |
| openDDA_OMP | 4 | 10.162 | 9.218 | 0.477 | 3.298 | 3.774 | 3.155 |

Table 6.9: Comparative benchmark run on NEMO between ADDA 0.76 and the double-precision serial and OpenMP-based shared-memory variants of OpenDDA 0.41, using BiCGSTAB, $BICG_{sym}$ & $QMR_{sym}$, for $K = J = P = 200$, which produced a computational domain with $N = 8 \times 10^6$ sites in total and a representation of the spherical target with $N_d = 4,188,896$ occupied sites.

The results of this comparative benchmark, see Table 6.8, compound the performance gains obtained for the $K = J = P = 200$ case in Fig. 6.7. The average time per iteration is between [**2.2** → **20.7**] times faster, the average time per matrix-vector product is between [**2.4** → **23.1**] times faster, the average time to initialise tensor components is between [**4.9** → **64.2**] times faster and the OpenDDA variants also require approximately [**13.1**% → **14.6**%] less memory for the same calculation.

With sole access to FRANKLIN no longer available, it was decided move on and to quickly test for any performance related architectural dependencies. The first architectural stop was NEMO. The results for the $K = J = P = 200$ comparison, which this time use a maximum of 4 threads rather than the 16 used on FRANKLIN, see Table 6.9, show performance gains that, although still quite appreciable, are not as profound as for the same system on FRANKLIN. Overall, the average time per iteration is between [**1.1** → **4.0**] times faster, the average time per matrix-vector product is between [**1.2** → **4.2**] times faster, the average time to initialise tensor components is between [**4.9** → **17.4**] times faster and the OpenDDA variants also

| | cpus | Iteration | Product | Tensor components | | | Memory |
|---|---|---|---|---|---|---|---|
| | | | | Creation | DFT | Total | |
| **BiCGSTAB** | | (s) | (s) | (s) | (s) | (s) | (GB) |
| ADDA | 1 | 56.968 | 28.454 | | | 44.671 | 1.554 |
| openDDA_S | 1 | 48.254 | 23.463 | 1.395 | 7.072 | 8.467 | 1.325 |
| openDDA_OMP | 2 | 28.067 | 13.290 | 0.743 | 4.019 | 4.762 | 1.329 |
| **$BICG_{sym}$** | | (s) | (s) | (s) | (s) | (s) | (GB) |
| ADDA | 1 | 30.183 | 28.221 | | | 44.641 | 1.317 |
| openDDA_S | 1 | 24.066 | 23.179 | 1.395 | 7.014 | 8.408 | 1.088 |
| openDDA_OMP | 2 | 13.884 | 13.203 | 0.743 | 3.996 | 4.739 | 1.092 |
| **$QMR_{sym}$** | | (s) | (s) | (s) | (s) | (s) | (GB) |
| ADDA | 1 | 30.414 | 28.547 | | | 44.704 | 1.554 |
| openDDA_S | 1 | 24.470 | 23.192 | 1.395 | 7.032 | 8.427 | 1.325 |
| openDDA_OMP | 2 | 14.297 | 13.169 | 0.743 | 4.018 | 4.762 | 1.329 |

Table 6.10: Comparative benchmark run on HOBBES between ADDA 0.76 and the double-precision serial and OpenMP-based shared-memory variants of OpenDDA 0.41, using BiCGSTAB, $BICG_{sym}$ & $QMR_{sym}$, for $K=J=P=200$, which produced a computational domain with $N=3,375,000$ sites in total and a representation of the spherical target with $N_d=1,768,496$ occupied sites.

require approximately [**14.1**% → **17.3**%] less memory for the same calculation.

The results seem to suggest that the OpenDDA framework seems to favour high-cache environments. Each of the Itanium 2 processors on FRANKLIN was equipped with 6 MB of onboard cache, however, NEMO has just 1 MB of onboard cache per Opteron. To test this hypothesis and add another test architecture to the list, it was decided to run one last serial and OpenMP-based shared-memory comparison on HOBBES. Although, this may seem like a contradiction, since HOBBES is a distributed-memory cluster, only one node was used for the benchmark. The reason HOBBES was chosen is that it is an Intel based architecture with Xeon processors that only have $\frac{1}{2}$ MB of onboard cache, and thus, provides a perfect testing ground to test the cache dependent performance hypothesis. The system size had to be scaled down to to fit in the 2 GB that HOBBES has per node and, since there are only two processors per node, the OpenMP-based shared-memory implementation used a maximum of 2 threads. The system size was dropped to $K = J = P = 150$, which produced a computational domain with $N = 3,375,000$ sites in total and a

| | SYSTEM SIZES FOR THE COMPARISON | | |
|---|---|---|---|
| **Nodes** | **K=J=P** | **N** | **$N_d$** |
| 4 | 240 | 13,824,000 | 7,238,592 |
| 8 | 310 | 29,791,000 | 15,599,512 |
| 16 | 380 | 54,872,000 | 28,732,192 |
| 32 | 480 | 110,592,000 | 57,904,928 |
| 64 | 600 | 216,000,000 | 113,098,912 |

Table 6.11: System sizes, associated size of the computational domain and the number of occupied sites in the computational domain for the MPI-based comparative benchmark with ADDA across on 4, 8, 16, 32 & 64 nodes on WALTON.

representation of the spherical target with $N_d = 1,768,496$ occupied sites.

The results for this benchmark, see Table 6.10, show comparable performance results to the previous results on NEMO, see Table 6.9, and seem to corroborate the hypothesis that the OpenDDA framework does tend to favour high-cache environments. Overall, the average time per iteration is between $[\mathbf{1.2} \rightarrow \mathbf{2.2}]$ times faster, the average time per matrix-vector product is between $[\mathbf{1.2} \rightarrow \mathbf{2.2}]$ times faster, the average time to initialise tensor components is between $[\mathbf{5.3} \rightarrow \mathbf{9.4}]$ times faster and the OpenDDA variants also require approximately $[\mathbf{14.5}\% \rightarrow \mathbf{17.4}\%]$ less memory for the same calculation.

The final comparative benchmark that was performed was for the MPI-based distributed-memory variant of the OpenDDA framework. This benchmark involved a series of comparisons of the double-precision version with ADDA. To get an idea of the relative performance across the available range, the benchmarks were run across on 4, 8, 16, 32 & 64 nodes on WALTON using BiCGSTAB, $\text{BICG}_{\text{sym}}$ & $\text{QMR}_{\text{sym}}$. For consistency, due to the fact that the sizes produce memory requirements per node that stay roughly constant across the five different scales, the system sizes were taken to be the same as for the full system benchmark of the MPI-based distributed implementation of the OpenDDA framework from §6.3, where for convenience, the relevant values have been re-quoted in Table 6.11. An ancillary and, due to the looming deadlines, extremely welcome benefit of this choice is that all requisite timing and memory usage estimations, except for the average time to compute the six independent tensor components, already exist for the OpenDDA framework as part of the full system benchmarks in §6.3. Thus, apart for the time taken to create the tensor components, which only required a quick and lightweight benchmark, for OpenDDA,

| | cpus | Iteration | Product | Tensor components | | | Memory | |
|---|---|---|---|---|---|---|---|---|
| | | | | Creation | DFT | Total | Total | Per node |
| **BiCGSTAB** | | (s) | (s) | (s) | (s) | (s) | (GB) | (GB) |
| ADDA | 4 | 64.798 | 31.179 | | | 41.744 | 7.636 | 2.152 |
| openDDA_MPI | 4 | 54.478 | 25.874 | 17.388 | 0.846 | 18.234 | 6.690 | 1.678 |
| ADDA | 8 | 76.763 | 36.818 | | | 49.917 | 15.396 | 2.259 |
| openDDA_MPI | 8 | 63.931 | 30.587 | 24.422 | 0.887 | 25.309 | 14.678 | 1.842 |
| ADDA | 16 | 79.906 | 36.793 | | | 45.154 | 27.625 | 2.054 |
| openDDA_MPI | 16 | 66.072 | 31.028 | 19.439 | 0.841 | 20.280 | 27.461 | 1.730 |
| ADDA | 32 | 81.532 | 38.153 | | | 44.639 | 54.327 | 2.020 |
| openDDA_MPI | 32 | 71.555 | 34.116 | 23.292 | 0.817 | 24.109 | 54.402 | 1.740 |
| ADDA | 64 | 98.051 | 46.906 | | | 62.863 | 116.670 | 2.202 |
| openDDA_MPI | 64 | 92.184 | 43.708 | 39.404 | 0.785 | 40.189 | 108.857 | 1.744 |
| **$BICG_{sym}$** | | (s) | (s) | (s) | (s) | (s) | (GB) | (GB) |
| ADDA | 4 | 33.057 | 30.579 | | | 41.231 | 6.665 | 1.818 |
| openDDA_MPI | 4 | 27.286 | 25.918 | 17.330 | 0.846 | 18.176 | 5.719 | 1.436 |
| ADDA | 8 | 37.577 | 35.518 | | | 49.187 | 13.304 | 1.872 |
| openDDA_MPI | 8 | 32.338 | 30.528 | 24.173 | 0.887 | 25.060 | 12.586 | 1.581 |
| ADDA | 16 | 39.872 | 38.178 | | | 45.784 | 23.772 | 1.690 |
| openDDA_MPI | 16 | 30.073 | 30.442 | 19.939 | 0.841 | 20.780 | 23.608 | 1.489 |
| ADDA | 32 | 40.583 | 37.882 | | | 45.689 | 45.561 | 1.657 |
| openDDA_MPI | 32 | 35.835 | 34.053 | 22.989 | 0.817 | 23.806 | 46.636 | 1.497 |
| ADDA | 64 | 50.448 | 47.097 | | | 61.916 | 101.502 | 1.823 |
| openDDA_MPI | 64 | 45.784 | 43.147 | 38.413 | 0.785 | 39.198 | 93.689 | 1.507 |
| **$QMR_{sym}$** | | (s) | (s) | (s) | (s) | (s) | (GB) | (GB) |
| ADDA | 4 | 33.161 | 32.533 | | | 41.777 | 7.636 | 2.152 |
| openDDA_MPI | 4 | 27.677 | 26.024 | 17.591 | 0.846 | 18.437 | 6.690 | 1.678 |
| ADDA | 8 | 39.287 | 35.890 | | | 48.121 | 15.396 | 2.259 |
| openDDA_MPI | 8 | 32.771 | 30.215 | 24.456 | 0.887 | 25.343 | 14.678 | 1.842 |
| ADDA | 16 | 40.948 | 37.374 | | | 46.340 | 27.625 | 2.054 |
| openDDA_MPI | 16 | 33.255 | 31.219 | 20.070 | 0.841 | 20.911 | 27.461 | 1.730 |
| ADDA | 32 | 40.296 | 38.142 | | | 46.532 | 54.327 | 2.020 |
| openDDA_MPI | 32 | 37.573 | 35.335 | 22.699 | 0.817 | 23.516 | 54.402 | 1.740 |
| ADDA | 64 | 50.777 | 47.531 | | | 61.392 | 116.670 | 2.202 |
| openDDA_MPI | 64 | 46.850 | 43.834 | 38.191 | 0.785 | 38.976 | 108.857 | 1.744 |

Table 6.12: Comparative benchmark run on WALTON between ADDA 0.76 and the double-precision MPI-based distributed-memory variant of OpenDDA 0.41, using BiCGSTAB, $BICG_{sym}$ & $QMR_{sym}$, across 4, 8, 16, 32 & 64 nodes.

the pertinent data has been distilled directly from Tables D.2→D.6.

For the estimation of the memory requirements of ADDA, on top of the base estimation which was discussed for the previous serial and OpenMP-based shared-memory comparisons, to allow meaningful comparison for the MPI case, the quoted estimations also include the contributions from ADDA's MPI communication buffers, [*BT_buffer* & *BT_rbuffer*]. Apart from the total memory requirements, to look at the efficiency of the decomposition of the associated data across the active nodes, the results also give the maximum memory requirement per node.

The results for this comparison are shown in Fig. 6.12. Due to the amount of data involved, brief summaries of the main points for both the performance and memory requirements are contained below. Apart from the enhanced performance, in terms of the memory requirements, it is worth noting that the custom 'cyclic-plane' decomposition within the OpenDDA framework, as detailed in §4.3.1, allows for a more even distribution of the data across the active nodes. This is clearly evident from the fact that the percentage difference results for the maximum memory requirement per node are significantly larger than those for the total memory required. This point is exemplified by the results for the comparison across 32 nodes, where although OpenDDA requires slightly more memory in total for this case, the maximum memory required by any individual node is significantly less.

- **PERFORMANCE**
  - **For the average time per iteration**: Across 4, 8, 16, 32 & 64 nodes, OpenDDA is between [**15.9**% → **17.5**%], [**14.7**% → **16.7**%], [**17.3**% → **24.6**%], [**6.8**% → **12.2**%] & [**6.0**% → **9.3**%] times faster, respectively.
  - **For the average time per matrix-vector product**: Across 4, 8, 16, 32 & 64 nodes, OpenDDA is between [**15.2**% → **20.0**%], [**14.1**% → **16.9**%], [**15.7**% → **20.3**%], [**7.4**% → **10.6**%] & [**6.8**% → **8.4**%] times faster, respectively.
  - **For the average time to initialise tensor components**: across 4, 8, 16, 32 & 64 nodes, OpenDDA is also between [**55.9**% → **56.3**%], [**47.3**% → **49.3**%], [**54.6**% → **55.1**%], [**46.0**% → **49.5**%] & [**36.1**% → **36.7**%] times faster, respectively.

| | DDSCAT | OpenDDA (DOUBLE) |
|---|---|---|
| **Serial** | $Q_{ext}$=1.26420 | $Q_{ext}$=1.265033441114 |
| | $Q_{abs}$=0.91097 | $Q_{abs}$=0.911212684195 |
| | $Q_{sca}$=0.35327 | $Q_{sca}$=0.353820756918 |
| **OpenMP**[4] | ---- | $Q_{ext}$=1.265033441114 |
| | ---- | $Q_{abs}$=0.911212684195 |
| | ---- | $Q_{sca}$=0.353820756918 |
| **MPI**[4] | ---- | $Q_{ext}$=1.265033441114 |
| | ---- | $Q_{abs}$=0.911212684195 |
| | ---- | $Q_{sca}$=0.353820756918 |

Table 6.13: Results for the operational verification of the new OpenDDA framework.

- **MEMORY REQUIREMENTS**
  - **Across 4 nodes**: OpenDDA requires approximately [**12.4**% → **14.2**%] less memory in total and the maximum memory per node is approximately [**21.0**% → **22.0**%] less.
  - **Across 8 nodes**: OpenDDA requires approximately [**4.7**% → **5.4**%] less memory in total and the maximum memory per node is approximately [**15.6**% → **18.5**%] less.
  - **Across 16 nodes**: OpenDDA requires approximately [**0.6**% → **0.7**%] less memory in total and the maximum memory per node is approximately [**11.9**% → **15.8**%] less.
  - **Across 32 nodes**: OpenDDA requires approximately [**0.1**% → **2.3**%] **MORE** memory in total **BUT** the maximum memory per node is approximately [**9.7**% → **13.9**%] **LESS**.
  - **Across 64 nodes**: OpenDDA requires approximately [**6.7**% → **7.7**%] less memory in total and the maximum memory per node is approximately [**17.3**% → **20.8**%] less.

## 6.5 Result verification

To finish off this results chapter, the following is a rough-and-ready verification of the OpenDDA framework that was performed on NEMO. At present, due to the project's time constraints, the only post-processing that OpenDDA incorporates is the calculation of the extinction, absorption & scattering cross-sections, $C_{sca} \equiv C_{ext} - C_{abs}$, and these cross sections scaled by the geometrical cross section of a sphere of equal volume, $\pi a^2$, see §1.2, the associated efficiencies, $Q_{sca}$, $Q_{ext}$ & $Q_{abs}$ [5]. These were added purely as a last minute verification tool for the new framework. Table 6.13 presents the efficiency results for a $K = J = P = 100$ cubic target as calculated by DDSCAT [5] and OpenDDA for the following set of parameters.

- **Shape:** CUBE, $K = J = P = 100 \Rightarrow N = N_d = 1 \times 10^6$
- **Incident wavelength:** $\lambda = 3.175\,\mu\text{m}$
- **Effective radius:** $a = 0.5\,\mu\text{m}$
- **Incident polarisation:** $\hat{\text{e}}_0 = \hat{\text{x}}$
- **Complex refractive index:** $m = 1.63631 + 0.372i$[31]
- **Target orientation:** $\phi = \theta = \psi = 0^\circ$

The results produced by both DDSCAT [5] and the three different variants of the double-precision OpenDDA framework are in good agreement. Not only do these results verify the operation of OpenDDA against the well proven output of the widely used DDSCAT code but they also demonstrate the level to which the three architecture specific variants of the OpenDDA framework agree. It is a good indication of the numerical stability of the parallel implementations, especially the much more involved distributed-memory implementation. Although, the serial and shared-memory implementations are conceptually quite similar, algorithmically, the distributed-memory implementation is a completely different entity, yet all three predict identical values.

[31] Complex refractive index of ice for $\lambda = 3.175\,\mu\text{m}$ from the REFICE routine, originally developed by Warren in 1984 [176], and subsequently improved by Warren, Gao and Wiscombe in 1995 (Unpublished, source code available in the PyARTS package, a python package developed to compliment the Atmospheric Radiative Transfer Simulator - ARTS. Available from the Satellite Atmospheric Science Group, http://www.sat.ltu.se/arts/tools).

*"It is a mistake to think you can solve any major problems just with potatoes."*

**Douglas Adams**

*"Well.. I'll tell you something about your famous future.. Everyday I wake up, its still the present, the same grimy boring present.. I don't think this future thing exists.."*

**Cale - Titan A.E.**

*"If I had only known, I would have been a locksmith."*

**Albert Einstein**

# 7 Conclusions & Future work

## 7.1 Conclusions

The aim of the project was to design and develop a highly-optimised, portable and maintainable computational framework for the DDA. The OpenDDA framework provides not only improved performance from a serial standpoint but also incorporates both OpenMP-based shared-memory and MPI-based distributed-memory implementations for enhanced performance in the respective parallel environments. On top of the architectural specific nature of the implementations, each also exists in float-precision, double-precision and long double-precision. During the meticulous design and development process, every conceivable effort was made at every stage to maximise the performance, portability and maintainability of each of the components and the new framework as a whole. This even involved incorporating custom complex number support to circumvent any portability issues associated with the lack of native support in the ANSI C89 standard. On top of this, the new framework also incorporates improved Akima interpolation, SFMT Mersenne Twister pseudorandom number generation and an extremely intuitive and user-friendly interface for specifying the simulation parameters, including a sturdy and forgiving parser and a straightforward human-readable input control file.

The new framework incorporates a whole host of iterative methods that have been custom-built to integrate seamlessly with the new highly-optimised DFT kernel that efficiently calculates the matrix-vector products that are required by these iterative schemes. In terms of computational efficiency, a key ingredient is that these custom-

built implementations only store information corresponding to occupied lattice sites, i.e., this boils down to the elimination of zero storage within all iterative solution vectors. For the majority of cases, this provides a significant reduction in the memory required to store these vectors as the target being simulated does not fully utilise the computational domain, generally having fewer occupied sites towards the exterior of the domain. For the prescribed benchmarks, all serial, OpenMP-based shared-memory and MPI-based distributed-memory results seem to be in good agreement that, for the systems that were tested, the same three iterative methods, CG, $\text{BiCG}_{sym}$ & $\text{QMR}_{sym}$, which are three of the most basic schemes, consistently outperform the more involved schemes.

The new highly-optimised DFT kernel performs the requisite 3D DFTs, which are instrumental in the efficient calculation of the matrix-vector products, as ensembles of 1D transforms using the advanced complex interface provided by version 3 of the extremely efficient & highly-portable FFTW package. By doing so, due to the manipulations that are required to facilitate their use, an inordinate fraction of the constituent 1D transforms, which are superfluous, can be removed. The cumulative effect of the constituent components associated with this optimisation reduces the storage associated with the matrix by over 87% and with the vector components by over 75%, which together, provides a reduction of over 83% in the memory required to perform these matrix-vector products. Furthermore, the optimisation reduces the number of requisite 1D transforms that have to be computed by over 60%, which has a comparably profound effect on the time to calculate these matrix-vector products. On top of these improvements, due to the fact that, computationally, the new framework only requires the creation of a limited number of 1D DFT plans, rather than the full 3D DFT plans required by the original algorithm, the initialisation times, which can be significant for the 3D case, become completely negligible. Not only does this optimisation provide a significant reduction in the required memory, and a comparably significant improvement in the performance, but it is also permits parallel techniques to be employed to further enhance the performance. Both the resulting OpenMP-based shared-memory and MPI-based distributed-memory implementations possess extremely good scalability and associated performance. For a $K = J = P = 600$ system, with $N = 216 \times 10^6$ sites in the computational domain and $N_d = 113,098,912$ occupied sites in the representation of the spherical target, the OpenMP-based shared-memory implementation, which required $\approx 50\,\text{GB}$ of memory to perform the calculation, reduced the time from $2575.95\,\text{s} \rightarrow 156.05\,\text{s}$ using 20

threads, i.e., a speedup of $\approx 16.51$ relative to the calculation using 1 thread. For the equivalent system in the distributed environment, the calculation required $\approx 2.22$ GB per node & 107.54 s across 32 nodes and just $\approx 1.19$ GB per node & 54.78 s across 64 nodes, i.e., a speedup of $\approx 1.96$ relative to the 32 node calculation.

As is seen just above, within the distributed-memory implementation of the new framework, the novel parallel memory allocation and domain decomposition schemes provide a decrease in memory per node that is directly proportional to the number of nodes and also provide inherent load balancing. Due to the fact that, as mentioned above, the OpenDDA framework only stores information corresponding to occupied lattice sites, depending on the target being simulated, there can be quite an imbalance in the number of occupied sites per plane. However, the new framework employs custom-designed 'cyclic-plane' decomposition rather than the standard 'block' decomposition, and as a result, boasts an extremely well-balanced data distribution across the active nodes.

To support the distributed DFT kernel which is fundamentally based on these novel parallel memory allocation and domain decomposition schemes, a custom local transpose algorithm and several custom distributed-transpose algorithms have been incorporated into the new framework. Each has been specifically designed to be able to treat completely arbitrary systems that are decomposed across an arbitrary number of nodes. There are four main distributed algorithms that can be considered for use, i.e., algorithms **0**, **2**, **3** & **5**. Algorithms **0** & **3** are based on the exploitation of the eager protocol within the MPI standard, and from empirical evidence, tend to be significantly faster than algorithms **2** & **5**, which are based on the use of local transposes to simplify the subsequent algorithm, especially for larger system sizes across more nodes. For algorithms **0** & **3**, the only caveat, which is a side-effect of the deliberate exploitation of the eager protocol, is that they are only guaranteed to function with MPICH1 in default configuration. For other MPICH1 configurations and other MPI APIs, the so-called 'EAGER_LIMIT' must be altered explicitly. As was mentioned previously, this is not considered as a portability issue as MPICH1 is trivial to obtain & install and any self-respecting computing facility worth its salt should have a default MPICH1 install.

In comparison to other DDA implementations, the new OpenDDA framework performs extremely well. Using 16 threads for the OpenMP-based implementation of the OpenDDA framework, when compared to DDSCAT, for a $K = J = P = 200$ system, with $N = 8 \times 10^6$ sites in the computational domain and $N_d = 4,188,896$ oc-

cupied sites in the representation of the spherical target, the serial & OpenMP-based implementations of OpenDDA require $\approx 49.7\% \rightarrow 51.4\%$ less memory. Furthermore, the average time per iteration, average time per matrix-vector product & average time to initialise the matrix data are approximately $[3.4 \rightarrow 24.8]$, $[3.0 \rightarrow 23.5]$ & $[3.3 \rightarrow 41.6]$ times faster, respectively. For the same comparison with ADDA, the OpenDDA framework requires $\approx 11.8\% \rightarrow 17.3\%$ less memory and the average time per iteration, average time per matrix-vector product & average time to initialise the matrix data are approximately $[2.6 \rightarrow 18.3]$, $[2.8 \rightarrow 21.0]$ & $[4.6 \rightarrow 53.0]$ times faster, respectively. For larger system sizes this disparity is even more pronounced. However, results seem to indicate that the OpenDDA frameworks seems to favour high-cache environments. For the same system run on a smaller cache environment, the performance ratios are marginally less. Using just 4 threads for the comparison, the OpenDDA framework requires $\approx 14.1\% \rightarrow 17.3\%$ less memory and the average time per iteration, average time per matrix-vector product & average time to initialise the matrix data are approximately $[1.1 \rightarrow 4.0]$, $[1.2 \rightarrow 4.2]$ & $[4.9 \rightarrow 17.4]$ times faster, respectively.

Apart from these core optimisations, another novel optimisation which provides a significant reduction in the associated storage, is the bit-fielding of the integer storage associated with the spatial and compositional specification of the target. For a $K = J = P = 500$ system, bit-fielding reduces the memory requirements for the pertinent data from approximately 3.26 GB to just over 0.1 GB.

In the application domain, the implications of these serial and parallel optimisations are profound and can be viewed from two different perspectives. If time is of the essence, then simulations can be performed in just a fraction of the time, and of course, with just a fraction of the memory footprint. However, if system size is paramount, then for equivalent computational overheads, much larger simulations can be performed. Either the size can remain the same and the computational overheads can be significantly reduced or the computational overheads can remain the same and the system size can be increased.

The DDA is, in principle, allowing for the consequent computational overheads imposed by its 3D volumetric nature, capable to treating targets of arbitrary geometry and optical properties. The scheme offers complete freedom in regards to the description of both the spatial extent and composition of the target. This inherent flexibility in terms of the description of the system to be simulated makes the scheme extremely powerful and applicable in a multitude of disparate scientific fields.

However, the price for this functionality, and the single reason that DDA is often dismissed, is the potentially prohibitive limitations associated with the computational scale required to accurately represent the target and to produce meaningful results. DDA's general applicability, future prospects and true potential are reliant on alleviating the associated computational burden via algorithmic and/or computational improvement of the scheme's kernel. Of course, as computational resources become cheaper and faster, and as algorithms become increasingly more efficient, the ideal situation and eventual aim is to be able to perform real-time simulations. Although this may be a long way off for full 3D volumetric simulations, the new OpenDDA framework represents the next logical step in the evolution towards the real-time simulation of non-spherical, porous and inhomogeneous particles.

The new computational framework can be considered to be the most efficient in the world due to: (a) the bit-fielding optimisations employed to streamline the storage of integer data associated with both the spatial and compositional specification of the target; (b) the calculation of the requisite 3D DFTs as ensembles of 1D DFTs, the associated trivialisation of the DFT plan creation, and the subsequent removal of the superfluous 1D transforms from the kernel; (c) the host of custom-built iterative methods designed to integrate seamlessly with the new DFT kernel; (d) the elimination of all zero storage within the iterative solution vectors; (e) the custom-built complex number support to circumvent any portability issues associated with the lack of native support in the ANSI C89 standard; (f) the implementation of improved Akima univariate data interpolation; (g) the implementation of SFMT Mersenne Twister pseudorandom number generation; (h) the provision of an extremely intuitive and user-friendly interface for specifying the simulation parameters, including a sturdy and forgiving parser and a straightforward human-readable input control file; (i) the custom-built OpenMP-based shared-memory implementation of the framework; (j) the custom-built MPI-based distributed-memory implementation of the framework; (k) the extremely efficient data distribution and inherent load balancing associated with the custom-designed 'cyclic-plane' decomposition in the distributed-memory implementation; (l) the decrease in memory per node that is directly proportional to the number of active nodes associated with the novel parallel memory allocation in the distributed-memory implementation; (m) the efficient custom-designed 3D local and distributed transpose algorithms that support the distributed DFT kernel that are capable of treating completely arbitrary systems that are decomposed across an arbitrary number of nodes.

In terms of truly large-scale simulations, apart from the associated time constraints, which are themselves subjective, with the possibility of multiple processors collaborating on a single simulation, and with the novel domain decomposition and parallel memory allocation functionality provided by the distributed implementation of the new framework, the applicability of the scheme is essentially only limited by the number of available nodes. Whereas pre-existing serial-based kernels are limited to targets that are comparable or moderately larger than the wavelength, the prescribed optimisations and improvements associated with the new serial and parallel computational frameworks augment the applicability of the scheme as a whole. Whether it is single particles that are much larger than the wavelength, or particulate, chemical or biological coagulations, agglomerations or aggregations, such as atmospheric ice crystals, meso-scale bacterial clusters, and large cosmic dust aggregates, all are now, within reason, in the realm of DDA applicability.

As a final remark, for interested parties, the source code for all variants of the OpenDDA framework is available from the author upon request.

## 7.2 Future work

In terms of the remit of the project, which was to custom design and develop a highly-optimised, portable, maintainable and user-friendly computational infrastructure for the DDA, not only has the task been fulfilled successfully, but the resulting framework far surpasses any and all initial hopes and expectations for the project.

However, as it stands, the OpenDDA framework is by no means ready for mainstream applicability. As mentioned previously, at present, the OpenDDA framework incorporates only token post-processing exclusively for operational verification of the new infrastructure. Although the requisite post-processing may not be in place, the fundamental infrastructural support for most of the post-processing has already been incorporated into the framework. All that is required to advance OpenDDA to the point where it could become publicly distributable and freely available is the addition of the relevant post-processing, the details of which are, at present, beyond the scope and time restrictions of the project.

Apart from the post-processing aspects, although OpenDDA does support the provision of custom targets in a user-supplied file, due to the limited in-built support

for target generation, the applicability and usability of the new framework would benefit from the addition of a broader set of target generation routines.

From a computational perspective, there are several outstanding issues that should be considered as part of any future development plans. First of all, although OpenDDA has been informally tried and tested with a variety of the most common C compilers, including the Intel Compilers[32], the PathScale Compilers[33], the Portland Group PGI Compilers[34] and the GNU Compiler Collection[35] (GCC), it would be sensible to perform a more formal and rigorous investigation for a wide range of available compilers and to compile a list of recommended compiler flags for each. Furthermore, to extend the scope of the new framework, ports of the new framework to both the Windows and the Mac OS X operating systems would be a worthwhile consideration.

A nice feature that has been implemented in ADDA [188] is the ability to checkpoint. This involves pausing the execution and storing the current state by writing the execution data to disk. Subsequently, the stored data can be read in from disk and the job can resume from where it left off. This functionality can be useful in the parallel environment where fair-share policies require that walltimes be allocated to restrict resource usage. The associated limit on the amount of work that can be done in any one sitting can be circumvented by setting up the system so that when an encroaching walltime is detected, the system automatically saves the current state to disk and exits.

As a final remark, in terms of performance benchmarking, it would be nice to extend the prescribed analysis to encompass a broader parameter space. Benchmarks could be carried out for other target configurations and physical parameters, including larger refractive index values, to ascertain the effects on the performance and convergence properties of the iterative solvers.

[32]Intel Compilers, http://www.intel.com
[33]PathScale Compilers, http://www.pathscale.com
[34]Portland Group PGI Compilers, http://www.pgroup.com
[35]GCC, the GNU Compiler Collection, http://gcc.gnu.org

# Bibliography


[1] OpenMP Architecture Review Board. *OpenMP Application Program Interface* (2005).

[2] Gropp, W. & Lusk, E. *User's guide for* `MPICH`*, a portable implementation of MPI.* Mathematics and Computer Science Division, Argonne National Laboratory (1996). ANL-96/6.

[3] Gropp, W. *et al.* `MPICH2` *User's Guide.* Mathematics and Computer Science Division, Argonne National Laboratory (2006). 1.0.5.

[4] Gabriel, E. *et al.* Open MPI: Goals, concept, and design of a next generation MPI implementation. In *Proceedings, 11th European PVM/MPI Users' Group Meeting*, 97–104 (Budapest, Hungary, 2004).

[5] Draine, B. T. & Flatau, P. J. User guide for the Discrete Dipole Approximation code DDSCAT 6.1 (2004). URL `http://arxiv.org/abs/astro-ph/0409262v2`.

[6] Draine, B. T. The discrete-dipole approximation and its application to interstellar graphite grains. *The Astrophysical Journal* **333**, 848–872 (1988).

[7] Hoekstra, A. G. & Sloot, M. A. The Discrete Dipole Approximation - Possibilities and problems to simulate elastic light scattering. In Eremin, Y. & Wriedt, T, (eds.) *Proceedings of the 1st Workshop on Electromagnetic and Light Scattering - Theory and Applications, Bremen, Germany* (1996).

[8] Lakhtakia, A. Strong and weak forms of the method of moments and the coupled dipole method for scattering of time-harmonic electromagnetic fields. *Int. J. Mod Phy. C* **3**, 583 – 603 (1992). Corrections: 4, 1993, 721 - 722.

[9] Lakhtakia, A. & Mulholland, G. W. On two numerical techniques for light scattering by dielectric agglomerated structures. *J. Res. Natl. Inst. Stand. Technol.* **98**, 699–716 (1993).

[10] DeVoe, H. Optical properties of molecular aggregates. I. Classical model of electronic absorption and refraction. *J. Chem. Phys.* **41**, 393–400 (1964).

[11] DeVoe, H. Optical properties of molecular aggregates. II. Classical theory of the refraction, absorption and optical activity of solution and crystals. *J. Chem. Phys.* **43**, 3199–3208 (1965).

[12] Purcell, E. M. & Pennymaker, C. R. Scattering and absorption of light by non-spherical dielectric grains. *The Astrophysical Journal* **186**, 705–714 (1973).

[13] Goodman, J. J., Draine, B. T. & Flatau, P. J. Application of fast-Fourier transform techniques to the discrete dipole approximation. *Opt. Lett.* **16**, 1198–1200 (1991).

[14] Draine, B. T. & Goodman, J. Beyond Clausius-Mossotti - Wave propagation on a polarizable point lattice and the discrete dipole approximation. *The Astrophysical Journal* **405**, 685–697 (1993).

[15] Draine, B. T. & Flatau, P. J. Discrete-Dipole Approximation for scattering calculations. *J. Opt. Soc. Am. A.* **11**, 1491–1499 (1994).

[16] Brimblecombe, P. *Air Composition and Chemistry* (Cambridge University Press, 1995), 2nd edn.

[17] Pruppacher, H. R. & Klett, J. D. *Microphysics of Clouds and Precipitation* (Springer, 1996).

[18] Wikipedia. Cosmic dust - Wikipedia, The Free Encyclopedia (2008). URL `http://en.wikipedia.org/wiki/Cosmic_dust`.

[19] Graps, A. Cosmic dust and its evolution. Southwest Research Institute, Boulder, Colorado (2000). URL `ftp://ftp.amara.com/pub/papers/dustevolve.txt`.

[20] Li, A. Optical properties of dust. In Mann, I., Nakamura, A. & Mukai, T. (eds.) *Lecture Notes in Physics Series: Small Bodies in Planetary Sciences* (Springer, 2007).

[21] Riess, A. G., Press, W. H. & Kirshner, R. P. Is the dust obscuring supernovae in distant galaxies the same as dust in the milky way? *The Astrophysical Journal* **473**, 588–594 (1996).

[22] Calzetti, D. The dust opacity of starforming galaxies. *Publications of the Astronomical Society of the Pacific* **113**, 1449–1485 (2001).

[23] Elvis, M., Marengo, M. & Karovska, M. Smoking quasars: A new source for cosmic dust. *The Astrophysical Journal Letters* **567**, L107–L110 (2002).

[24] Beelen, A. *et al.* 350 $\mu$m dust emission from highredshift quasars. *The Astrophysical Journal* **642**, 694–701 (2006).

[25] Hanner, M. S. Physical characteristics of cometary dust from optical studies. In Halliday, I. & McIntosh, B. A. (eds.) *Solid Particles in the Solar System*, vol. 90 of *IAU Symposium*, 223–235 (1980).

[26] McDonnell, J. A. M., Lamy, P. L. & Pankiewicz, G. S. Physical properties of cometary dust. In Newburn, Jr., R. L., Neugebauer, M. & Rahe, J. (eds.) *IAU Colloq. 116: Comets in the post-Halley era*, vol. 167 of *Astrophysics and Space Science Library*, 1043–1073 (1991).

[27] Sykes, M. V. & Walker, R. G. Cometary dust trails. I - Survey. *Icarus* **95**, 180–210 (1992).

[28] Cowen, R. Bright comet poses puzzles: Hyakutake's tails of mystery. *Science News* **149**, 346–347 (1996).

[29] Petrova, E. V., Jockers, K. & Kiselev, N. N. Light scattering by aggregates with sizes comparable to the wavelength: An application to cometary dust. *Icarus* **148**, 526–536 (2000).

[30] Ishiguro, M. *et al.* Discovery of the dust trail of the stardust comet sample return mission target: 81P/Wild 2. *The Astrophysical Journal Letters* **589**, 101–104 (2003).

[31] Jones, G. H. *et al.* The Dust Halo of Saturn's Largest Icy Moon, Rhea. *Science* **319**, 1380–1384 (2008).

[32] Granato, G. L., Danese, L. & Franceschini, A. Thick tori around Active Galactic Nuclei: The case for extended tori and consequences for their Xray and infrared emission. *The Astrophysical Journal* **486**, 147–159 (1997).

[33] Nenkova, M., Ivezić, Ž. & Elitzur, M. Dust emission from Active Galactic Nuclei. *The Astrophysical Journal Letters* **570**, 9–12 (2002).

[34] Freudling, W., Siebenmorgen, R. & Haas, M. Hot dust in radioloud Active Galactic Nuclei. *The Astrophysical Journal Letters* **599**, 13–16 (2003).

[35] Krolik, J. H. AGN obscuring tori supported by infrared radiation pressure. *The Astrophysical Journal* **661**, 52–59 (2007).

[36] Tadhunter, C. *et al.* The heating mechanism for the warm/cool dust in powerful, radioloud Active Galactic Nuclei. *The Astrophysical Journal Letters* **661**, 13–16 (2007).

[37] Baggaley, W. J. Interstellar dust in the solar system. *Earth, Moon, and Planets* **95**, 197–209 (2004).

[38] Suh, K. Pulsation phasedependent dust shell models for oxygenrich Asymptotic Giant Branch stars. *The Astrophysical Journal* **615**, 485–494 (2004).

[39] Suh, K. Dust grains in AGB starts as sources of interstellar dust. *Journal of The Korean Astronomical Society* **37**, 289–294 (2004).

[40] Lebzelter, T., Posch, T., Hinkle, K., Wood, P. R. & Bouwman, J. Tracing the development of dust around evolved stars: The case of 47 Tuc. *The Astrophysical Journal Letters* **653**, 145–148 (2006).

[41] Chiar, J. E. & Tielens, A. G. G. M. Circumstellar carbonaceous material associated with latetype dusty WC WolfRayet stars. *The Astrophysical Journal Letters* **550**, 207–211 (2001).

[42] DeMarco, O. & Soker, N. A new look at the evolution of WolfRayet central stars of planetary nebulae. *Publications of the Astronomical Society of the Pacific* **114**, 602–611 (2002).

[43] Marchenko, S. V., Moffat, A. F. J., Vacca, W. D., Cote, S. & Doyon, R. Massive binary WR 112 and properties of Wolf-Rayet dust. *The Astrophysical Journal Letters* **565**, 59–62 (2002).

[44] Monnier, J. D., Tuthill, P. G., Danchi, W. C., Murphy, N. & Harries, T. J. The Keck aperturemasking experiment: Nearinfrared sizes of dusty WolfRayet stars. *The Astrophysical Journal* **655**, 1033–1045 (2007).

[45] Johnson, J. J. & Jones, T. J. From red giant to planetary nebula - dust, asymmetry, and polarization. *The Astrophysical Journal* **101**, 1735–1751 (1991).

[46] Lee, T. H. & Kwok, S. Dust extinction in compact planetary nebulae. *The Astrophysical Journal* **632**, 340–354 (2005).

[47] Gehrz, R. D. *et al.* The infrared spectrum of the optically thin dust shell of V705 Cassiopeiae (Nova Cassiopeiae 1993). *The Astrophysical Journal Letters* **448**, 119–122 (1995).

[48] Smith, R. K. & Dwek, E. Soft Xray scattering and halos from dust. *The Astrophysical Journal* **503**, 831–842 (1998).

[49] Dwek, E. The detection of cold dust in Cassiopeia A: Evidence for the formation of metallic needles in the ejecta. *The Astrophysical Journal* **607**, 848–854 (2004).

[50] Nozawa, T. *et al.* Evolution of dust in primordial supernova remnants: Can dust grains formed in the ejecta survive and be injected into the early interstellar medium? *The Astrophysical Journal* **666**, 955–966 (2007).

[51] Ouellette, N., Desch, S. J. & Hester, J. J. Interaction of supernova ejecta with nearby protoplanetary disks. *The Astrophysical Journal* **662**, 1268–1281 (2007).

[52] Arendt, R. G., Dwek, E. & Moseley, S. H. Newly synthesized elements and pristine dust in the Cassiopeia A supernova remnant. *The Astrophysical Journal* **521**, 234–245 (1999).

[53] Morgan, H. L., Dunne, L., Eales, S. A., Ivison, R. J. & Edmunds, M. G. Cold dust in Keplers supernova remnant. *The Astrophysical Journal Letters* **597**, 33–36 (2003).

[54] Tappe, A., Rho, J. & Reach, W. T. Shock processing of interstellar dust and polycyclic aromatic hydrocarbons in the supernova remnant N132D. *The Astrophysical Journal* **653**, 267–279 (2006).

[55] Rho, J. *et al.* Freshly formed dust in the Cassiopeia A supernova remnant as revealed by the Spitzer Space Telescope. *The Astrophysical Journal* **673**, 271–282 (2008).

[56] Kimura, H., Mann, I. & Jessberger, E. K. Composition, structure, and size distribution of dust in the local interstellar cloud. *The Astrophysical Journal* **583**, 314–321 (2003).

[57] Kimura, H., Mann, I. & Jessberger, E. K. Elemental abundances and mass densities of dust and gas in the local interstellar cloud. *The Astrophysical Journal* **582**, 846–858 (2003).

[58] Calzetti, D. *et al.* The dust content and opacity of actively starforming galaxies. *The Astrophysical Journal* **533**, 682–695 (2000).

[59] Kraemer, K. E. *et al.* Dust characteristics of massive starforming sites in the midinfrared. *The Astrophysical Journal* **561**, 282–298 (2001).

[60] Mason, R. E. *et al.* Dust and PAH emission in the starforming active nucleus of NGC 1097. *The Astrophysical Journal* **659**, 241–249 (2007).

[61] Garay, G., Mardones, D., Brooks, K. J., Videla, L. & Contreras, Y. A multi-wavelength study of young massive star forming regions. II. The dust environment. *The Astrophysical Journal* **666**, 309–320 (2007).

[62] Rosenberg, J. L. *et al.* Dust properties and star formation rates in starforming dwarf galaxies. *The Astrophysical Journal* **674**, 814–830 (2008).

[63] Ishiguro, M. *et al.* First detection of visible zodiacal dust bands from ground-based observations. *The Astrophysical Journal* **511**, 432–435 (1999).

[64] Reynolds, R. J., Madsen, G. J. & Moseley, S. H. New measurements of the motion of the zodiacal dust. *The Astrophysical Journal* **612**, 1206–1213 (2004).

[65] Draine, B. T. On the submillimeter opacity of protoplanetary disks. *The Astrophysical Journal* **636**, 1114–1120 (2006).

[66] Blum, J., Schrapler, R., Davidsson, B. J. R. & TrigoRodriguez, J. M. The physics of protoplanetesimal dust agglomerates. I. Mechanical properties and relations to primitive bodies in the Solar System. *The Astrophysical Journal* **652**, 1768–1781 (2006).

[67] SiciliaAguilar, A. *et al.* Silicate dust in evolved protoplanetary disks: Growth, sedimentation, and accretion. *The Astrophysical Journal* **659**, 1637–1660 (2007).

[68] Ciesla, F. J. Dust coagulation and settling in layered protoplanetary disks. *The Astrophysical Journal* **654**, 159–162 (2007).

[69] Williams, J. P. & Andrews, S. M. The dust properties of eight debris disk candidates as determined by submillimeter photometry. *The Astrophysical Journal* **653**, 1480–1485 (2006).

[70] Besla, G. & Wu, Y. Formation of narrow dust rings in circumstellar debris disks. *The Astrophysical Journal* **655**, 528–540 (2007).

[71] Brownlee, D. E., Tsou, P., Clark, B. C., Swanson, P. N. & Vellinga, J. Stardust: Comet coma sample return plus interstellar dust, science and technical approach (1994). URL `http://stardust.jpl.nasa.gov/science/details.html`.

[72] Jenniskens, P. *Meteor Showers and their Parent Comets* (Cambridge University Press, Cambridge, 2006).

[73] Bradley, J. Organic molecules transport strongest spectral signature of interplanetary dust particles, Lawrence Livermore National Laboratory, California. (2005). URL `https://publicaffairs.llnl.gov/news/news_releases/2005`.

[74] Zeilik, M. & Gregory, S. A. *Introductory Astronomy & Astrophysics* (Saunders College Publishing, 1998), 4th edn.

[75] Gustafson, B. A. S. Physics of zodiacal dust. *Annual Review of Earth and Planetary Sciences* **22**, 553–595 (1994).

[76] Wikipedia. Zodiacal dust - Wikipedia, The Free Encyclopedia (2008). URL `http://en.wikipedia.org/wiki/Zodiacal_dust`.

[77] Wikipedia. Zodiacal light - Wikipedia, The Free Encyclopedia (2008). URL `http://en.wikipedia.org/wiki/Zodiacal_light`.

[78] Darling, D. The Internet Encyclopedia of Science: Zodiacal Cloud (2008). URL `http://www.daviddarling.info/encyclopedia/Z/zodiacal_cloud.html`.

[79] Meixner, M., Campbell, M. T., Welch, W. J. & Likkel, L. Massloss histories of three carbonrich evolved stars as revealed by 12CO emission. *The Astrophysical Journal* **509**, 392–414 (1998).

[80] Hiriart, D. & Kwan, J. Circumstellar gas, dust emission, and mass loss from evolved carbon stars. *The Astrophysical Journal* **532**, 1006–1020 (2000).

[81] Tinsley, B. M. & Cameron, A. G. W. Possible influence of comets on the chemical evolution of the galaxy. *Astrophysics and Space Science* **31**, 31–35 (1974).

[82] Stern, S. A. & Shull, J. M. The influence of Oort clouds on the mass and chemical balance of the interstellar medium. *The Astrophysical Journal* **359**, 506–511 (1990).

[83] Weingartner, J. C. & Draine, B. T. Interstellar depletion onto very small dust grains. *The Astrophysical Journal* **517**, 292–298 (1999).

[84] Williams, D. A. Gas and dust in the interstellar medium. *Journal of Physics: Conference Series* **6**, 1–17 (2005).

[85] Mamon, G. A., Glassgold, A. E. & Huggins, P. J. The photodissociation of CO in circumstellar envelopes. *The Astrophysical Journal* **328**, 797–808 (1988).

[86] Shchekinov, Y. A. Heating of interstellar gas by galactic ultraviolet radiation. *Soviet Astronomy* **23**, 454–459 (1979).

[87] Li, A. On the absorption and emission properties of interstellar grains. In Popescu, C. C. & Tuffs, R. J. (eds.) *The Spectral Energy Distributions of Gas-Rich Galaxies: Confronting Models with Data*, vol. 761 of *American Institute of Physics Conference Series*, 123–133 (2005).

[88] Silk, J. On the fragmentation of cosmic gas clouds. ii - opacity-limited star formation. *The Astrophysical Journal* **214**, 152–160 (1977).

[89] Palla, F., Salpeter, E. E. & Stahler, S. W. Primordial star formation - The role of molecular hydrogen. *The Astrophysical Journal* **271**, 632–641 (1983).

[90] Blain, A. W., Smail, I., Ivison, R. J. & Kneib, J. P. The history of star formation in dusty galaxies. *Monthly Notices of the Royal Astronomical Society* **302**, 632–648 (1999).

[91] Schneider, R., Ferrara, A., Salvaterra, R., Omukai, K. & Bromm, V. Low-mass relics of early star formation. *Nature* **422**, 869 (2003).

[92] Davies, R. D. The relation between interstellar gas, dust and the emission from neutral hydrogen at 21 cm. *Monthly Notices of the Royal Astronomical Society* **116**, 443–452 (1956).

[93] Schweizer, F. Dust and gas - overview. In de Zeeuw, P. T. (ed.) *Structure and Dynamics of Elliptical Galaxies*, vol. 127 of *IAU Symposium*, 109–122 (1987).

[94] Lazarian, A. & Yan, H. Grain dynamics in magnetized interstellar gas. *The Astrophysical Journal Letters* **566**, 105–108 (2002).

[95] Formation and evolution of solids in space. *Space Science Reviews* **90** (1999).

[96] Iatà, M. A., Saija, R., Denti, P., Borghese, F. & Cecchi-Pestellini, C. Exploring the dusty universe. *Atti della Accademia Peloritana dei Pericolanti, Classe di Scienze Fisiche, Matematiche e Naturali* **LXXXIII** (2005).

[97] Wikipedia. Planet formation - Wikipedia, The Free Encyclopedia (2008). URL `http://en.wikipedia.org/wiki/Planet_formation`.

[98] Wikipedia. Star formation - Wikipedia, The Free Encyclopedia (2008). URL `http://en.wikipedia.org/wiki/Star_formation`.

[99] Henning, T. H. & Mutschke, H. Optical properties of cosmic dust analogs. In Sitko, M. L., Sprague, A. L. & Lynch, D. K. (eds.) *Proceedings of the Astronomical Society of the Pacific Conference Series*, 253–272 (2000).

[100] Leinert, C. & Gruen, E. *Interplanetary Dust* (Springer Verlag, 1990).

[101] Mathews, J. D., Janches, D., Meisel, D. D. & Zhou, Q. H. The micrometeoroid mass flux into the upper atmosphere: Arecibo results and a comparison with prior estimates. *Geophysical Research Letters* **28**, 1929–1932 (2001).

[102] Irwin, J. A. *Astrophysics: Decoding the cosmos* (Wiley, 2007).

[103] Brucato, J. R., Strazzulla, G., Baratta, G. A., Saladino, R. & di Mauro, E. Role of cosmic dust analogues in prebiotic chemistry. *International Journal of Astrobiology* **6**, 76–77 (2007).

[104] Greenberg, J. M., Zhao, N. & Hage, J. Chemical evolution of interstellar dust, comets and the origins of life. *Ann. Phys.* **14**, 103–131 (1989).

[105] Basiuk, V. .A & Navarro-Gonzalez, R. Dust in the Universe: implications for terrestrial prebiotic chemistry. *Orig. Life Evol. Biosph.* **25**, 457–493 (1995).

[106] Irvine, W. M. Extraterrestrial organic matter: a review. *Orig. Life Evol. Biosph.* **28**, 365–383 (1998).

[107] Llorca, J. Organic matter in comets and cometary dust. *Int. Microbiol.* **8**, 5–12 (2005).

[108] Matthews, C. N. & Minard, R. D. Hydrogen cyanide polymers, comets and the origin of life. *Faraday Discuss.* **133**, 393–401 (2006).

[109] Lasue, J. & Levasseur-Regourd, A. C. Cosmic dust optical properties: Numerical simulations and future laboratory measurements in microgravity. *Advances in Space Research* **39**, 345–350 (2007).

[110] Xing, Z. & Hanner, M. S. Light scattering by aggregate particles. *Astron. Astrophys.* **324**, 805–820 (1997).

[111] Okamoto, H. & Xu, Y.-L. Light scattering by irregular interplanetary dust particles. *Earth, Planets, and Space* **50**, 577–585 (1998).

[112] Langley Research Center, N. NASA Facts - FS-1996-08-11-LaRC - Atmospheric aerosols: What are they, and why are they so important? (1996). URL `http://www.nasa.gov/centers/langley/news/factsheets/science.html`.

[113] Wikipedia. Particulate - Wikipedia, The Free Encyclopedia (2008). URL `http://en.wikipedia.org/wiki/Particulate`.

[114] NASA. Visible Earth: A catalog of NASA images and animations of our home planet (2008). URL `http://visibleearth.nasa.gov`.

[115] Harlow, D. Ash plume from the Mount Pinatubo eruption. Mount Pinatubo, Philippines - Photo Archives, U. S. Geological Survey (1991). URL `http://vulcan.wr.usgs.gov/Volcanoes/Philippines/Pinatubo/images.html`.

[116] Wikipedia. Aerodynamic diameter - Wikipedia, The Free Encyclopedia (2008). URL `http://en.wikipedia.org/wiki/Aerodynamic_diameter`.

[117] U. S. Government Printing Office. Code of Federal Regulations, Title 40, Volume 5, Part 53, Section 61, 40CFR53.61 (2006).

[118] Griffiths, W. D., Mark, D. & Marshall, I. A. *Aerosol Particle Size Analysis: Good Calibration Practices* (The Royal Society of Chemistry, 1998).

[119] Brunshidle, T. P., Konowalchuk, B., Nabeel, I. & Sullivan, J. E. Ultrafine particles: A review of the measurement, emission, particle characteristics and potential human health impacts of ultrafine particles (2003). URL http://enhs.umn.edu/5103/particles/intro.html.

[120] Jennings, S. G. *Introduction to aerosol effects on climate* (University of Arizona Press, Tucson & London, 1993).

[121] Pope III, C. A. *et al.* Lung cancer, cardiopulmonary mortality, and long-term exposure to fine particulate air pollution. *Journal of the American Medical Association* **287**, 1132–1141 (2002).

[122] Götz, G., Mészáros, E. & Vali, G. *Atmospheric Particles and Nuclei* (Akadémiai Kiadó, Budapest, 1991).

[123] Atmospheric aerosols and global climate: Atmospheric aerosols & climate forcing. School of Physics, University of New South Wales, Sydney (2008). URL http://www.phys.unsw.edu.au/RESEARCH/ATMOSPHERIC/atmospheric_research.html.

[124] Wikipedia. Global dimming - Wikipedia, The Free Encyclopedia (2008). URL http://en.wikipedia.org/wiki/Global_dimming.

[125] Robock, A., Oman, L. & Stenchikov, G. L. Nuclear winter revisited with a modern climate model and current nuclear arsenals: Still catastrophic consequences. *Journal of Geophysical Research (Atmospheres)* **112**, 1–14 (2007).

[126] Wikipedia. Nuclear winter - Wikipedia, The Free Encyclopedia (2008). URL http://en.wikipedia.org/wiki/Nuclear_winter.

[127] Wikipedia. Volcano - Wikipedia, The Free Encyclopedia (2008). URL http://en.wikipedia.org/wiki/Volcano.

[128] Browning, K. A. *et al.* Environmental effects from burning oil wells in kuwait. *Nature* **351**, 363–367 (1991).

[129] Harrison, R. M. & van Grieken, R. E. (eds.) *Atmospheric Particles, IUPAC Series on Analytical and Physical Chemistry of Environmental Systems* (Wiley, 1998).

[130] Wikipedia. Benzopyrene - Wikipedia, The Free Encyclopedia (2008). URL http://en.wikipedia.org/wiki/Benzopyrene.

[131] Wikipedia. Diesel particulate matter - Wikipedia, The Free Encyclopedia (2008). URL http://en.wikipedia.org/wiki/Diesel_particulate_matter.

[132] Wikipedia. Photolithography - Wikipedia, The Free Encyclopedia (2008). URL http://en.wikipedia.org/wiki/Photolithography.

[133] Wikipedia. Very-large-scale integration - Wikipedia, The Free Encyclopedia (2008). URL http://en.wikipedia.org/wiki/Very-large-scale_integration.

[134] Weber, C., Jensen, D. & Hirleman, E. D. What drives defect detection technology? *MICRO* **16**, 51–75 (1998).

[135] Jaffer, A. FreeSnell - Thin-Film Optical Simulator - Version 1b7 (2008). URL http://www-swiss.ai.mit.edu/∼jaffer/FreeSnell.

[136] Warner, T. L. & Hirleman, E. D. Toward classification of particle properties using light scattering techniques. *Journal of the Institute of Environmental Sciences* **40**, 15–21 (1997).

[137] Germer, T. A. Polarized light scattering techniques for surface wafer inspection. In *Electronic-Enhanced Optics, Optical Sensing in Semiconductor Manufacturing, Electro-Optics in Space, Broadband Optical Networks, 2000. Digest of the IEEE LEOS Summer Topical Meetings , Aventura, Florida, USA* (2000).

[138] Mulholland, G. W. & Germer, T. A. Modelling, measurement, and standards for wafer inspection. In *Proceedings of the Government Microcircuits Applications and Critical Technologies (GOMACTech) Conference: "Countering Asymmetric Threats", Tampa, Florida, USA* (2003).

[139] Schmehl, R., Nebeker, B. M. & Hirlemam, E. Discrete dipole approximation for scattering by features on surfaces by means of a two-dimensional fast-fourier transform technique. *J. Opt. Soc Am. A* **14**, 3026–3036 (1997).

[140] Germer, T. A. Scattering by slightly non-spherical particles on surfaces. In *Proceedings of the* $7^{\text{th}}$ *Conference on Electromagnetic and Light Scattering by Nonspherical Particles: Theory, Measurements, and Applications, Bremen, Germany* (2003).

[141] Taubenblatt, M. A. & Tran, T. K. Calculation of light scattering from particles and structures on a surface using the coupled-dipole method. *J. Opt. Soc. Am. A* **10**, 912–919 (1993).

[142] Nebeker, B. M. & Schmehl, R. DDSURF - Modelling electromagnetic light scattering from features on, near, or below flat surfaces (2008). URL `http://meweb.ecn.purdue.edu/~nebeker/ddsurf/DDSURFmain.html`.

[143] Nebeker, B. M., Starr, G. W. & Hirleman, E. Evaluation of iteration methods used when modelling scattering features on surfaces using the discrete dipole approximation. *J. Quant. Spectrosc. Radiat. Trans.* **60**, 493–500 (1998).

[144] Draine, B. T. & Flatau, P. J. User Guide for the Discrete Dipole Approximation Code DDSCAT 6.1 (2006). URL `http://arxiv.org/abs/astro-ph/0409262`.

[145] Nebeker, B. M., de la Peña, J. L. & Hirleman, E. D. Comparisons of the discrete-dipole approximation and modified double interaction model methods to predict light scattering from small features on surfaces. *J. Quant. Spectrosc. Radiat. Trans.* **70**, 749–759 (2001).

[146] Mishchenko, M. I., Hovenier, J. W. & Travis, L. D. *Light Scattering by Non-spherical Particles: Theory, Measurements and Applications* (Academic Press, 2000), 1st edn.

[147] Wikipedia. Organelle - Wikipedia, The Free Encyclopedia (2008). URL `http://en.wikipedia.org/wiki/Organelles`.

[148] Wikipedia. Red blood cell - Wikipedia, The Free Encyclopedia (2008). URL `http://en.wikipedia.org/wiki/Red_blood_cell`.

[149] Wikipedia. White blood cell - Wikipedia, The Free Encyclopedia (2008). URL `http://en.wikipedia.org/wiki/White_blood_cell`.

[150] Sickle Cell Advocates for Research & Empowerment, I. S. What is Sickle Cell Disease? - Sickle Cell Disease: The Basics (2008). URL `http://www.defiers.com/scd.html`.

[151] Wikipedia. Dysplasia - Wikipedia, The Free Encyclopedia (2008). URL `http://en.wikipedia.org/wiki/Dysplasia`.

[152] Liu, Y. *et al.* Simultaneous measurement of angular and spectral properties of light scattering for early cancer detection. *IEEE J. Sel. Top. Quant. Elec.* **9**, 243–256 (2003).

[153] Bigio, I. Elastic-scattering spectroscopy for cancer detection: What have we learned from preliminary clinical studies? In *Proceedings of the Frontiers in Optics, Optical Society of America Technical Digest Series* (2004).

[154] Yurkin, M. A. & Hoekstra, A. G. Capabilities of the discrete dipole approximation for simulation light scattering by biological cells. In *Proceedings of the NATO Advanced Research Workshop: Optics of Biological Particles, Novosibirsk, Russia*, 19–20 (2005).

[155] Johnson, R. M. Ektacytometry of red blood cells. *Methods in Enzymology* **173**, 35–54 (1989).

[156] Streekstra, G. J., Hoekstra, A. G., Nijhof, E. J. & Heethaar, R. M. Light scattering by red blood cells in ektacytometry: Fraunhofer versus anomalous diffraction. *Appl. Opt.* **32**, 2266–2272 (1993).

[157] Wagner, F. F. & Flegel, W. A. Principles and applications of red blood cell flow cytometry. *Infusionsther Transfusionsmed* **25**, 342–346 (1998).

[158] Shvalov, A. N. *et al.* Light-scattering properties of individual erythrocytes. *Appl. Opt.* **38**, 230–235 (1999).

[159] Yurkin, M. A. *et al.* Experimental and theoretical study of light scattering by individual mature red blood cells by use of scanning flow cytometry and a discrete dipole approximation. *Appl. Opt.* **44**, 5249–5256 (2005).

[160] Wriedt, T., Hellmers, J. C., Eremina, E. & Schuh, R. Light scattering by single erythrocyte: Comparison of different methods. *J. Quant. Spectrosc. Radiat. Transf.* **100**, 444–456 (2006).

[161] Wikipedia. Blood - Wikipedia, The Free Encyclopedia (2008). URL `http://en.wikipedia.org/wiki/Blood`.

[162] Wikipedia. Platelet - Wikipedia, The Free Encyclopedia (2008). URL `http://en.wikipedia.org/wiki/Platelet`.

[163] Wikipedia. Mean corpuscular volume - Wikipedia, The Free Encyclopedia (2008). URL http://en.wikipedia.org/wiki/Mean_corpuscular_volume.

[164] Wikipedia. Haemoglobin - Wikipedia, The Free Encyclopedia (2008). URL http://en.wikipedia.org/wiki/Haemoglobin.

[165] Wikipedia. Sickle-cell disease - Wikipedia, The Free Encyclopedia (2008). URL http://en.wikipedia.org/wiki/Sickle_cell.

[166] Yurkin, M. A., Hoekstra, A. G., Brock, R. S. & Lu, J. Q. Systematic comparison of the discrete dipole approximation and the finite difference time domain method for large dielectric scatterers. *Opt. Express* **15**, 17902–17911 (2007).

[167] Mishchenko, M. I., Travis, L. D. & Lacis, A. A. *Scattering, Absorption, and Emission of Light by Small Particles* (Cambridge University Press, 2002), 1st edn.

[168] Bohren, C. F. & Huffman, D. R. *Absorption and scattering of light by small particles* (Wiley, 1998), 1st edn.

[169] Steinke, J. M. & Shepherd, A. P. Comparison of Mie theory and the light scattering of red blood cells. *Appl. Opt.* **27**, 4027–4033 (1988).

[170] Akima, H. A new method of interpolation and smooth curve fitting based on local procedures. *Association for Computing Machinery* **17**, 589–602 (1970).

[171] Akima, H. A method of univariate interpolation that has the accuracy of a third-degree polynomial. *Association for Computing Machinery, Transactions on Mathematical Software* **17**, 341–366 (1991).

[172] Epperson, J. F. *An introduction to numerical methods and analysis* (Wiley, 2002), 1st edn.

[173] Forsythe, G. E., Malcolm, M. A. & Moler, C. B. *Computer methods for mathematical computations* (Prentice Hall, 1977).

[174] Akima, H. A method of smooth curve fitting. Tech. Rep. ESSA Tech. Rep. ERL 101-ITS 73, US Government Printing Office, Washington, D.C. (1969).

[175] Akima, H. Algorithm 697: Univariate interpolation that has the accuracy of a third-degree polynomial. *ACM Transactions on Mathematical Software (TOMS)* **17**, 367–367 (1991).

[176] Warren, S. G. Optical constants of ice from the ultraviolet to the microwave. *Applied Optics* **23**, 218–229 (1984).

[177] Frigo, M. & Johnson, S. G. Fftw: An adaptive software architecture for the fft. *Proc. ICASSP* **3**, 13811384 (1998).

[178] Frigo, M. & Johnson, S. G. The design and implementation of FFTW3. *Proceedings of the IEEE, (invited paper), Special Issue on Program Generation, Optimization, and Platform Adaptation.* **93**, 216–231 (2005).

[179] Intel Corporation. Intel Math Kernel Library (MKL) (2007). Version 9.1.

[180] Galassi, M. *et al.* *GNU Scientific Library - reference manual.* GNU Project (2007). Version 1.9.

[181] Rabinowitz, S. How to find the square root of a complex number. *Mathematics and Informatics Quarterly* **3**, 54–56 (1993).

[182] Hestenes, M. R. & Stiefel, E. Methods of conjugate gradient for solving linear systems. *J. Res. Nat. Bur. Stand.* **49**, 409–436 (1952).

[183] Golub, G. H. & Van Loan, C. F. *Matrix computations* (The Johns Hopkins University Press, Baltimore, 1996), 3rd edn.

[184] Barrett, R. *et al.* *Templates for the solution of linear systems: Building blocks for iterative methods* (SIAM, Philadelphia, 1994), 2nd edn.

[185] Shewchuk, J. R. An Introduction to the Conjugate Gradient Method Without the Agonizing Pain (1994).

[186] Press, W. H., Teukolsky, S. A., Vetterling, W. T. & Flannery, B. P. *Numerical Recipes in C: The Art of Scientific Computing* (Cambridge University Press, 1992).

[187] van der Vorst, H. A. Bi-CGSTAB : A fast and smoothly converging variant of Bi-CG for the solution of nonsymmetric linear systems. *SIAM J. Sci. Stat. Comput.* **13**, 631–644 (1992).

[188] Yurkin, M. A. & Hoekstra, A. G. *User Manual for the Discrete Dipole Approximation Code "Amsterdam DDA"* (2006). Version 0.76.

[189] Freund, R. Conjugate gradient-type methods for linear systems with complex symmetric coefficient matrices. *SIAM Journal on Scientific and Statistical Computing* **13**, 425–448 (1992).

[190] Sonneveld, P. CGS, a fast Lanczos-type solver for nonsymmetric linear systems. *SIAM J. Sci. Stat. Comput.* **10**, 36–52 (1989).

[191] Fletcher, R. Conjugate gradient methods for indefinite systems. In Watson, G. A. (ed.) *Proceedings of the Dundee Biennial Conference on Numerical Analysis*, 73–89 (1975).

[192] Sleijpen, G. L. G. & Fokkema, D. R. BiCGSTAB(L) for linear equations involving unsymmetric matrices with complex spectrum. *Elect. Trans. Numer. Anal.* **1**, 11–32 (1993).

[193] Freund, R. & Nachtigal, N. A quasi-minimal residual method for non-hermitian linear systems. *Numer. Math.* **60**, 315–339 (1991).

[194] Freund, R. & Nachtigal, N. An implementation of the QMR method based on coupled two-term recurrences. *SIAM J. Sci. Comput.* **15**, 313–337 (1992).

[195] Bücker, H. M. & Sauren, M. A parallel version of the unsymmetric Lanczos algorithm and its application to QMR. Tech. Rep. KFA-ZAM-IB-9606, Central Institute for Applied Mathematics, Research Centre Jülich, Germany (1996).

[196] Freund, R. A transpose-free quasi-minimal residual algorithm for non-Hermitian linear systems. *SIAM J. Sci. Comput.* **14**, 470–482 (1993).

[197] Yeung, M. & Chan, T. F. ML(k)BiCGSTAB: A BiCGSTAB variant based on multiple Lanczos starting vectors. *SIAM J. Sci. Comput.* **21**, 1263–1290 (1999).

[198] Yeung, M. & Boley, D. ML(n)BiCGSTAB: reformulations and implementations In press.

[199] Saad, Y. *Iterative methods for sparse linear systems* (SIAM, 2003), 2nd edn.

[200] Strang, G. A proposal for Toeplitz matrix calculations. *Stud. Appl. Math.* **74**, 171–176 (1986).

[201] Chan, T. An optimal circulant preconditioner for toeplitz systems. *SIAM J. Sci. Statist. Comput.* **9**, 766–771 (1988).

[202] Chan, T. & Strang, G. Toeplitz equations by conjugate gradients with circulant preconditioner. *SIAM J. Sci. Statist. Comput.* **10**, 104–119 (1989).

[203] Saito, M. & Matsumoto, M. SIMD-oriented fast mersenne twister: a 128-bit pseudorandom number generator. In *Proceedings of Monte Carlo and Quasi-Monte Carlo Methods, Springer* (2006).

[204] Matsumoto, M. & Nishimura, T. Mersenne Twister: A 623-dimensionally equidistributed uniform pseudo-random number generator. *ACM Transactions on Modeling and Computer Simulation* **8**, 3–30 (1998).

[205] Davis, P. J. *Circulant matrices* (Chelsea Publishing, New York, 1994), 2nd edn.

[206] Hunt, B. R. A matrix theory proof of the discrete convolution theorem. *IEEE Trans. Audio Electroacoust* **AU-19**, 285–288 (1971).

[207] Flatau, P. J. & Draine, B. T. Light scattering by rectangular solids in the discrete-dipole approximation: a new algorithm expoliting the block-toeplitz structure. *J. Opt. Soc. Am. A.* **7**, 593–600 (1990).

[208] Gropp, W., Lusk, E., Doss, N. & Skjellum, A. A high-performance, portable implementation of the MPI message passing interface standard. *Parallel Comput.* **22**, 789–828 (1996).

[209] Balay, S. *et al.* PETSc Users Manual. Tech. Rep. ANL-95/11 - Revision 2.1.5, Argonne National Laboratory (2004).

[210] Whaley, R. C., Petitet, A. & Dongarra, J. J. Automated empirical optimization of software and the ATLAS project. *Parallel Computing* **27**, 3–35 (2001).

[211] Woan, G. *The Cambridge Handbook of Physics Formulas* (Cambridge University Press, 2001).

[212] Yurkin, M. A., Maltsev, V. P. & Hoekstra, A. G. The discrete dipole approximation for simulation of light scattering by particles much larger than the wavelength. *J. Quant. Spectrosc. Radiat. Transf.* **106**, 546–557 (2007).

[213] Van Loan, C. F. *Computational frameworks for the Fast Fourier Transform* (Society for Industrial and Applied Mathematics, Philadelphia, 1992).

[214] da Cunha, R. D. & Hopkins, T. PIM 2.2: The parallel iterative methods package for systems of linear equations. user's guide (fortran 77 version) (1997).

# A

# Circulant proofs

## A.1 Show that the downward-shift permutation matrix ${}^{[n]}\psi$ is a fundamental building block for circulant structure

This is a heavily supplemented adaptation from [213]. A circulant matrix may be formally defined by,

$$\begin{aligned} {}^{[n]}\mathrm{C} &= circ\left(c_0, c_{n-1}, c_{n-2}, \ldots, c_1\right), \\ &= c_0\left[{}^{[n]}\psi^0\right] + c_1\left[{}^{[n]}\psi^1\right] + c_2\left[{}^{[n]}\psi^2\right] + \cdots + c_{n-1}\left[{}^{[n]}\psi^{n-1}\right], \\ &= \sum_{i=0}^{n-1} c_i\left[{}^{[n]}\psi^i\right], \end{aligned} \tag{A.1.1}$$

where ${}^{[n]}\psi$ is the downward-shift permutation matrix and $(c_0, c_1, c_2, \ldots, c_{n-1})$ are the elements of the first column, ${}^{[n]}\mathrm{c}$, of the circulant matrix ${}^{[n]}\mathrm{C}$. Decreasing powers of $i$ introduce a cyclic up-shift in the columns of ${}^{[n]}\psi$, whereas increasing powers of $i$ introduce a cyclic down-shift in the columns. If $u_j$ is the $j^{th}$ column of the identity matrix ${}^{[n]}\mathbf{I}$, then ${}^{[n]}\psi = [u_1|u_2|u_3|\ldots|u_{n-1}|u_0]$ and due to the cyclic property ${}^{[n]}\psi^i = {}^{[n]}\psi^{imodn}\,\forall\, i$. Explicitly, for $n = 4$, these downward-shift permutation matrices are:

$$
\begin{aligned}
{}^{[4]}\psi^0 &= [u_0|u_1|u_2|u_3] = \begin{array}{c} \begin{array}{cccc} u_0 & u_1 & u_2 & u_3 \end{array} \\ \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \end{array} = {}^{[4]}\mathbf{I} \\
{}^{[4]}\psi^1 &= [u_1|u_2|u_3|u_0] = \begin{array}{c} \begin{array}{cccc} u_1 & u_2 & u_3 & u_0 \end{array} \\ \begin{pmatrix} 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix} \end{array} \\
{}^{[4]}\psi^2 &= [u_2|u_3|u_0|u_1] = \begin{array}{c} \begin{array}{cccc} u_2 & u_3 & u_0 & u_1 \end{array} \\ \begin{pmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{pmatrix} \end{array} \\
{}^{[4]}\psi^3 &= [u_3|u_0|u_1|u_2] = \begin{array}{c} \begin{array}{cccc} u_3 & u_0 & u_1 & u_2 \end{array} \\ \begin{pmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \end{pmatrix} \end{array}
\end{aligned}
$$

Substituting these into Eq. A.1.1 gives the circulant matrix ${}^{[4]}\mathrm{C}$.

$$
\begin{aligned}
{}^{[4]}\mathrm{C} &= c_0\left[{}^{[4]}\psi^0\right] + c_1\left[{}^{[4]}\psi^1\right] + c_2\left[{}^{[4]}\psi^2\right] + c_3\left[{}^{[4]}\psi^3\right] \\
&= c_0{}^{[4]}\mathbf{I} + c_1\begin{pmatrix} 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix} + c_2\begin{pmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{pmatrix} + c_3\begin{pmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \end{pmatrix} \\
&= \begin{pmatrix} c_0 & c_3 & c_2 & c_1 \\ c_1 & c_0 & c_3 & c_2 \\ c_2 & c_1 & c_0 & c_3 \\ c_3 & c_2 & c_1 & c_0 \end{pmatrix}
\end{aligned} \tag{A.1.2}
$$

To prove this more formally, we can compare the $j^{th}$ column of the LHS and RHS of Eq. A.1.1. The RHS has the form:

$$\begin{aligned} & \left(c_0 \left[{}^{[n]}\psi^0\right] + c_1 \left[{}^{[n]}\psi^1\right] + c_2 \left[{}^{[n]}\psi^2\right] +, \ldots, + c_{n-2} \left[{}^{[n]}\psi^{n-2}\right] + c_{n-1} \left[{}^{[n]}\psi^{n-1}\right]\right) u_j \\ = \; & c_0 \left[{}^{[n]}\mathbf{I}\right] u_j + c_1 \left[{}^{[n]}\psi^1\right] u_j + c_2 \left[{}^{[n]}\psi^2\right] u_j +, \ldots, + c_{n-2} \left[{}^{[n]}\psi^{n-2}\right] u_j + c_{n-1} \left[{}^{[n]}\psi^{n-1}\right] u_j \end{aligned}$$

Using the fact that ${}^{[n]}\psi^i u_j = u_{[i+j]modn}$ allows this to be rewritten as,

$$c_0 u_{jmodn} + c_1 u_{[j+1]modn} +, \ldots, + c_{n-2} u_{[j+n-2]modn} + c_{n-1} u_{[j+n-1]modn}, \tag{A.1.3}$$

which for $n = 4, j = 0, \ldots, n-1$ Eq. A.1.3 becomes,

$$c_0 u_{jmod4} + c_1 u_{[j+1]mod4} + c_2 u_{[j+2]mod4} + c_3 u_{[j+3]mod4},$$

which explicitly expands to,

$$j = 0 \;\; c_0 u_0 + c_1 u_1 + c_2 u_2 + c_3 u_3 = \begin{pmatrix} c_0 \\ c_1 \\ c_2 \\ c_3 \end{pmatrix} = {}^{[n]}\psi^{0\,[n]}\mathrm{c} \equiv {}^{[n]}\mathrm{C} u_0,$$

$$j = 1 \;\; c_0 u_1 + c_1 u_2 + c_2 u_3 + c_3 u_0 = \begin{pmatrix} c_3 \\ c_0 \\ c_1 \\ c_2 \end{pmatrix} = {}^{[n]}\psi^{1\,[n]}\mathrm{c} \equiv {}^{[n]}\mathrm{C} u_1,$$

$$j = 2 \;\; c_0 u_2 + c_1 u_3 + c_2 u_0 + c_3 u_1 = \begin{pmatrix} c_2 \\ c_3 \\ c_0 \\ c_1 \end{pmatrix} = {}^{[n]}\psi^{2\,[n]}\mathrm{c} \equiv {}^{[n]}\mathrm{C} u_2,$$

$$j = 3 \;\; c_0 u_3 + c_1 u_0 + c_2 u_1 + c_3 u_2 = \begin{pmatrix} c_1 \\ c_2 \\ c_3 \\ c_0 \end{pmatrix} = {}^{[n]}\psi^{3\,[n]}\mathrm{c} \equiv {}^{[n]}\mathrm{C} u_3,$$

$$= {}^{[n]}\psi^{j\,[n]}\mathrm{c} = {}^{[n]}\mathrm{C}u_j = j^{th} \text{ column of the LHS of Eq. A.1.1.} \tag{A.1.4}$$

The $j^{th}$ column of the matrix polynomial equals the $j^{th}$ column of ${}^{[n]}\mathrm{C}$. Since this holds for all $j$, the equality is established

## A.2 Show that the Fourier matrix ${}^{[n]}\mathrm{F}$ diagonalises ${}^{[n]}\psi$

This is a heavily supplemented adaptation from [213]. To show that that the Fourier matrix ${}^{[n]}\mathrm{F}$ diagonalises ${}^{[n]}\psi$, i.e.,

$$ {}^{[n]}\psi = {}^{[n]}\mathrm{F}^{\mathrm{H}\,[n]}\Omega^{[n]}\mathrm{F}, \tag{A.2.5}$$

where ${}^{[n]}\psi$ is the downward-shift permutation matrix, ${}^{[n]}\mathrm{F}$ if the Fourier matrix as defined in Eq. 3.6 and ${}^{[n]}\Omega$ is given by,

$$ {}^{[n]}\Omega = diag\left(1, \omega, \omega^2, \omega^3, \ldots, \omega^{n-1}\right),\ \omega = e^{-\frac{2\pi\imath}{n}}. \tag{A.2.6}$$

Using the fact that the Fourier matrix, ${}^{[n]}\mathrm{F}$, is unitary i.e. ${}^{[n]}\mathrm{F}^{-1} = {}^{[n]}\mathrm{F}^{\mathrm{H}} \equiv \left[{}^{[n]}\mathrm{F}^{*}\right]^{\mathrm{T}} \equiv \left[{}^{[n]}\mathrm{F}^{\mathrm{T}}\right]^{*} \equiv {}^{[n]}\mathrm{F}^{*}$ allows Eq. A.2.6 to be reformulated as:

$$ {}^{[n]}\mathrm{F}^{[n]}\psi \quad = \quad {}^{[n]}\Omega^{[n]}\mathrm{F} \tag{A.2.7}$$

We now proceed by comparing the LHS and RHS of Eq. A.2.7 i.e. the $kj^{th}$ element of ${}^{[n]}\mathrm{F}^{[n]}\psi$ with the $kj^{th}$ element of ${}^{[n]}\Omega^{[n]}\mathrm{F}$. The LHS can be expanded as:

$$\begin{aligned} \left[{}^{[n]}\mathrm{F}^{[n]}\psi\right]_{kj} &= {}^{[n]}\mathrm{F}^{\mathrm{T}}u_k{}^{[n]}\psi u_j = {}^{[n]}\mathrm{F}^{\mathrm{T}}u_k u_{(j+1)modn} \\ &= \frac{1}{\sqrt{n}}\left[\omega^{(k)(0)}\ \ \omega^{(k)(1)}\ \ \omega^{(k)(2)}\ \ \ldots\ \ \omega^{(k)(n-1)}\right] u_{(j+1)modn} \end{aligned} \tag{A.2.8}$$

$$= \frac{1}{\sqrt{n}}\left[\omega^{(k)(j+1)modn}\right] \tag{A.2.9}$$

${}^{[n]}\mathrm{F}^{\mathrm{T}}u_k$ in Eq. A.2.8 extracts the $k^{th}$ row of ${}^{[n]}\mathrm{F}$ and ${}^{[n]}\psi u_j$ extracts the $j^{th}$ column of ${}^{[n]}\psi$. Similarly, the RHS of Eq. A.2.7 can be expanded as:

$$\left[{}^{[n]}\Omega{}^{[n]}\mathrm{F}\right]_{kj} = {}^{[n]}\Omega^{\mathrm{T}}u_k{}^{[n]}\mathrm{F}u_j = {}^{[n]}\Omega^{\mathrm{T}}u_k\frac{1}{\sqrt{n}}\begin{pmatrix}\omega^{(0)(j)}\\ \omega^{(1)(j)}\\ \omega^{(2)(j)}\\ \vdots\\ \omega^{(n-1)(j)}\end{pmatrix} \tag{A.2.10}$$

$$= \frac{1}{\sqrt{n}}\omega^k\omega^{(k)(j)} = \frac{1}{\sqrt{n}}\omega^{k(j+1)modn} \tag{A.2.11}$$

${}^{[n]}\mathrm{F}u_j$ in Eq. A.2.10 extracts the $j^{th}$ column of ${}^{[n]}\mathrm{F}$. ${}^{[n]}\Omega^{\mathrm{T}}u_k$ in Eq. A.2.10 extracts the $k^{th}$ row of ${}^{[n]}\Omega$ but ${}^{[n]}\Omega$ is diagonal so the only non-zero element in the row is $\omega^k$ and when multiplied with ${}^{[n]}\mathrm{F}u_j$ only left with $\omega^k$ by the $k^{th}$ element of the vector ${}^{[n]}\mathrm{F}u_j$ which is $\omega^{(k)(j)}$. It is worth noting that in Eq. A.2.11, since $j$ and $k$ are in the range $0,\ldots,n-1$, $modn$ is needed as $k\,(j+1)$ can be greater than $n-1$. Since $k$ and $j$ are arbitrary ${}^{[n]}\mathrm{F}{}^{[n]}\psi = {}^{[n]}\Omega{}^{[n]}\mathrm{F}$ is established.

## A.3 Show that an arbitrary circulant matrix ${}^{[n]}\mathrm{C}$ is diagonalised by the Fourier matrix ${}^{[n]}\mathrm{F}$

This is a heavily supplemented adaptation from [213]. To show that an arbitrary circulant matrix ${}^{[n]}\mathrm{C}$ is diagonalised by the Fourier matrix ${}^{[n]}\mathrm{F}$, i.e.,

$${}^{[n]}\mathrm{F}{}^{[n]}\mathrm{C}{}^{[n]}\mathrm{F}^{\mathrm{H}} = diag\left(\sqrt{n}\left[{}^{[n]}\mathrm{F}\right]{}^{[n]}\mathrm{c}\right), \tag{A.3.12}$$

where ${}^{[n]}\mathrm{F}$ is the Fourier matrix as defined in Eq. 3.6, ${}^{[n]}\mathrm{C}$ is an arbitrary circulant matrix as defined by Eqs. 3.3 & 3.4 and ${}^{[n]}\mathrm{c}$ is the first column of the circulant matrix ${}^{[n]}\mathrm{C}$. As before we verify Eq. A.3.12 by comparing the LHS and RHS. The LHS can be expanded as:

$$
\begin{aligned}
{}^{[n]}\mathrm{F}\,{}^{[n]}\mathrm{C}\,{}^{[n]}\mathrm{F}^{\mathrm{H}} &= {}^{[n]}\mathrm{F}\left[\sum_{i=0}^{n-1} c_i \left[{}^{[n]}\psi^i\right]\right]{}^{[n]}\mathrm{F}^{\mathrm{H}} \\
&= \sum_{i=0}^{n-1} c_i \left[{}^{[n]}\mathrm{F}\right]\left[{}^{[n]}\psi^i\right]{}^{[n]}\mathrm{F}^{\mathrm{H}} = \sum_{i=0}^{n-1} c_i \left({}^{[n]}\mathrm{F}\,{}^{[n]}\psi\,{}^{[n]}\mathrm{F}^{\mathrm{H}}\right)^i \qquad \text{(A.3.13)}
\end{aligned}
$$

Using Eqs. A.2.7 & A.2.6 it is obvious that,

$$
{}^{[n]}\mathrm{F}\,{}^{[n]}\psi\,{}^{[n]}\mathrm{F}^{\mathrm{H}} = {}^{[n]}\Omega = diag(1, \omega, \omega^2, \omega^3, \ldots, \omega^{n-1}),\ \omega = e^{-\frac{2\pi\imath}{n}}, \qquad \text{(A.3.14)}
$$

and thus, Eq. A.3.13 becomes,

$$
\begin{aligned}
{}^{[n]}\mathrm{F}\,{}^{[n]}\mathrm{C}\,{}^{[n]}\mathrm{F}^{\mathrm{H}} &= \sum_{i=0}^{n-1} c_i diag\left(1, \omega^i, \omega^{2i}, \ldots, \omega^{(n-1)i}\right), \\
&= c_0 diag\left(1, 1, 1, \ldots, 1\right) + c_1 diag\left(1, \omega, \omega^2, \ldots, \omega^{n-1}\right) + \\
&\quad c_2 diag\left(1, \omega^2, \omega^4, \ldots, \omega^{2(n-1)}\right) + \cdots + \\
&\quad c_{n-1} diag\left(1, \omega^{n-1}, \omega^{2(n-1)}, \ldots, \omega^{(n-1)(n-1)}\right), \\
&= diag\left[\sum_{i=0}^{n-1} c_i \ \ \sum_{i=0}^{n-1} c_i\omega^i \ \ \sum_{i=0}^{n-1} c_i\omega^{2i} \ \ \cdots \ \ \sum_{i=0}^{n-1} c_i\omega^{(n-1)i}\right]. \qquad \text{(A.3.15)}
\end{aligned}
$$

The RHS of Eq. A.3.12 can be expanded as:

$$
\begin{aligned}
diag\left(\sqrt{n}\left[{}^{[n]}\mathrm{F}\right]{}^{[n]}\mathrm{c}\right) &= diag\left[\begin{pmatrix} 1 & 1 & 1 & 1 & \ldots & 1 \\ 1 & \omega & \omega^2 & \omega^3 & \ddots & \omega^{n-1} \\ 1 & \omega^2 & \omega^4 & \omega^6 & \ddots & \omega^{2(n-1)} \\ 1 & \omega^3 & \omega^6 & \omega^9 & \ddots & \omega^{3(n-1)} \\ \vdots & \ddots & \ddots & \ddots & \ddots & \vdots \\ 1 & \omega^{n-1} & \omega^{2(n-1)} & \omega^{3(n-1)} & \ldots & \omega^{(n-1)(n-1)} \end{pmatrix}\begin{pmatrix} c_0 \\ c_1 \\ c_2 \\ c_3 \\ \vdots \\ c_{n-1} \end{pmatrix}\right] \\
&= diag\left[\sum_{i=0}^{n-1} c_i \ \ \sum_{i=0}^{n-1} c_i\omega^i \ \ \sum_{i=0}^{n-1} c_i\omega^{2i} \ \ \cdots \ \ \sum_{i=0}^{n-1} c_i\omega^{(n-1)i}\right] \qquad \text{(A.3.16)}
\end{aligned}
$$

Eq. A.3.15 = Eq. A.3.16 $\Rightarrow$ ${}^{[n]}\mathrm{F}\,{}^{[n]}\mathrm{C}\,{}^{[n]}\mathrm{F}^{\mathrm{H}} = diag\left(\sqrt{n}\left[{}^{[n]}\mathrm{F}\right]{}^{[n]}\mathrm{c}\right)$

## A.4 Construction of a circulant of level-two and order $[J'][K']$

Formally, a level-two circulant, ${}^{[J'][K']}\mathrm{C}$, of order $[J'][K'] = [2J-1][2K-1]$ has the form,

$$ {}^{[J'][K']}\mathrm{C} \;=\; \sum_{j=0}^{J'-1} {}^{[J']}\psi^{j} \otimes \left[{}^{[K']}\mathrm{C}_j\right], \tag{A.4.17} $$

where each of the ${}^{[K']}\mathrm{C}_j$ is a level-one circulant of order $[K']$,

$$ {}^{[K']}\mathrm{C}_j \;=\; \sum_{i=0}^{K'-1} \mathrm{C}_{j,i} \left[{}^{[K']}\psi^{i}\right]. $$

For $J = K = 3 \Rightarrow J' = K' = 5$,

$$ {}^{[5][5]}\mathrm{C} \;=\; \sum_{j=0}^{4} \left[{}^{[5]}\psi^{j}\right] \otimes \left[{}^{[5]}\mathrm{C}_j\right], \tag{A.4.18} $$

where each of the ${}^{[5]}\mathrm{C}_j$ is a level-one circulant of order $[5]$:

$$ {}^{[5]}\mathrm{C}_j \;=\; \sum_{i=0}^{4} \mathrm{C}_{j,i} \left[{}^{[5]}\psi^{i}\right] $$

$$ \begin{aligned} {}^{[5]}\mathrm{C}_0 &= \sum_{i=0}^{4} \mathrm{C}_{0,i} \left[{}^{[5]}\psi^{i}\right] \\ &= a\left[{}^{[5]}\psi^{0}\right] + b\left[{}^{[5]}\psi^{1}\right] + c\left[{}^{[5]}\psi^{2}\right] + d\left[{}^{[4]}\psi^{3}\right] + e\left[{}^{[5]}\psi^{4}\right] \\ &= \begin{pmatrix} a & b & c & e & d \\ d & a & b & c & e \\ e & d & a & b & c \\ c & e & d & a & b \\ b & c & e & d & a \end{pmatrix} \\ {}^{[5]}\mathrm{C}_1 &= \sum_{i=0}^{4} \mathrm{C}_{1,i} \left[{}^{[5]}\psi^{i}\right] \\ &= f\left[{}^{[5]}\psi^{0}\right] + g\left[{}^{[5]}\psi^{1}\right] + h\left[{}^{[5]}\psi^{2}\right] + j\left[{}^{[4]}\psi^{3}\right] + i\left[{}^{[5]}\psi^{4}\right] \end{aligned} $$

$$
\begin{aligned}
&= \begin{pmatrix} f & g & h & j & i \\ i & f & g & h & j \\ j & i & f & g & h \\ h & j & i & f & g \\ g & h & j & i & f \end{pmatrix} \\
{}^{[5]}\mathrm{C}_2 &= \sum_{i=0}^{4} \mathrm{C}_{2,i} \left[{}^{[5]}\psi^i\right] \\
&= k\left[{}^{[5]}\psi^0\right] + l\left[{}^{[5]}\psi^1\right] + m\left[{}^{[5]}\psi^2\right] + o\left[{}^{[4]}\psi^3\right] + n\left[{}^{[5]}\psi^4\right] \\
&= \begin{pmatrix} k & l & m & o & n \\ n & k & l & m & o \\ o & n & k & l & m \\ m & o & n & k & l \\ l & m & o & n & k \end{pmatrix} \\
{}^{[5]}\mathrm{C}_3 &= \sum_{i=0}^{4} \mathrm{C}_{3,i} \left[{}^{[5]}\psi^i\right] \\
&= p\left[{}^{[5]}\psi^0\right] + q\left[{}^{[5]}\psi^1\right] + r\left[{}^{[5]}\psi^2\right] + t\left[{}^{[4]}\psi^3\right] + s\left[{}^{[5]}\psi^4\right] \\
&= \begin{pmatrix} p & q & r & t & s \\ s & p & q & r & t \\ t & s & p & q & r \\ r & t & s & p & q \\ q & r & t & s & p \end{pmatrix} \\
{}^{[5]}\mathrm{C}_4 &= \sum_{i=0}^{4} \mathrm{C}_{4,i} \left[{}^{[5]}\psi^i\right] \\
&= u\left[{}^{[5]}\psi^0\right] + v\left[{}^{[5]}\psi^1\right] + w\left[{}^{[5]}\psi^2\right] + y\left[{}^{[4]}\psi^3\right] + x\left[{}^{[5]}\psi^4\right] \\
&= \begin{pmatrix} u & v & w & x & y \\ y & u & v & w & x \\ x & y & u & v & w \\ w & x & y & u & v \\ v & w & x & y & u \end{pmatrix}
\end{aligned}
$$

$$
\begin{aligned}
{}^{[5][5]}\mathrm{C} &= \left[{}^{[5]}\psi^0\right] \otimes \left[{}^{[5]}\mathrm{C}_0\right] + \left[{}^{[5]}\psi^1\right] \otimes \left[{}^{[5]}\mathrm{C}_1\right] + \left[{}^{[5]}\psi^2\right] \otimes \left[{}^{[5]}\mathrm{C}_2\right] + \left[{}^{[5]}\psi^3\right] \otimes \left[{}^{[5]}\mathrm{C}_3\right] + \left[{}^{[5]}\psi^4\right] \otimes \left[{}^{[5]}\mathrm{C}_4\right] \\
&= \left[{}^{[5]}\psi^0\right] \otimes \begin{pmatrix} a & b & c & e & d \\ d & a & b & c & e \\ e & d & a & b & c \\ c & e & d & a & b \\ b & c & e & d & a \end{pmatrix} + \left[{}^{[5]}\psi^1\right] \otimes \begin{pmatrix} f & g & h & j & i \\ i & f & g & h & j \\ j & i & f & g & h \\ h & j & i & f & g \\ g & h & j & i & f \end{pmatrix} + \left[{}^{[5]}\psi^2\right] \otimes \begin{pmatrix} k & l & m & o & n \\ n & k & l & m & o \\ o & n & k & l & m \\ m & o & n & k & l \\ l & m & o & n & k \end{pmatrix} \\
&\qquad + \left[{}^{[5]}\psi^3\right] \otimes \begin{pmatrix} p & q & r & t & s \\ s & p & q & r & t \\ t & s & p & q & r \\ r & t & s & p & q \\ q & r & t & s & p \end{pmatrix} + \left[{}^{[5]}\psi^4\right] \otimes \begin{pmatrix} u & v & w & x & y \\ y & u & v & w & x \\ x & y & u & v & w \\ w & x & y & u & v \\ v & w & x & y & u \end{pmatrix} \\
&= \begin{pmatrix} {}^{[5]}\mathrm{C}_0 & {}^{[5]}\mathrm{C}_4 & {}^{[5]}\mathrm{C}_3 & {}^{[5]}\mathrm{C}_2 & {}^{[5]}\mathrm{C}_1 \\ {}^{[5]}\mathrm{C}_1 & {}^{[5]}\mathrm{C}_0 & {}^{[5]}\mathrm{C}_4 & {}^{[5]}\mathrm{C}_3 & {}^{[5]}\mathrm{C}_2 \\ {}^{[5]}\mathrm{C}_2 & {}^{[5]}\mathrm{C}_1 & {}^{[5]}\mathrm{C}_0 & {}^{[5]}\mathrm{C}_4 & {}^{[5]}\mathrm{C}_3 \\ {}^{[5]}\mathrm{C}_3 & {}^{[5]}\mathrm{C}_2 & {}^{[5]}\mathrm{C}_1 & {}^{[5]}\mathrm{C}_0 & {}^{[5]}\mathrm{C}_4 \\ {}^{[5]}\mathrm{C}_4 & {}^{[5]}\mathrm{C}_3 & {}^{[5]}\mathrm{C}_2 & {}^{[5]}\mathrm{C}_1 & {}^{[5]}\mathrm{C}_0 \end{pmatrix}
\end{aligned}
\tag{A.4.19}
$$

$$
= \left(\begin{array}{ccccc|ccccc|ccccc|ccccc|ccccc}
a & b & c & e & d & u & v & w & x & y & p & q & r & t & s & k & l & m & o & n & f & g & h & j & i \\
d & a & b & c & e & y & u & v & w & x & s & p & q & r & t & n & k & l & m & o & i & f & g & h & j \\
e & d & a & b & c & x & y & u & v & w & t & s & p & q & r & o & n & k & l & m & j & i & f & g & h \\
c & e & d & a & b & w & x & y & u & v & r & t & s & p & q & m & o & n & k & l & h & j & i & f & g \\
b & c & e & d & a & v & w & x & y & u & q & r & t & s & p & l & m & o & n & k & g & h & j & i & f \\
\hline
f & g & h & j & i & a & b & c & e & d & u & v & w & x & y & p & q & r & t & s & k & l & m & o & n \\
i & f & g & h & j & d & a & b & c & e & y & u & v & w & x & s & p & q & r & t & n & k & l & m & o \\
j & i & f & g & h & e & d & a & b & c & x & y & u & v & w & t & s & p & q & r & o & n & k & l & m \\
h & j & i & f & g & c & e & d & a & b & w & x & y & u & v & r & t & s & p & q & m & o & n & k & l \\
g & h & j & i & f & b & c & e & d & a & v & w & x & y & u & q & r & t & s & p & l & m & o & n & k \\
\hline
k & l & m & o & n & f & g & h & j & i & a & b & c & e & d & u & v & w & x & y & p & q & r & t & s \\
n & k & l & m & o & i & f & g & h & j & d & a & b & c & e & y & u & v & w & x & s & p & q & r & t \\
o & n & k & l & m & j & i & f & g & h & e & d & a & b & c & x & y & u & v & w & t & s & p & q & r \\
m & o & n & k & l & h & j & i & f & g & c & e & d & a & b & w & x & y & u & v & r & t & s & p & q \\
l & m & o & n & k & g & h & j & i & f & b & c & e & d & a & v & w & x & y & u & q & r & t & s & p \\
\hline
p & q & r & t & s & k & l & m & o & n & f & g & h & j & i & a & b & c & e & d & u & v & w & x & y \\
s & p & q & r & t & n & k & l & m & o & i & f & g & h & j & d & a & b & c & e & y & u & v & w & x \\
t & s & p & q & r & o & n & k & l & m & j & i & f & g & h & e & d & a & b & c & x & y & u & v & w \\
r & t & s & p & q & m & o & n & k & l & h & j & i & f & g & c & e & d & a & b & w & x & y & u & v \\
q & r & t & s & p & l & m & o & n & k & g & h & j & i & f & b & c & e & d & a & v & w & x & y & u \\
\hline
u & v & w & x & y & p & q & r & t & s & k & l & m & o & n & f & g & h & j & i & a & b & c & e & d \\
y & u & v & w & x & s & p & q & r & t & n & k & l & m & o & i & f & g & h & j & d & a & b & c & e \\
x & y & u & v & w & t & s & p & q & r & o & n & k & l & m & j & i & f & g & h & e & d & a & b & c \\
w & x & y & u & v & r & t & s & p & q & m & o & n & k & l & h & j & i & f & g & c & e & d & a & b \\
v & w & x & y & u & q & r & t & s & p & l & m & o & n & k & g & h & j & i & f & b & c & e & d & a
\end{array}\right)
$$

## A.5 The diagonalisation of a level-two circulant matrix by the level-two Fourier matrix

Formally, a level-two circulant, ${}^{[J']}[K']\mathrm{C}$, of order $[J'][K'] = [2J-1][2K-1]$ has the form, i.e., combine Eqs. 3.16 & 3.17,

$$
{}^{[J']}[K']\mathrm{C} = \sum_{j=0}^{J'-1} \left[{}^{[J']}\psi^{j}\right] \otimes \left[\sum_{i=0}^{K'-1} \mathrm{C}_{j,i} \left[{}^{[K']}\psi^{i}\right]\right] = \sum_{j=0}^{J'-1}\sum_{i=0}^{K'-1} \mathrm{C}_{j,i} \left[{}^{[J']}\psi^{j}\right] \otimes \left[{}^{[K']}\psi^{i}\right], \tag{A.5.20}
$$

and using the fact that the Fourier matrix diagonalises the downward-shift permutation matrix,

$$
\begin{aligned}
{}^{[J']}\psi^{j} &= {}^{[J']}\mathrm{F}^{\mathrm{H}} \left[{}^{[J']}\Omega^{j}\right] {}^{[J']}\mathrm{F}, \quad {}^{[J']}\Omega = diag\left(1, \omega, \omega^2, \omega^3, \ldots, \omega^{J'-1}\right), \quad \omega = e^{\frac{-2\pi\imath}{J'}}, \\
{}^{[K']}\psi^{i} &= {}^{[K']}\mathrm{F}^{\mathrm{H}} \left[{}^{[K']}\Omega^{i}\right] {}^{[K']}\mathrm{F}, \quad {}^{[K']}\Omega = diag\left(1, \omega, \omega^2, \omega^3, \ldots, \omega^{K'-1}\right), \quad \omega = e^{\frac{-2\pi\imath}{K'}},
\end{aligned}
$$

allows Eq. A.5.20 to be reformulated as,

$$
\begin{aligned}
{}^{[J']}[K']\mathrm{C} &= \sum_{j=0}^{J'-1}\sum_{i=0}^{K'-1} \mathrm{C}_{j,i} \left[\left({}^{[J']}\mathrm{F}^{\mathrm{H}} \left[{}^{[J']}\Omega^{j}\right] {}^{[J']}\mathrm{F}\right) \otimes \left({}^{[K']}\mathrm{F}^{\mathrm{H}} \left[{}^{[K']}\Omega^{i}\right] {}^{[K']}\mathrm{F}\right)\right], \\
&= \sum_{j=0}^{J'-1}\sum_{i=0}^{K'-1} \mathrm{C}_{j,i} \left({}^{[J']}\mathrm{F} \otimes {}^{[K']}\mathrm{F}\right)^{\mathrm{H}} \left(\left[{}^{[J']}\Omega^{j}\right] \otimes \left[{}^{[K']}\Omega^{i}\right]\right) \left({}^{[J']}\mathrm{F} \otimes {}^{[K']}\mathrm{F}\right),
\end{aligned}
$$

$$
\begin{aligned}
&= \left({}^{[J']}\mathrm{F} \otimes {}^{[K']}\mathrm{F}\right)^{\mathrm{H}} \left[\sum_{j=0}^{J'-1}\sum_{i=0}^{K'-1} \mathrm{C}_{j,i}\left(\left[{}^{[J']}\Omega^{j}\right] \otimes \left[{}^{[K']}\Omega^{i}\right]\right)\right] \left({}^{[J']}\mathrm{F} \otimes {}^{[K']}\mathrm{F}\right), \\
&= {}^{[J'][K']}\mathrm{F}^{\mathrm{H}} \left[{}^{[J'][K']}\Lambda\right] {}^{[J'][K']}\mathrm{F}. \qquad \text{(A.5.21)}
\end{aligned}
$$

In the above equation,

$$
\begin{aligned}
{}^{[J'][K']}\mathrm{F} &= {}^{[J']}\mathrm{F} \otimes {}^{[K']}\mathrm{F}, \\
{}^{[J'][K']}\Lambda &= \sum_{j=0}^{J'-1}\sum_{i=0}^{K'-1} \mathrm{C}_{j,i}\left(\left[{}^{[J']}\Omega^{j}\right] \otimes \left[{}^{[K']}\Omega^{i}\right]\right), \\
&= \sum_{j=0}^{J'-1}\sum_{i=0}^{K'-1} \mathrm{c}_{[jK'+i]}\left(\left[{}^{[J']}\Omega^{j}\right] \otimes \left[{}^{[K']}\Omega^{i}\right]\right), \\
&\equiv diag\left(\sqrt{J'K'}\left[{}^{[J'][K']}\mathrm{F}\right] {}^{[J'][K']}\mathrm{c}\right),
\end{aligned}
$$

where ${}^{[J'][K']}\mathrm{c}$ is the first column of the level-two circulant matrix, ${}^{[J'][K']}\mathrm{C}$, of order $[J'][K']$. Eq. A.5.21 was equivalently demonstrated [205] using the '$+\imath$' convention, $\omega = e^{\frac{+2\pi\imath}{n}}$, ${}^{[n]}\Omega = diag\left(1, \omega, \omega^2, \omega^3, \ldots, \omega^{n-1}\right)$, which implies that ${}^{[n]}\mathrm{F}^{\mathrm{H}}\left[{}^{[n]}\Omega^{i}\right]{}^{[n]}\mathrm{F} = {}^{[n]}\pi^{i}$ and thus, the diagonalisation was derived via the forward-shift permutation matrix, ${}^{[n]}\pi$, rather than the downward-shift permutation matrix, ${}^{[n]}\psi = [u_1|u_2|u_3|\ldots|u_{n-1}|u_0] = {}^{[n]}\pi^{\mathrm{T}} = [u_{n-1}|u_0|u_1|\ldots|u_{n-2}]^{\mathrm{T}}$, where $u_j$ is the $j^{th}$ column of the identity matrix ${}^{[n]}\mathbf{I}$.

## A.6 Construction of a circulant of level-three and order $[P'][J'][K']$

Formally, a level-three circulant, ${}^{[P'][J'][K']}\mathrm{C}$, of order $[P'][J'][K'] = [2P–1][2J–1][2K–1]$ has the form,

$$
{}^{[P'][J'][K']}\mathrm{C} = \sum_{k=0}^{P'-1}\left[{}^{[P']}\psi^{k}\right] \otimes \left[{}^{[J'][K']}\mathrm{C}_{k}\right], \qquad \text{(A.6.22)}
$$

where each of the ${}^{[J'][K']}\mathrm{C}_k$ is a level-two circulant of order $[J'][K']$,

$$
{}^{[J'][K']}\mathrm{C}_{k} = \sum_{j=0}^{J'-1}\left[{}^{[J']}\psi^{j}\right] \otimes \left[{}^{[K']}\mathrm{C}_{k,j}\right], \qquad \text{(A.6.23)}
$$

where each of the ${}^{[K']}\mathrm{C}_{k,j}$ is a level-one circulant of order $[K']$,

$$
{}^{[K']}\mathrm{C}_{k,j} \;=\; \sum_{i=0}^{K'-1} \mathrm{C}_{k,j,i} \left[{}^{[K']}\psi^{i}\right]. \tag{A.6.24}
$$

For $P = J = K = 3 \Rightarrow P' = J' = K' = 5$,

$$
{}^{[5][5][5]}\mathrm{C} \;=\; \sum_{k=0}^{4} \left[{}^{[5]}\psi^{k}\right] \otimes \left[{}^{[5][5]}\mathrm{C}_{k}\right], \tag{A.6.25}
$$

where each of the ${}^{[4][4]}\mathrm{C}_{k}$ is a level-two circulant of order $[5][5]$,

$$
{}^{[5][5]}\mathrm{C}_{k} \;=\; \sum_{j=0}^{4} \left[{}^{[5]}\psi^{j}\right] \otimes \left[{}^{[5]}\mathrm{C}_{k,j}\right], \tag{A.6.26}
$$

where each of the ${}^{[4]}\mathrm{C}_{k,j}$ is a level-one circulant of order $[5]$,

$$
{}^{[5]}\mathrm{C}_{k,j} \;=\; \sum_{i=0}^{4} \mathrm{C}_{k,j,i} \left[{}^{[5]}\psi^{i}\right]. \tag{A.6.27}
$$

Rather than explicitly working through the entire three levels of construction, the construction will be elaborated from the level-two system defined by Eq. A.4.19 which is identical to the system defined by Eq. A.6.26. The top level is constructed from five level-two matrices.

$$
\begin{aligned}
{}^{[5][5][5]}\mathrm{C} \;&=\; \sum_{k=0}^{4} \left[{}^{[5]}\psi^{k}\right] \otimes \left[{}^{[5][5]}\mathrm{C}_{k}\right] \\
&=\; \left[{}^{[5]}\psi^{0}\right] \otimes \left[{}^{[5][5]}\mathrm{C}_{0}\right] + \left[{}^{[5]}\psi^{1}\right] \otimes \left[{}^{[5][5]}\mathrm{C}_{1}\right] + \left[{}^{[5]}\psi^{2}\right] \otimes \left[{}^{[5][5]}\mathrm{C}_{2}\right] \\
&\qquad\qquad + \left[{}^{[5]}\psi^{3}\right] \otimes \left[{}^{[5][5]}\mathrm{C}_{3}\right] + \left[{}^{[5]}\psi^{4}\right] \otimes \left[{}^{[5][5]}\mathrm{C}_{4}\right] \\
&=\; \left[{}^{[5]}\psi^{0}\right] \otimes \begin{pmatrix} {}^{[5]}\mathrm{C}_{0,0} & {}^{[5]}\mathrm{C}_{0,4} & {}^{[5]}\mathrm{C}_{0,3} & {}^{[5]}\mathrm{C}_{0,2} & {}^{[5]}\mathrm{C}_{0,1} \\ {}^{[5]}\mathrm{C}_{0,1} & {}^{[5]}\mathrm{C}_{0,0} & {}^{[5]}\mathrm{C}_{0,4} & {}^{[5]}\mathrm{C}_{0,3} & {}^{[5]}\mathrm{C}_{0,2} \\ {}^{[5]}\mathrm{C}_{0,2} & {}^{[5]}\mathrm{C}_{0,1} & {}^{[5]}\mathrm{C}_{0,0} & {}^{[5]}\mathrm{C}_{0,4} & {}^{[5]}\mathrm{C}_{0,3} \\ {}^{[5]}\mathrm{C}_{0,3} & {}^{[5]}\mathrm{C}_{0,2} & {}^{[5]}\mathrm{C}_{0,1} & {}^{[5]}\mathrm{C}_{0,0} & {}^{[5]}\mathrm{C}_{0,4} \\ {}^{[5]}\mathrm{C}_{0,4} & {}^{[5]}\mathrm{C}_{0,3} & {}^{[5]}\mathrm{C}_{0,2} & {}^{[5]}\mathrm{C}_{0,1} & {}^{[5]}\mathrm{C}_{0,0} \end{pmatrix} + \left[{}^{[5]}\psi^{1}\right] \otimes \begin{pmatrix} {}^{[5]}\mathrm{C}_{1,0} & {}^{[5]}\mathrm{C}_{1,4} & {}^{[5]}\mathrm{C}_{1,3} & {}^{[5]}\mathrm{C}_{1,2} & {}^{[5]}\mathrm{C}_{1,1} \\ {}^{[5]}\mathrm{C}_{1,1} & {}^{[5]}\mathrm{C}_{1,0} & {}^{[5]}\mathrm{C}_{1,4} & {}^{[5]}\mathrm{C}_{1,3} & {}^{[5]}\mathrm{C}_{1,2} \\ {}^{[5]}\mathrm{C}_{1,2} & {}^{[5]}\mathrm{C}_{1,1} & {}^{[5]}\mathrm{C}_{1,0} & {}^{[5]}\mathrm{C}_{1,4} & {}^{[5]}\mathrm{C}_{1,3} \\ {}^{[5]}\mathrm{C}_{1,3} & {}^{[5]}\mathrm{C}_{1,2} & {}^{[5]}\mathrm{C}_{1,1} & {}^{[5]}\mathrm{C}_{1,0} & {}^{[5]}\mathrm{C}_{1,4} \\ {}^{[5]}\mathrm{C}_{1,4} & {}^{[5]}\mathrm{C}_{1,3} & {}^{[5]}\mathrm{C}_{1,2} & {}^{[5]}\mathrm{C}_{1,1} & {}^{[5]}\mathrm{C}_{1,0} \end{pmatrix} \\
&+\; \left[{}^{[5]}\psi^{2}\right] \otimes \begin{pmatrix} {}^{[5]}\mathrm{C}_{2,0} & {}^{[5]}\mathrm{C}_{2,4} & {}^{[5]}\mathrm{C}_{2,3} & {}^{[5]}\mathrm{C}_{2,2} & {}^{[5]}\mathrm{C}_{2,1} \\ {}^{[5]}\mathrm{C}_{2,1} & {}^{[5]}\mathrm{C}_{2,0} & {}^{[5]}\mathrm{C}_{2,4} & {}^{[5]}\mathrm{C}_{2,3} & {}^{[5]}\mathrm{C}_{2,2} \\ {}^{[5]}\mathrm{C}_{2,2} & {}^{[5]}\mathrm{C}_{2,1} & {}^{[5]}\mathrm{C}_{2,0} & {}^{[5]}\mathrm{C}_{2,4} & {}^{[5]}\mathrm{C}_{2,3} \\ {}^{[5]}\mathrm{C}_{2,3} & {}^{[5]}\mathrm{C}_{2,2} & {}^{[5]}\mathrm{C}_{2,1} & {}^{[5]}\mathrm{C}_{2,0} & {}^{[5]}\mathrm{C}_{2,4} \\ {}^{[5]}\mathrm{C}_{2,4} & {}^{[5]}\mathrm{C}_{2,3} & {}^{[5]}\mathrm{C}_{2,2} & {}^{[5]}\mathrm{C}_{2,1} & {}^{[5]}\mathrm{C}_{2,0} \end{pmatrix} + \left[{}^{[5]}\psi^{3}\right] \otimes \begin{pmatrix} {}^{[5]}\mathrm{C}_{3,0} & {}^{[5]}\mathrm{C}_{3,4} & {}^{[5]}\mathrm{C}_{3,3} & {}^{[5]}\mathrm{C}_{3,2} & {}^{[5]}\mathrm{C}_{3,1} \\ {}^{[5]}\mathrm{C}_{3,1} & {}^{[5]}\mathrm{C}_{3,0} & {}^{[5]}\mathrm{C}_{3,4} & {}^{[5]}\mathrm{C}_{3,3} & {}^{[5]}\mathrm{C}_{3,2} \\ {}^{[5]}\mathrm{C}_{3,2} & {}^{[5]}\mathrm{C}_{3,1} & {}^{[5]}\mathrm{C}_{3,0} & {}^{[5]}\mathrm{C}_{3,4} & {}^{[5]}\mathrm{C}_{3,3} \\ {}^{[5]}\mathrm{C}_{3,3} & {}^{[5]}\mathrm{C}_{3,2} & {}^{[5]}\mathrm{C}_{3,1} & {}^{[5]}\mathrm{C}_{3,0} & {}^{[5]}\mathrm{C}_{3,4} \\ {}^{[5]}\mathrm{C}_{3,4} & {}^{[5]}\mathrm{C}_{3,3} & {}^{[5]}\mathrm{C}_{3,2} & {}^{[5]}\mathrm{C}_{3,1} & {}^{[5]}\mathrm{C}_{3,0} \end{pmatrix}
\end{aligned}
$$

$$
\begin{aligned}
&+ \quad \left[{}^{[5]}\psi^4\right] \otimes \begin{pmatrix} {}^{[5]}\mathrm{C}_{4,0} & {}^{[5]}\mathrm{C}_{4,4} & {}^{[5]}\mathrm{C}_{4,3} & {}^{[5]}\mathrm{C}_{4,2} & {}^{[5]}\mathrm{C}_{4,1} \\ {}^{[5]}\mathrm{C}_{4,1} & {}^{[5]}\mathrm{C}_{4,0} & {}^{[5]}\mathrm{C}_{4,4} & {}^{[5]}\mathrm{C}_{4,3} & {}^{[5]}\mathrm{C}_{4,2} \\ {}^{[5]}\mathrm{C}_{4,2} & {}^{[5]}\mathrm{C}_{4,1} & {}^{[5]}\mathrm{C}_{4,0} & {}^{[5]}\mathrm{C}_{4,4} & {}^{[5]}\mathrm{C}_{4,3} \\ {}^{[5]}\mathrm{C}_{4,3} & {}^{[5]}\mathrm{C}_{4,2} & {}^{[5]}\mathrm{C}_{4,1} & {}^{[5]}\mathrm{C}_{4,0} & {}^{[5]}\mathrm{C}_{4,4} \\ {}^{[5]}\mathrm{C}_{4,4} & {}^{[5]}\mathrm{C}_{4,3} & {}^{[5]}\mathrm{C}_{4,2} & {}^{[5]}\mathrm{C}_{4,1} & {}^{[5]}\mathrm{C}_{4,0} \end{pmatrix} \\
&= \begin{pmatrix} {}^{[5][5]}\mathrm{C}_0 & {}^{[5][5]}\mathrm{C}_4 & {}^{[5][5]}\mathrm{C}_3 & {}^{[5][5]}\mathrm{C}_2 & {}^{[5][5]}\mathrm{C}_1 \\ {}^{[5][5]}\mathrm{C}_1 & {}^{[5][5]}\mathrm{C}_0 & {}^{[5][5]}\mathrm{C}_4 & {}^{[5][5]}\mathrm{C}_3 & {}^{[5][5]}\mathrm{C}_2 \\ {}^{[5][5]}\mathrm{C}_2 & {}^{[5][5]}\mathrm{C}_1 & {}^{[5][5]}\mathrm{C}_0 & {}^{[5][5]}\mathrm{C}_4 & {}^{[5][5]}\mathrm{C}_3 \\ {}^{[5][5]}\mathrm{C}_3 & {}^{[5][5]}\mathrm{C}_2 & {}^{[5][5]}\mathrm{C}_1 & {}^{[5][5]}\mathrm{C}_0 & {}^{[5][5]}\mathrm{C}_4 \\ {}^{[5][5]}\mathrm{C}_4 & {}^{[5][5]}\mathrm{C}_3 & {}^{[5][5]}\mathrm{C}_2 & {}^{[5][5]}\mathrm{C}_1 & {}^{[5][5]}\mathrm{C}_0 \end{pmatrix}
\end{aligned}
$$

## A.7 The diagonalisation of a level-three circulant matrix by the level-three Fourier matrix

Formally, a level-three circulant, ${}^{[P'][J'][K']}\mathrm{C}$, of order $[P'][J'][K'] = [2P–1][2J–1][2K–1]$ has the form, i.e., combine Eqs. 3.20→3.22,

$$
\begin{aligned}
{}^{[P'][J'][K']}\mathrm{C} &= \sum_{k=0}^{P'-1} \left[{}^{[P']}\psi^k\right] \otimes \left[\sum_{j=0}^{J'-1} \left[{}^{[J']}\psi^j\right] \otimes \left[\sum_{i=0}^{K'-1} \mathrm{C}_{k,j,i} \left[{}^{[K']}\psi^i\right]\right]\right], \\
&= \sum_{k=0}^{P'-1} \sum_{j=0}^{J'-1} \sum_{i=0}^{K'-1} \mathrm{C}_{k,j,i} \left[{}^{[P']}\psi^k\right] \otimes \left[{}^{[J']}\psi^j\right] \otimes \left[{}^{[K']}\psi^i\right], \qquad \text{(A.7.28)}
\end{aligned}
$$

and using the fact that the Fourier matrix diagonalises the downward-shift permutation matrix,

$$
\begin{aligned}
{}^{[P']}\psi^k &= {}^{[P']}\mathrm{F}^{\mathrm{H}} \left[{}^{[P']}\Omega^k\right] {}^{[P']}\mathrm{F}, & {}^{[P']}\Omega &= diag\left(1, \omega, \omega^2, \omega^3, \ldots, \omega^{P'-1}\right), & \omega &= e^{\frac{-2\pi\imath}{P'}}, \\
{}^{[J']}\psi^j &= {}^{[J']}\mathrm{F}^{\mathrm{H}} \left[{}^{[J']}\Omega^j\right] {}^{[J']}\mathrm{F}, & {}^{[J']}\Omega &= diag\left(1, \omega, \omega^2, \omega^3, \ldots, \omega^{J'-1}\right), & \omega &= e^{\frac{-2\pi\imath}{J'}}, \\
{}^{[K']}\psi^i &= {}^{[K']}\mathrm{F}^{\mathrm{H}} \left[{}^{[K']}\Omega^i\right] {}^{[K']}\mathrm{F} & {}^{[K']}\Omega &= diag\left(1, \omega, \omega^2, \omega^3, \ldots, \omega^{K'-1}\right), & \omega &= e^{\frac{-2\pi\imath}{K'}},
\end{aligned}
$$

allows Eq. A.7.28 to be reformulated as,

$$
\begin{aligned}
{}^{[P'][J'][K']}\mathrm{C} &= \sum_{k=0}^{P'-1} \sum_{j=0}^{J'-1} \sum_{i=0}^{K'-1} \mathrm{C}_{k,j,i} \Big[\left({}^{[P']}\mathrm{F}^{\mathrm{H}} \left[{}^{[P']}\Omega^k\right] {}^{[P']}\mathrm{F}\right) \otimes \\
&\qquad \left({}^{[J']}\mathrm{F}^{\mathrm{H}} \left[{}^{[J']}\Omega^j\right] {}^{[J']}\mathrm{F}\right) \otimes \left({}^{[K']}\mathrm{F}^{\mathrm{H}} \left[{}^{[K']}\Omega^i\right] {}^{[K']}\mathrm{F}\right)\Big], \\
&= \sum_{k=0}^{P'-1} \sum_{j=0}^{J'-1} \sum_{i=0}^{K'-1} \mathrm{C}_{k,j,i} \left({}^{[P']}\mathrm{F} \otimes {}^{[J']}\mathrm{F} \otimes {}^{[K']}\mathrm{F}\right)^{\mathrm{H}} \Big(\left[{}^{[P']}\Omega^k\right] \otimes \\
&\qquad \left[{}^{[J']}\Omega^j\right] \otimes \left[{}^{[K']}\Omega^i\right]\Big) \left({}^{[P']}\mathrm{F} \otimes {}^{[J']}\mathrm{F} \otimes {}^{[K']}\mathrm{F}\right),
\end{aligned}
$$

$$
\begin{aligned}
&= \left({}^{[P']}\mathrm{F} \otimes {}^{[J']}\mathrm{F} \otimes {}^{[K']}\mathrm{F}\right)^{\mathrm{H}} \left[\sum_{k=0}^{P'-1}\sum_{j=0}^{J'-1}\sum_{i=0}^{K'-1} \mathrm{C}_{k,j,i} \left(\left[{}^{[P']}\Omega^{k}\right] \otimes \right.\right. \\
&\qquad\qquad \left.\left.\left[{}^{[J']}\Omega^{j}\right] \otimes \left[{}^{[K']}\Omega^{i}\right]\right)\right] \left({}^{[P']}\mathrm{F} \otimes {}^{[J']}\mathrm{F} \otimes {}^{[K']}\mathrm{F}\right), \\
&= {}^{[P'][J'][K']}\mathrm{F}^{\mathrm{H}} \left[{}^{[P'][J'][K']}\Lambda\right] {}^{[P'][J'][K']}\mathrm{F}. \qquad \text{(A.7.29)}
\end{aligned}
$$

In the above equation,

$$
\begin{aligned}
{}^{[P'][J'][K']}\mathrm{F} &= {}^{[P']}\mathrm{F} \otimes {}^{[J']}\mathrm{F} \otimes {}^{[K']}\mathrm{F}, \\
{}^{[P'][J'][K']}\Lambda &= \sum_{k=0}^{P'-1}\sum_{j=0}^{J'-1}\sum_{i=0}^{K'-1} \mathrm{C}_{k,j,i} \left(\left[{}^{[P']}\Omega^{k}\right] \otimes \left[{}^{[J']}\Omega^{j}\right] \otimes \left[{}^{[K']}\Omega^{i}\right]\right), \\
&= \sum_{k=0}^{P'-1}\sum_{j=0}^{J'-1}\sum_{i=0}^{K'-1} \mathrm{c}_{[kJ'K'+jK'+i]} \left(\left[{}^{[P']}\Omega^{k}\right] \otimes \left[{}^{[J']}\Omega^{j}\right] \otimes \left[{}^{[K']}\Omega^{i}\right]\right), \\
&= diag\left(\sqrt{P'J'K'}\left[{}^{[P'][J'][K']}\mathrm{F}\right] {}^{[P'][J'][K']}\mathrm{c}\right),
\end{aligned}
$$

where ${}^{[P'][J'][K']}\mathrm{c}$ is the first column of the level-three circulant matrix, ${}^{[P'][J'][K']}\mathrm{C}$. Eq. A.7.29 was equivalently demonstrated [205] using the '$+\imath$' convention, $\omega = e^{\frac{+2\pi\imath}{n}}$, ${}^{[n]}\Omega = diag\left(1, \omega, \omega^2, \omega^3, \ldots, \omega^{n-1}\right)$, which implies that ${}^{[n]}\mathrm{F}^{\mathrm{H}}\left[{}^{[n]}\Omega^{i}\right]{}^{[n]}\mathrm{F} = {}^{[n]}\pi^{i}$, and thus, the diagonalisation was derived via the forward-shift permutation matrix, ${}^{[n]}\pi$, rather than the downward-shift permutation matrix, ${}^{[n]}\psi = [u_1|u_2|u_3|\ldots|u_{n-1}|u_0] = {}^{[n]}\pi^{\mathrm{T}} = [u_{n-1}|u_0|u_1|\ldots|u_{n-2}]^{\mathrm{T}}$, where $u_j$ is the $j^{th}$ column of the identity matrix ${}^{[n]}\mathbf{I}$.

# B

# OpenMP library routines and compiler directives

The following sections describe the various OpenMP library routines, compiler directives and optional clauses that were needed in the creation of the OpenMP version of the DFT matrix-vector product discussed in §3.6. Their descriptions have been abstracted from the Official OpenMP Specifications document [1]. §B.1 describes the library routines,

- `omp_set_num_threads`()
- `omp_get_num_threads`()
- `omp_get_thread_num`()

and §B.2 describes the compiler directives,

- **#pragma** omp parallel
- **#pragma** omp parallel **for**
- **#pragma** omp master
- **#pragma** omp critical

and optional clauses,

- `private`()
- `firstprivate`()
- `lastprivate`()
- `schedule`()
- `reduction`()

## B.1 OpenMP library routines

**void** `omp_set_num_threads`(**int** `num_threads`): The `omp_set_num_threads`() routine affects the number of threads to be used for subsequent parallel regions that do not specify a `num_threads` clause, by setting the value of the *n*threads-var internal control variable.

**int** `omp_get_num_threads`(**void**): The `omp_get_num_threads`() routine returns the number of threads in the current team. It returns the number of threads in the team executing the parallel region to which the routine region binds. If called from the sequential part of a program, this routine returns 1.

**int** `omp_get_thread_num`(**void**): The `omp_get_thread_num`() routine returns the thread number, within the team, of the thread executing the parallel region from which the routine is called. It returns the thread number of the current thread, within the team executing the parallel region to which the routine region binds. The thread number is an integer between 0 and one less than the value returned by `omp_get_num_threads`(), inclusive. The thread number of the master thread of the team is 0 and the routine returns 0 if it is called from the sequential part of a program.

## B.2 OpenMP compiler directives

**#pragma** omp parallel: This directive explicitly instructs the compiler to parallelise the chosen block of code. This is the fundamental construct that starts parallel execution. When a thread encounters a parallel construct, a team of threads is created to execute the parallel region. The thread which encountered the parallel construct becomes the master thread of the new team, with a thread number of zero for the duration of the parallel region. All threads in the new team, including the master thread, execute the region. Once the team is created, the number of threads in the team remains constant for the duration of that parallel region. Within a parallel region, thread numbers uniquely identify each thread. Thread numbers are consecutive whole numbers ranging from zero for the master thread up to one less than the number of threads within the team. A thread may obtain its own thread number by a call to the `omp_get_thread_num`() library routine. There is an implied barrier at the end of a parallel region, after which, only the master thread of the team continues execution.

**#pragma** omp parallel **for**: This directive instructs the compiler to distribute loop iterations within the team of threads that encounters this work-sharing construct. The for loop iteration variable is implicitly made private in scope for the duration of loop execution i.e. the main loop iteration variable is private by default. The loop construct specifies that the iterations of the associated loop will be executed in parallel. The iterations of the loop are distributed across threads that already exist in the team executing the parallel region to which the loop region binds.
**#pragma** omp critical: The critical construct restricts execution of the associated structured block to a single thread at a time. Region execution is restricted to a single thread at a time among all the threads in the program, without regard to the team(s) to which the threads belong.
**#pragma** omp master: The master construct specifies a structured block that is executed by the master thread of the team.

### B.2.1 Clauses

`private(list)`: The private clause declares one or more list items to be private to a thread. Each thread in the team that references a list item that appears in a private clause in any statement in the construct receives a new list item whose language-specific attributes are derived from the original list item. Inside the construct, all references to the original list item are replaced by references to the new list item. The value of the original list item is not defined upon entry to the region. The original list item must not be referenced within the region. The value of the original list item is not defined upon exit from the region.
`firstprivate(list)`: The firstprivate clause declares one or more list items to be private to a thread, and initializes each of them with the value that the corresponding original item has when the construct is encountered.
`lastprivate(list)`: The lastprivate clause declares one or more list items to be private to a thread, and causes the corresponding original list item to be updated after the end of the region.
`schedule(kind,chunk_size)`: The schedule clause specifies how iterations of the loop are divided into contiguous non-empty subsets, called chunks, and how these chunks are assigned among threads of the team. `kind` can be **static**, `dynamic`, `guided` or `runtime`. **static**: Iterations are divided into chunks of size `chunk_size`, and the chunks are statically assigned to threads in the team in a round-robin fashion in the

order of the thread number. Note that the last chunk to be assigned may have a smaller number of iterations. When no `chunk_size` is specified, the iteration space is divided into chunks which are approximately equal in size, and each thread is assigned at most one chunk. `dynamic`: Iterations are assigned to threads in chunks as the threads request them. The thread executes the chunk of iterations, then requests another chunk, until no chunks remain to be assigned. Each chunk contains `chunk_size` iterations, except for the last chunk to be assigned, which may have fewer iterations. When no `chunk_size` is specified, it defaults to 1. `guided`: Iterations are assigned to threads in chunks as the threads request them. The thread executes the chunk of iterations, then requests another chunk, until no chunks remain to be assigned. For a `chunk_size` of 1, the size of each chunk is proportional to the number of unassigned iterations divided by the number of threads, decreasing to 1. For a `chunk_size` with value k (greater than 1), the size of each chunk is determined in the same way with the restriction that the chunks do not contain fewer than k iterations (except for the last chunk to be assigned, which may have fewer than k iterations). When no `chunk_size` is specified, it defaults to 1. `runtime`: The decision regarding scheduling is deferred until run time, and the schedule and chunk size are taken from the *r*un-sched-var control variable.

`reduction`(`operator`:`list`): The reduction clause specifies an operator and one or more list items. For each list item, a private copy is created on each thread, and is initialised appropriately for the operator i.e., +(0), *(1), -(0)... After the end of the region, the original list item is updated with the values of the private copies using the specified operator.

# C
# Various iterative methods

The following sections provide the pseudocode for the custom-built implementations of various iterative schemes. They include a wide selection from the host of well-known iterative schemes. For a more detailed description of the subtleties and inner workings of the individual methods, the reader is directed to either the accompanying references or for a more general description of the various schemes, the document '*Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods*' [184]. For an introductory elucidation of the more fundamental underlying theory, the reader is advised to consider the extremely well-written, instructive and aptly named document, '*An Introduction to the Conjugate Gradient Method Without the Agonizing Pain*' [185].

The following is a list of the all the various schemes that have been included. Each is preceded by a description of the constituent vectors and variables and where relevant, a quick note on the computational omission of any superfluous storage.

- Conjugate-Gradients (CG) [182, 183, 184, 185], see §C.1.
- Conjugate-Gradients Squared (CGS) [190, 184], see §C.2.
- BiConjugate-Gradients (BiCG) [191, 184], see §C.3.
- BiConjugate-Gradients for symmetric systems: ($\text{BiCG}_{sym}$) [189], see §C.4.
- Stabilised version of the BiConjugate-Gradients (BiCGSTAB) [187], see §C.5.
- Restarted, stabilised version of the BiConjugate-Gradients (RBiCGSTAB) [192], see §C.6.

- Quasi-minimal residual with coupled two-term recurrences (QMR) [193, 194, 195], see §C.7.
- Quasi-minimal residual for symmetric systems ($\text{QMR}_{sym}$) [189], see §C.8.
- Transpose-free quasi-minimal residual (TFQMR) [196], see §C.9.
- A variant of the stabilised version of the BiConjugate-Gradients (BiCGSTAB) based on multiple Lanczos starting vectors ($\text{ML(n)BiCGSTAB}_{orig}$) [197], see §C.10.
- A variant of the stabilised version of the BiConjugate-Gradients (BiCGSTAB) based on multiple Lanczos starting vectors (ML(n)BiCGSTAB) [198] Appendix. Algorithm 2.2′ ML(n)BiCGSTAB with preconditioning, see §C.11.
- A variant of the stabilised version of the BiConjugate-Gradients (BiCGSTAB) based on multiple Lanczos starting vectors ($\text{ML(n)BiCGSTAB}_{ss}$) [198] Appendix. Algorithm 2.3′ ML(n)BiCGSTAB with preconditioning (storage-saving), see §C.12.

## C.1 Conjugate-Gradients: CG

**Complex vectors:** $r, p, z, q$
**Complex variables:** $\rho, \rho_1, \alpha, \beta$
**Real variables:** $n_0, n_1$

$$initialguess = \begin{cases} 0, & \text{Initial guess for x is zero;} \\ 1, & \text{Initial guess for x is non-zero.} \end{cases}$$

$$precond = \begin{cases} 0, & \text{No preconditioning;} \\ 1, & \text{Preconditioning.} \end{cases}$$

Computationally, the vector $q = Ap$ is not required as it can be stored in the matrix-vector product result buffer and the vector $z$ is only required if preconditioning is used.

```
1.        if(initialguess = 1){ r = b − Ax}
          else{ r = b}
          n0 = ||r||, if(n0 < breakdown){n0 = 1}
          while(iteration ≤ maximum){
2.&3.         if(precond = 1){ 2. z = M^{-1} r; 3. ρ1 = r^T z}
              else{ 3. ρ1 = r^T r}
              if(iteration = 0){
4.                if(precond = 1){ 4. p = z}
                  else{ 4. p = r}
              }
```

```
            else{
                Check for breakdown |ρ| < breakdown
5.              β = ρ1 / ρ
6.              if(precond = 1){ 6. p = z + βp}
                else{ 6. p = r + βp}
            } End if iteration
7.          q = Ap
            Check for breakdown |p^T q| < breakdown
8.          α = ρ1 / (p^T q)
9.          x = x + αp
10.         r = r − αq
11.         Check stopping criterion
                n1 = ||r||, if(n1/n0 < tolerance){Finish}
            ρ = ρ1
            iteration++
        } End
```

Adapted from [184]

## C.2 Conjugate-Gradients Squared: CGS

**Complex vectors:** $r, \tilde{r}, p, \hat{p}, u, q, \hat{q}, \hat{v}$

**Complex variables:** $\rho, \rho_1, \alpha, \beta$

**Real variables:** $n_0, n_1$

$$initialguess = \begin{cases} 0, & \text{Initial guess for x is zero;} \\ 1, & \text{Initial guess for x is non-zero.} \end{cases}$$

$$precond = \begin{cases} 0, & \text{No preconditioning;} \\ 1, & \text{Preconditioning.} \end{cases}$$

Computationally, the vectors $\hat{v} = A\hat{p}$ and $\hat{q} = Au$ are not required as they can be stored in the matrix-vector product result buffer and the vector $\hat{p}$ is only required if preconditioning is used.

```
1.          if(initialguess = 1){ r = b − Ax}
            else{ r = b}
            n0 = ||r||, if(n0 < breakdown){n0 = 1}
2.          r̃ = r
            while(iteration ≤ maximum){
3.              ρ1 = r̃^T r
4.&5.           if(iteration = 0){ 4. u = r; 5. p = u}
                else{
                    Check for breakdown |ρ| < breakdown
```

**6.** $\beta = \frac{\rho_1}{\rho}$

**7.** $u = r + \beta q$

**8.** $p = u + \beta(q + \beta p)$

} End if iteration

**9.&10.** **if**$(precond = 1)\{$ **9.** $\hat{p} = M^{-1}p$; **10.** $\hat{v} = A\hat{p}\}$

**else**$\{$ **10.** $\hat{v} = Ap\}$

Check for breakdown $|\tilde{r}^T\hat{v}| < breakdown$

**11.** $\alpha = \frac{\rho_1}{\tilde{r}^T\hat{v}}$

**12.** $q = u - \alpha\hat{v}$

**13.** **if**$(precond = 1)\{$ $u = M^{-1}(u + q)\}$

**else**$\{$ $u = u + q\}$

**14.** $x = x + \alpha u$

**15.** $\hat{q} = Au$

**16.** $r = r - \alpha\hat{q}$

**17.** Check stopping criterion

$n_1 = ||r||$, **if**$(\frac{n_1}{n_0} < tolerance)\{$**Finish**$\}$

$\rho = \rho_1$

iteration++

} End

Adapted from [184]

## C.3 BiConjugate-Gradients: BiCG

**Complex vectors:** $r, \tilde{r}, p, \tilde{p}, q, z, \tilde{z}$

**Complex variables:** $\rho, \rho_1, \alpha, \beta$

**Real variables:** $n_0, n_1$

$$initialguess = \begin{cases} 0, & \text{Initial guess for x is zero;} \\ 1, & \text{Initial guess for x is non-zero.} \end{cases}$$

$$precond = \begin{cases} 0, & \text{No preconditioning;} \\ 1, & \text{Preconditioning.} \end{cases}$$

Computationally, the vector $\tilde{q} = A^T\tilde{p}$ is not required as it can be stored in the matrix-vector product result buffer and the vectors $z$ and $\tilde{z}$ are only required if preconditioning is used.

**1.** **if**$(initialguess = 1)\{$ $r = b - Ax\}$

**else**$\{$ $r = b\}$

$n_0 = ||r||$, **if**$(n_0 < breakdown)\{n_0 = 1\}$

**2.** $\tilde{r} = r$

**while**(iteration $\leq$ maximum)$\{$

**3.**, **4.**&**5.** **if**$(precond = 1)\{$**3.** $z = M^{-1}r$; **4.** $\tilde{z} = \left[M^T\right]^{-1}\tilde{r}$; **5.** $\rho_1 = z^T\tilde{r}\}$

```
                else{ 5. ρ1 = r^T r̃}
                if(iteration = 0){
6.&7.               if(precond = 1){ 6. p = z; 7. p̃ = z̃}
                    else{ 6. p = r; 7. p̃ = r̃}
                else{
                    Check for breakdown |ρ| < breakdown
8.                  β = ρ1/ρ
9.&10.              if(precond = 1){ 9. p = z + βp; 10. p̃ = z̃ + βp̃}
                    else{ 9. p = r + βp; 10. p̃ = r̃ + βp̃}
                } End if iteration
11.             q = Ap
12.             q̃ = Ap̃
                Check for breakdown |p̃^T q| < breakdown
13.             α = ρ1/(p̃^T q)
14.             x = x + αp
15.             r = r − αq
16.             r̃ = r̃ − αq̃
17.             Check stopping criterion
                    n1 = ||r||, if(n1/n0 < tolerance){Finish}
                ρ = ρ1
                iteration++
            } End
```

Adapted from [184]

## C.4 BiConjugate-Gradients for symmetric systems: BiCG$_{sym}$

**Complex vectors:** $r, p$

**Complex variables:** $\rho, \rho_1, \alpha, \beta, \delta$

**Real variables:** $n_0, n_1$

$$initialguess = \begin{cases} 0, & \text{Initial guess for x is zero;} \\ 1, & \text{Initial guess for x is non-zero.} \end{cases}$$

$$precond = \begin{cases} 0, & \text{No preconditioning;} \\ 1, & \text{Preconditioning.} \end{cases}$$

Computationally, $Ap$ can be stored in the matrix-vector product result buffer.

```
1.      if(initialguess = 1){ r = b − Ax}
        else{ r = b}
        n0 = ||r||, if(n0 < breakdown){n0 = 1}
```

**2.** $p = r$
**3.** $\rho = r^T r$
**while**(iteration $\leq$ maximum){
**4.&5.** $\alpha = p^T A p$
Check for breakdown $|\alpha| < breakdown$
**6.** $\delta = \frac{\rho}{\alpha}$
**7.** $x = x + \delta p$
**8.** $r = r - \delta A p$
Check stopping criterion
$n_1 = ||r||$, **if**($\frac{n_1}{n_0} < tolerance$){**Finish**}
**9.** $\rho_1 = r^T r$
Check for breakdown $|\rho| < breakdown$
**10.** $\beta = \frac{\rho_1}{\rho}$
$\rho = \rho_1$
**11.** $p = r + \beta p$
iteration++
} End

## C.5 Stabilised version of the BiConjugate-Gradients: BiCGSTAB

**Complex vectors:** $r, \tilde{r}, p, \hat{p}, v, s, \hat{s}, t$
**Complex variables:** $\rho, \rho_1, \alpha, \omega, \beta$
**Real variables:** ratio, $n_0, n_1$
$initialguess = \begin{cases} 0, & \text{Initial guess for x is zero;} \\ 1, & \text{Initial guess for x is non-zero.} \end{cases}$
$precond = \begin{cases} 0, & \text{No preconditioning;} \\ 1, & \text{Preconditioning.} \end{cases}$

Computationally, the vector $t = As$ OR $t = A\hat{s}$ depending on whether preconditioning is used, is not required as it can be stored in the matrix-vector product result buffer and the vectors $\hat{p}$ and $\hat{s}$ are only required if preconditioning is used.

**1.** **if**($initialguess = 1$){ $r = b - Ax$}
**else**{ $r = b$}
$n_0 = ||r||$, **if**($n_0 < breakdown$){$n_0 = 1$}
**2.** $\tilde{r} = r$
**while**(ratio $> tolerance$ & iteration $\leq$ maximum){
**3.** $\rho_1 = \tilde{r}^T r$
**4.** **if**(iteration $= 0$){ $p = r$}

```
                    else{
                        Check for breakdown |ρ| < breakdown
5.                      β = ρ1α / ρω
6.                      p = r + β(p − ωv)
                    } End if iteration
7.&8.               if(precond = 1){ 7. p̂ = M^{-1}p; 8. v = Ap̂}
                    else{ 8. v = Ap}
                    Check for breakdown |r̃^T v| < breakdown
9.                  α = ρ1 / (r̃^T v)
10.                 s = r − αv
11.                 Check for soft breakdown ||s|| < breakdown
                        if(precond = 1){ x = x + αp̂; Finish}
                        else{ x = x + αp; Finish}
12.&13.             if(precond = 1){ 12. ŝ = M^{-1}s; 13. t = Aŝ}
                    else{ 13. t = As}
                    Check for breakdown |t^T t| < breakdown
14.                 ω = t^T s / t^T t
15.                 if(precond = 1){ x = x + αp̂ + ωŝ}
                    else{ x = x + αp + ωs}
16.                 r = s − ωt
17.                 Check stopping criterion
                        n1 = ||r||, ratio = n1/n0
                    ρ = ρ1
                    iteration++
                } End
```

Adapted from [184]

## C.6 Restarted, stabilised version of the BiConjugate-Gradients: RBiCGSTAB

**Complex vectors:** $r[restart+1], \tilde{r}, u[restart+1]$
**Complex variables:** $\rho_0, \rho, \alpha, \omega, \beta, \xi, \sigma[restart], \gamma[restart]$
$\gamma'[restart], \gamma''[restart], \tau[restart][restart]$
**Real variables:** $n_0, n_1$

```
1.      r0 = Q1(b − AQ2x)
        n0 = ||r0||, if(n0 < breakdown){n0 = 1}
2.      r̃ = r0
3.      u0 = 0
```

**4.** $\rho_0 = 1 + 0i, \alpha = 0 + 0i, \omega = 1 + 0i$
while (iteration $\leq$ maximum){
**5.** $\rho_0 = -\omega\rho_0$
for $j = 0, 1, \ldots, restart - 1$
**6.** $\rho = r_j^T \tilde{r}$
Check for breakdown $|\rho_0| < breakdown$
**7.** $\beta = \dfrac{\alpha\rho}{\rho_0}$
**8.** $\rho_0 = \rho$
**9.** $u_i = r_i - \beta u_i, i = 0, 1, \ldots, j$
**10.** $u_{j+1} = Q_1 A Q_2 u_j$
**11.** $\xi = u_{j+1}^T \tilde{r}$
Check for breakdown $|\xi| < breakdown$
**12.** $\alpha = \dfrac{\rho_0}{\xi}$
**13.** $r_i = r_i - \alpha u_{i+1}, i = 0, 1, \ldots, j$
**14.** $r_{j+1} = Q_1 A Q_2 r_j$
**15.** $x = x + \alpha u_0$
} End for j
**16.** Check stopping criterion
$n_1 = ||r||$, **if**$(\dfrac{n_1}{n_0} < tolerance)${**Finish**}
**17.** $\sigma_0 = r_1^T r_1$
Check for breakdown $|\sigma_0| < breakdown$
$\gamma_0' = \dfrac{r_0^T r_1}{\sigma_0}$
for $j = 2, 3, \ldots, restart$
Check for breakdown $|\sigma_{i-1}| < breakdown$
**18.** $\tau_{i-1,j-1} = \dfrac{r_j^T r_i}{\sigma_{i-1}}, r_j = r_j - \tau_{i-1,j-1} r_i, i = 1, 2, \ldots, j - 1$
**19.** $\sigma_{j-1} = r_j^T r_j$
Check for breakdown $|\sigma_{j-1}| < breakdown$
$\gamma_{j-1}' = \dfrac{r_0^T r_j}{\sigma_{j-1}}$
} End for j
**20.** $\gamma_{restart-1} = \omega = \gamma_{restart-1}'$
$\gamma_j = \gamma_j' - \sum_{i=j+1}^{restart} \tau_{j,i}\gamma_i, j = restart - 2 \ldots 0$
**21.** $\gamma_j'' = \gamma_{j+1} + \sum_{i=j+1}^{restart-1} \tau_{j,i}\gamma_{i+1}, j = 0 \ldots restart - 2$
**22.** $x = x + \gamma_0 r_0$
**23.** $r_0 = r_0 - \gamma_{restart-1}' r_{restart}$
**24.** $u_0 = u_0 - \gamma_{restart-1} u_{restart}$
for $j = 1, 2, \ldots, restart - 1$
**25.** $u_0 = u_0 - \gamma_{j-1} u_j$
**26.** $x = x + \gamma_{j-1}'' r_j$

**27.** $r_0 = r_0 - \gamma'_{j-1} r_j$
} End for j
iteration++
} End

Adapted from [214]

## C.7 Quasi-minimal residual with coupled two-term recurrences: QMR

**Complex vectors:** $r, d, s, p, q, \tilde{\omega}, \tilde{v}$
**Complex variables:** $\kappa, \kappa_0, \rho, \rho_0, \epsilon, \mu, \tau, \tau_0$
**Real variables:** $n_0, n_1, \gamma, \gamma_0, \xi, \xi_0, \lambda, \theta$

**1.** $\lambda = 1, \kappa = -1.0 + 0i, \theta = -1$
**2.** $\tilde{\omega} = \tilde{v} = r = Q_1(b - AQ_2x)$
$n_0 = ||r_0||$, if$(n_0 < breakdown)\{n_0 = 1\}$
**3.** $p = q = d = s = 0$
**4.** $\gamma = ||\tilde{v}||, \xi = ||\tilde{\omega}||, \rho = \tilde{\omega}^T\tilde{v}, \epsilon = (A^T\tilde{\omega})^T\tilde{v}, \mu = 0$
Check for breakdown $|\rho| < breakdown$
**5.** $\tau = \frac{\epsilon}{\rho}$
while (iteration $\leq$ maximum){
Check for breakdown $|\gamma| < breakdown$
**6.** $p = \frac{\tilde{v}}{\gamma} - \mu p$
Check for breakdown $|\xi| < breakdown$
**7.** $q = A^T\frac{\tilde{\omega}}{\xi} - \frac{\gamma\mu}{\xi}q$
**8.** $\tilde{v} = Ap - \frac{\tau}{\gamma}\tilde{v}$
**9.** $\tilde{\omega} = q - \frac{\tau}{\xi}\tilde{\omega}$
**10.** Check stopping criterion
$r = Q_1(b - AQ_2x)$
$n_1 = ||r||$, **if**$(\frac{n_1}{n_0} < tolerance)\{$**Finish**$\}$
**11.** $\gamma_0 = \gamma, \xi_0 = \xi, \rho_0 = \rho, \gamma = ||\tilde{v}||, \xi = ||\tilde{\omega}||, \rho = \tilde{\omega}^T\tilde{v}, \epsilon = (A^T\tilde{\omega})^T\tilde{v}$
Check for breakdown $|\gamma|$ OR $|\tau|$ OR $|\rho_0| < breakdown$
**12.** $\mu = \frac{\gamma_0\xi_0\rho}{\gamma\tau\rho_0}$
Check for breakdown $|\rho| < breakdown$
**13.** $\tau_0 = \tau, \tau = \frac{\epsilon}{\rho} - \gamma\mu$

Check for breakdown $|\lambda|$ OR $|\tau_0| < breakdown$

**14.** $\theta = \dfrac{|\tau_0|^2(1-\lambda)}{\lambda|\tau_0|^2 + |\gamma|^2}$

**15.** $\kappa_0 = \kappa, \kappa = \dfrac{-\gamma_0\bar{\tau}_0\kappa_0}{\lambda|\tau_0|^2 + |\gamma|^2}$

**16.** $\lambda = \dfrac{\lambda|\tau_0|^2}{\lambda|\tau_0|^2 + |\gamma|^2}$

**17.** $d = \theta d + \kappa p$

**18.** $s = \theta s + \kappa Ap$

**19.** $x = x + d$

**20.** $r = r - s$

iteration++

} End

Adapted from [214]

## C.8 Quasi-minimal residual for symmetric systems: $\text{QMR}_{sym}$

**Complex vectors:** $r, v, \tilde{v}, p, p_{old}$

**Complex variables:** $\alpha, \beta, \tau, \tilde{\tau}, s, s_{old}, \theta, \eta, \zeta, \tilde{\zeta}$

**Real variables:** $n_0, n_1, \omega, \omega_{old}, c, c_{old}$

$$initialguess = \begin{cases} 0, & \text{Initial guess for x is zero;} \\ 1, & \text{Initial guess for x is non-zero.} \end{cases}$$

$$precond = \begin{cases} 0, & \text{No preconditioning;} \\ 1, & \text{Preconditioning.} \end{cases}$$

Computationally, $Av$ can be stored in the matrix-vector product result buffer.

**1.** **if**$(initialguess = 1)\{\, r = b - Ax\}$

**else**$\{\, r = b\}$

$n_0 = ||r||$, **if**$(n_0 < breakdown)\{n_0 = 1\}$

**2.** $v = p = p_{old} = 0$

**3.** $\beta = \sqrt{\tilde{v}^T\tilde{v}} = \sqrt{r^T r}$

Check for breakdown $|\beta| < breakdown$

**4.** $\omega = \dfrac{||\tilde{v}||}{|\beta|} = \dfrac{||r||}{|\beta|}, \tilde{\tau} = \omega\beta$

**5.** $c = c_{old} = 1, s = s_{old} = 0, \omega_{old} = ||v|| = 0$

**6.** $v = \dfrac{\tilde{v}}{\beta} = \dfrac{r}{\beta}$

**while**(iteration $\leq$ maximum){

**7.** $\alpha = v^T Av$

**8.** **if**(iteration $= 0$)$\{\, \tilde{v} = Av - \alpha v\}$

**else**$\{\, \tilde{v} = Av - \alpha v - \beta v_{old}\}$

**9.** $\theta = \overline{s_{old}}\omega_{old}\beta$

**10.** $\eta = c_{old}c\omega_{old}\beta + \overline{s}\omega\alpha$

**11.** $\tilde{\zeta} = c\omega\alpha - sc_{old}\omega_{old}\beta$

**12.** $\beta = \sqrt{\tilde{v}^T\tilde{v}}$

Check for breakdown $|\beta| < breakdown$

$\omega_{old} = \omega, \omega = \dfrac{||\tilde{v}||}{|\beta|}$

**13.** $|\zeta| = \sqrt{|\tilde{\zeta}|^2 + \omega^2|\beta|^2}$

Check for breakdown $|\tilde{\zeta}| < breakdown$

**14.** $\zeta = \begin{cases} |\zeta|, & |\tilde{\zeta}| = 0; \\ \frac{|\zeta|\tilde{\zeta}}{|\tilde{\zeta}|}, & |\tilde{\zeta}| \neq 0. \end{cases}$

Check for breakdown $|\zeta| < breakdown$

**15.** $c_{old} = c, c = \tilde{\zeta}/\zeta$

**16.** $s_{old} = s, s = \omega\beta/\zeta$

**17.** **if**(iteration $= 0$){ $p = v/\zeta$}

**else if**(iteration $= 1$){ $p = (v - \eta p)/\zeta$ & set $p_{old} = p_{\text{iteration-1}}$}

**else**{ $p = (v - \eta p - \theta p_{old})/\zeta$ & set $p_{old} = p_{\text{iteration-1}}$}

**18.** $\tau = c\tilde{\tau}, \tilde{\tau} = -s\tilde{\tau}$

**19.** $x = x + \tau p$

$v_{old} = v, v = \dfrac{\tilde{v}}{\beta}$

**20.** $r = |s|^2 r + \dfrac{c\tilde{\tau}}{\omega} v$

Check stopping criterion

$n_1 = ||r||$, **if**($\dfrac{n_1}{n_0} < tolerance$){**Finish**}

iteration++

} End

## C.9 Transpose-free quasi-minimal residual: TFQMR

**Complex vectors:** $r, \tilde{r}, \omega, v, d, y_0, y_1$
**Complex variables:** $\eta, \eta_0, \rho, \rho_0, \sigma, \alpha, \beta$
**Real variables:** ratio$, n_0, n_1, \tau, \tau_0, \theta, \theta_0, c, \kappa, error$

**1.** $r = Q_1(b - AQ_2x)$

$n_0 = ||r||$, if($n_0 < breakdown$){$n_0 = 1$}

ratio $= 1, error = n_0 \times tolerance$

**2.** $\omega = y_1 = r$

**3.** $v = g = Q_1AQ_2y_1$

**4.** $d = 0$

**5.** $\tau = ||r||$

**6.** $\theta = 0, \eta = 0 + 0i$

**7.** $\tilde{r} = r$

**8.** $\rho = \tilde{r}^T r$

while (ratio > tolerance && iteration $\leq$ maximum){

**9.** $\sigma = \tilde{r}^T v$

Check for breakdown $|\sigma| < breakdown$

**10.** $\alpha = \dfrac{\rho}{\sigma}$

**11.** $y_1 = y_0 - \alpha v$

$y_0 = y_1$

for $m = 2 \times iteration - 1, \ldots, 2 \times iteration$

**12.** $\omega = \omega - \alpha Q_1 A Q_2 y_0$

$\theta_0 = \theta, \tau_0 = \tau$, Check for breakdown $|\tau_0| < breakdown$

**13.** $\theta = \dfrac{||\omega||}{\tau_0}$

**14.** $c = \dfrac{1}{\sqrt{1+\theta^2}}$

**15.** $\tau = \tau_0 \theta c$

**16.** $\eta_0 = \eta, \eta = c^2 \alpha$

**17.** $d = y_0 + \dfrac{\theta_0^2 \eta_0}{\alpha} d$

**18.** $x = x + \eta d$

**19.** $\kappa = \tau\sqrt{m+1}$

**20.** if($\kappa < error$){Check stopping criterion

$r = Q_1(b - AQ_2 x)$

$n_1 = ||r||$, ratio $= \dfrac{n_1}{n_0}$, **if**(ratio $< tolerance$){exitflag=1}

} End if

$y_0 = y_1$

} End for m

if(exitflag=1){Finish}

**21.** $\rho_0 = \rho, \rho = \tilde{r}^T \omega$

Check for breakdown $|\rho| < breakdown$

**22.** $\beta = \dfrac{\rho}{\rho_0}$

**23.** $y_1 = \omega + \beta y_0$

**24.** $v = Q_1 A Q_2 y_1 + \beta(Q_1 A Q_2 y_0 + \beta v)$

iteration++

} End

Adapted from [214]

## C.10 A variant of BiCGSTAB based on multiple Lanczos starting vectors: ML(n)BiCGSTAB$_{orig}$

**Complex vectors:** $r, g[n], \tilde{g}, d[n-1], \omega[n], q[n], u, \tilde{u}, zd, zg, z\omega$
**Complex variables:** $\alpha, \rho, c[n], \beta$
**Real variables:** $tolerance, n_0, n_1$

$$initialguess = \begin{cases} 0, & \text{Initial guess for x is zero;} \\ 1, & \text{Initial guess for x is non-zero.} \end{cases}$$

$$precond = \begin{cases} 0, & \text{No preconditioning;} \\ 1, & \text{Preconditioning.} \end{cases}$$

Computationally, the vectors $\tilde{g}$ and $\tilde{u}$ are only required if preconditioning is used.

**1.** Initialise $q$ vectors $q_i, i = 0, \ldots, n-1$ with the Mersenne Twister RNG [204] and Gram-Schmidt Orthogonalisation [183]
**2.** **if**$(initialguess = 1)\{\, r = b - Ax\}$
**else**$\{\, r = b\}$
$n_0 = ||r||$, **if**$(n_0 < breakdown)\{n_0 = 1\}$
**3.** $g_{n-1} = r$
**while**(iteration $\leq$ maximum){
**4.&5.** **if**$(precond = 1)\{$ **4.** $\tilde{g} = M^{-1} g_{n-1}$; **5.**$\omega_{n-1} = A\tilde{g}\}$
**else**$\{$ **5.**$\omega_{n-1} = A g_{n-1}\}$
**6.** $c_{n-1} = q_0^H \omega_{n-1}$
Check for breakdown $|c_{n-1}| < breakdown$
**7.** $\alpha = \dfrac{q_0^H r}{c_{n-1}}$
**8.** $u = r - \alpha\omega_{n-1}$
**if**$(precond = 1)\{$
**9.** $\tilde{u} = M^{-1} u$
Check for breakdown $||A\tilde{u}||^2 < breakdown$
**10.** $\rho = \dfrac{-[A\tilde{u}]^H u}{||A\tilde{u}||^2}$
**11.** $x = x - \rho\tilde{u} + \alpha\tilde{g}$
**12.** $r = \rho A\tilde{u} + u$
**else**{
Check for breakdown $||Au||^2 < breakdown$
**10.** $\rho = \dfrac{-[Au]^H u}{||Au||^2}$
**11.** $x = x - \rho u + \alpha g_{n-1}$
**12.** $r = \rho A u + u$
} End if
Check stopping criterion
$n_1 = ||r||$, **if**$(\dfrac{n_1}{n_0} <= tolerance)\{$**Finish**$\}$

```
        for i = 1, 2, ..., n
13.         zd = u, zg = r, zω = 0
            if(iteration > 0){
                for s = i, ..., n − 1
                    Check for breakdown |c_{s−1}| < breakdown
14.                 β = (−q_s^H zd) / c_{s−1}
15.                 zd = zd + β d_{s−1}
16.                 zg = zg + β g_{s−1}
17.                 zω = zω + β ω_{s−1}
                } End for s
            } End if iteration
            Check for breakdown |ρ c_{n−1}| < breakdown
18.         β = (−q_0^H (r + ρ z ω)) / (ρ c_{n−1})
19.         zg = zg + β g_{n−1}
20.         zω = ρ(zω + β ω_{n−1})
21.         zd = r + zω
            for s = 1, ..., i − 1
                Check for breakdown |c_{s−1}| < breakdown
22.             β = (−q_s^H zd) / c_{s−1}
23.             zd = zd + β d_{s−1}
24.             zg = zg + β g_{s−1}
            } End for s
25.         g_{i−1} = zg + zω
            if(i < n){
26.             d_{i−1} = zd − u
27.             c_{i−1} = q_i^H d_{i−1}
                Check for breakdown |c_{i−1}| < breakdown
28.             α = (q_i^H u) / c_{i−1}
29.             u = u − α d_{i−1}
                if(precond = 1){
30.                 g̃ = M^{−1} g_{i−1}
31.                 x = x + ρ α g̃
32.                 ω_{i−1} = A g̃
                else{
31.                 x = x + ρ α g_{i−1}
32.                 ω_{i−1} = A g_{i−1}
                } End if
33.             r = r − ρ α ω_{i−1}
            } End if
        } End for i
        iteration++
    } End
```

## C.11 A variant of BiCGSTAB based on multiple Lanczos starting vectors: ML(n)BiCGSTAB

**Complex vectors:** $r, q[n], g[n], \tilde{g}, u, \tilde{u}, d[n-1], \omega[n], z_d, z_g, z_\omega$
**Complex variables:** $\alpha, \rho, \beta, c[n], t_d, t_e, t_f$
**Real variables:** $tolerance, n_0, n_1$

$$initialguess = \begin{cases} 0, & \text{Initial guess for x is zero;} \\ 1, & \text{Initial guess for x is non-zero.} \end{cases}$$

$$precond = \begin{cases} 0, & \text{No preconditioning;} \\ 1, & \text{Preconditioning.} \end{cases}$$

Computationally, the vectors $\tilde{g}$ and $\tilde{u}$ are only required if preconditioning is used.

**1.** Initialise $q$ vectors $q_i, i = 0, \ldots, n-1$ with the Mersenne Twister RNG [204] and Gram-Schmidt Orthogonalisation [183]
**2.** **if**$(initialguess = 1)\{\, r = b - Ax\}$
**else**$\{\, r = b\}$
$n_0 = ||r||$, **if**$(n_0 < breakdown)\{n_0 = 1\}$
**3.** $g_{n-1} = r$
**3.&5.** **if**$(precond = 1)\{$ **4.** $\tilde{g} = M^{-1}r$, **5.**$\omega_{n-1} = A\tilde{g}\}$
**else**$\{$ **5.** $\omega_{n-1} = Ar\}$
**6.** $c_{n-1} = q_0^H \omega_{n-1}$
$t_e = q_0^H r$
**while** (iteration $\leq$ maximum){
Check for breakdown $|c_{n-1}| < breakdown$
**7.** $\alpha = \dfrac{q_0^H r}{c_{n-1}} = \dfrac{t_e}{c_{n-1}}$
**8.** $u = r - \alpha\omega_{n-1}$
**if**$(precond = 1)\{$
**9.** $\tilde{u} = M^{-1}u$
Check for breakdown $||A\tilde{u}||^2 < breakdown$
**10.** $\rho = \dfrac{-(A\tilde{u})^H u}{||A\tilde{u}||^2}$
**11.** $x = x - \rho\tilde{u} + \alpha\tilde{g}$
**12.** $r = \rho A\tilde{u} + u$
**else**{
Check for breakdown $||Au||^2 < breakdown$
**10.** $\rho = \dfrac{-(Au)^H u}{||Au||^2}$
**11.** $x = x - \rho u + \alpha z_g$
**12.** $r = \rho Au + u$
}
Check stopping criterion
$n_1 = ||r||$, **if**$(\dfrac{n_1}{n_0} <= tolerance)\{$**Finish**$\}$

$t_d = \rho c_{n-1}$
**for** $i = 1, 2, \ldots, n-1$
**13.** $z_d = u, z_g = 0, z_\omega = 0$
$t_f = q_i^H z_d$
**if**(iteration $> 0$){
Check for breakdown $|c_{i-1}| < breakdown$
$\beta = \dfrac{-q_i^H z_d}{c_{i-1}} = \dfrac{-t_f}{c_{i-1}}$
$z_d = z_d + \beta d_{i-1}$
$z_g = z_g + \beta g_{i-1}$
$z_\omega = z_\omega + \beta \omega_{i-1}$
**for** $s = i+1, \ldots, n-1$
Check for breakdown $|c_{s-1}| < breakdown$
**14.** $\beta = \dfrac{-q_s^H z_d}{c_{s-1}}$
**15.** $z_d = z_d + \beta d_{s-1}$
**16.** $z_g = z_g + \beta g_{s-1}$
**17.** $z_\omega = z_\omega + \beta \omega_{s-1}$
} End for s
} End if
**18.** $z_\omega = r + \rho z_\omega$
**19.** $\beta = \dfrac{-q_0^H z_\omega}{\rho c_{n-1}} = \dfrac{-q_0^H z_\omega}{t_d}$
**20.** $z_\omega = z_\omega + \rho \beta \omega_{n-1}$
**21.** $z_g = z_g + z_\omega + \beta g_{n-1}$
**for** $s = 1, \ldots, i-1$
Check for breakdown $|c_{s-1}| < breakdown$
**22.** $\beta = \dfrac{-q_s^H z_\omega}{c_{s-1}}$
**23.** $z_g = z_g + \beta g_{s-1}$
**24.** $z_\omega = z_\omega + \beta d_{s-1}$
} End for s
**25.** $d_{i-1} = z_\omega - u$
**26.** $c_{i-1} = q_i^H d_{i-1}$
**27.** $g_{i-1} = z_g$
**28.&29.** **if**($precond = 1$){ **28.** $\tilde{g} = M^{-1} z_g$ **29.** $\omega_{i-1} = A\tilde{g}$}
**else**{ **29.** $\omega_{i-1} = A z_g$}
Check for breakdown $|c_{i-1}| < breakdown$
**30.** $\alpha = \dfrac{q_i^H u}{c_{i-1}} = \dfrac{t_f}{c_{i-1}}$
**if**(i $< n-1$){
**31.** $u = u - \alpha d_{i-1}$
} End if
**32.** **if**($precond = 1$){ **32.** $x = x + \rho\alpha\tilde{g}$}
**else**{ **32.** $x = x + \rho\alpha z_g$}
**33.** $r = r - \rho\alpha\omega_{i-1}$
} End for i
$t_e = q_0^H r$

Check for breakdown $|\rho c_{n-1}| < breakdown$

**34.** $\beta = \dfrac{-q_0^H r}{\rho c_{n-1}} = \dfrac{-t_e}{t_d}$

**35.** $z_\omega = r + \rho\beta\omega_{n-1}$

**36.** $z_g = z_\omega + \beta g_{n-1}$

**for** $s = 1, \ldots, n-1$

Check for breakdown $|c_{s-1}| < breakdown$

**37.** $\beta = \dfrac{-q_s^H z_\omega}{c_{s-1}}$

**38.** $z_g = z_g + \beta g_{s-1}$

**39.** $z_\omega = z_\omega + \beta d_{s-1}$

} End for s

**40.** $g_{n-1} = z_g$

**41.&42.** **if**$(precond = 1)\{$ **41.** $\tilde{g} = M^{-1} z_g$ **42.** $\omega_{n-1} = A\tilde{g}\}$

**else**$\{$ **42.** $\omega_{n-1} = Az_g\}$

**43.** $c_{n-1} = q_0^H \omega_{n-1}$

iteration++

} End

## C.12 A variant of BiCGSTAB based on multiple Lanczos starting vectors: ML(n)BiCGSTAB$_{ss}$

**Complex vectors:** $r, q[n], g[n], \omega[n], z_g, z_\omega$

**Complex variables:** $\alpha, \rho, \beta, c[n], t_e$

**Real variables:** $tolerance, n_0, n_1$

$$initialguess = \begin{cases} 0, & \text{Initial guess for x is zero;} \\ 1, & \text{Initial guess for x is non-zero.} \end{cases}$$

$$precond = \begin{cases} 0, & \text{No preconditioning;} \\ 1, & \text{Preconditioning.} \end{cases}$$

**1.** Initialise $q$ vectors $q_i, i = 0, \ldots, n-1$ with the Mersenne Twister RNG [204] and Gram-Schmidt Orthogonalisation [183]

**2.** **if**$(initialguess = 1)\{ r = b - Ax\}$

**else**$\{ r = b\}$

$n_0 = ||r||$, **if**$(n_0 < breakdown)\{n_0 = 1\}$

**3.&4.** **if**$(precond = 1)\{$ **3.** $g_0 = M^{-1} r$; **4.**$\omega_{n-1} = Ag_0\}$

**else**$\{$ **3.** $g_0 = r$; **4.**$\omega_{n-1} = Ag_0\}$

**5.** $c_0 = q_0^H \omega_0$

$t_e = q_0^H r$

**while** (iteration $\leq$ maximum){
  **for** $i = 1, 2, \ldots, n-1$
    Check for breakdown $|c_{i-1}| < breakdown$
**6.** $\alpha = \frac{q_{i-1}^H r}{c_{i-1}} = \frac{t_e}{c_{i-1}}$
**7.** $x = x + \alpha g_{i-1}$
**8.** $r = r - \alpha \omega_{i-1}$
**9.** $z_\omega = r, z_g = 0$
    $t_e = q_i^H z_\omega$
    **if**(iteration $> 0$){
      Check for breakdown $|c_i| < breakdown$
      $\beta = \frac{-q_i^H z_\omega}{c_i} = \frac{-t_e}{c_i}$
      $z_\omega = z_\omega + \rho\beta\omega_i$
      $z_g = z_g + \beta g_i$
      **for** $s = i+1, \ldots, n-1$
        Check for breakdown $|c_s| < breakdown$
**10.** $\beta = \frac{-q_s^H z_\omega}{c_s} = \frac{-t_e}{c_s}$
**11.** $z_\omega = z_\omega + \rho\beta\omega_s$
**12.** $z_g = z_g + \beta g_s$
      } End for s
    } End if iteration
**13.** **if**($precond = 1$){ **13.** $z_g = z_g + M^{-1} z_\omega$}
    **else**{ **13.**$z_g = z_g + z_\omega$}
**14.** $z_\omega = A z_g$
    **for** $s = 0, \ldots, i-1$
      Check for breakdown $|c_s| < breakdown$
**15.** $\beta = \frac{-q_s^H z_\omega}{c_s}$
**16.** $z_\omega = z_\omega + \beta\omega_s$
**17.** $z_g = z_g + \beta g_s$
    } End for s
**18.** $g_i = z_g$
**19.** $\omega_i = z_\omega$
**20.** $c_i = q_i^H z_\omega$
  } End for i
  Check for breakdown $|c_{n-1}| < breakdown$
**21.** $\alpha = \frac{q_{n-1}^H r}{c_{n-1}} = \frac{t_e}{c_{n-1}}$
**22.** $x = x + \alpha g_{n-1}$
**23.** $r = r - \alpha \omega_{n-1}$
  Check stopping criterion
    $n_1 = ||r||$, **if**($\frac{n_1}{n_0} <= tolerance$){**Finish**}
  **if**($precond = 1$){
**24.** $z_g = M^{-1} r$
    Check for breakdown $||Az_g||^2 < breakdown$
**25.** $\rho = \frac{-(Az_g)^H r}{||Az_g||^2}$

**26.** $x = x - \rho z_g$

**27.** $r = \rho A z_g + r$

**else**{

Check for breakdown $||Ar||^2 < breakdown$

**25.** $\rho = \dfrac{-(Ar)^H r}{||Ar||^2}$

**26.** $x = x - \rho r$

**27.** $r = \rho A r + r$

} End if

**28.** $z_\omega = r, z_g = 0$

$t_e = q_0^H r$

$c = \rho c$

Check for breakdown $|\rho c_0| < breakdown$

$\beta = \dfrac{-q_0^H z_\omega}{\rho c_0} \dfrac{-t_e}{\rho c_0}$

$z_\omega = z_\omega + \rho\beta\omega_0$

$z_g = z_g + \beta g_0$

**for** $s = 1, \ldots, n-1$

Check for breakdown $|\rho c_s| < breakdown$

**29.** $\beta = \dfrac{-q_s^H z_\omega}{\rho c_s}$

**30.** $z_\omega = z_\omega + \rho\beta\omega_s$

**31.** $z_g = z_g + \beta g_s$

} End for s

**32.** **if**$(precond = 1)${ **32.** $z_g = z_g + M^{-1} z_\omega$}

**else**{ **32.**$z_g = z_g + z_\omega$}

**33.** $z_\omega = A z_g$

**34.** $c_0 = q_0^H z_\omega$

**35.** $g_0 = z_g, \omega_0 = z_\omega$

iteration++

} End

# D

# Full system performance tables

The appendix lists the results for the benchmarks relating to the performance of the MPI-based distributed-memory implementation of the OpenDDA framework, see §6.3. Due to that amount of associated timing data, each of the constituent components has been assigned a short identifying tag, as listed below.

- **SV**: Number of staring vectors
- **NI**: Number of iterations to converge to the prescribed tolerance
- **CT**: Time to converge to the prescribed tolerance
- **SU**: For Fig. D.1, this refers to the associated MPI speedup obtained with respect to the serial results in Fig. 6.15, using the 4 processors on NEMO
- **MVPs**: Number of matrix-vector products required by the iterative scheme during convergence
- **MVP**: Average time taken to perform a matrix-vector product
- **MVP BREAKDOWN**: A breakdown of the average time taken by each component of the matrix-vector product
- **VI**: Average time taken to initialise the zero-padded vector components
- **TDFT**: Average time taken to perform the DFT of the six independent tensor components
- **TFTR**: Average time taken to perform the forward distributed transpose required by the DFT of the six independent tensor components
- **CORE**: Average time taken to perform the DFT of the three zero-padded vector components, the element-wise FD multiplication of the tensor & vector components and the iDFT of the resultant
- **VFTR**: Average time taken to perform the forward distributed transpose required by the DFT of the three zero-padded vector components
- **VRTR**: Average time taken to perform the reverse distributed transpose required by the iDFT of the resultant of the FD element-wise multiplication of the tensor & vector components
- **NDDC**: Average time taken to normalise the resultant, add the diagonal contribution and compress the resultant to only include entries corresponding to occupied sites
- **Memory**: Memory required to perform the benchmark

| Scheme | TR | SV | NI | CT (s) | SU | MVPs | MVP (s) | VI (s) | TDFT (s) | TFTR (s) | CORE (s) | VFTR (s) | VRTR (s) | NDDC (s) | Memory Total (GB) | Memory Node (GB) | TR | SV | NI | CT (s) | SU | MVPs | MVP (s) | VI (s) | TDFT (s) | TFTR (s) | CORE (s) | VFTR (s) | VRTR (s) | NDDC (s) | Memory Total (GB) | Memory Node (GB) |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| bicg | **0** | - | 21 | 604.432 | 2.382 | 42 | 14.022 | 0.164 | 4.391 | 4.821 | 8.954 | 2.283 | 2.246 | 0.153 | 3.879 | 0.974 | **3** | - | 21 | 603.749 | 2.385 | 42 | 14.006 | 0.165 | 4.357 | 4.800 | 8.865 | 2.275 | 2.326 | 0.155 | 3.879 | 0.974 |
| bicg$_{sym}$ | | - | 21 | 306.181 | 2.422 | 21 | 14.142 | 0.170 | 4.361 | 4.765 | 8.823 | 2.249 | 2.311 | 0.153 | 3.318 | 0.833 | | - | 21 | 322.652 | 2.298 | 21 | 14.909 | 0.175 | 4.394 | 4.857 | 9.146 | 2.456 | 2.533 | 0.156 | 3.318 | 0.833 |
| bicgstab | | - | 12 | 353.834 | 2.393 | 24 | 14.282 | 0.170 | 4.385 | 5.040 | 8.871 | 2.317 | 2.378 | 0.153 | 3.879 | 0.974 | | - | 12 | 354.898 | 2.385 | 24 | 14.323 | 0.169 | 4.368 | 4.838 | 8.967 | 2.270 | 2.377 | 0.154 | 3.879 | 0.974 |
| cg | | - | 21 | 320.919 | 2.326 | 21 | 14.846 | 0.170 | 4.381 | 5.049 | 8.964 | 2.348 | 2.627 | 0.155 | 3.318 | 0.833 | | - | 21 | 316.658 | 2.358 | 21 | 14.645 | 0.171 | 4.368 | 4.756 | 8.954 | 2.404 | 2.490 | 0.152 | 3.318 | 0.833 |
| cgs | | - | 13 | 382.936 | 2.380 | 26 | 14.289 | 0.168 | 4.339 | 4.754 | 8.936 | 2.350 | 2.331 | 0.152 | 3.879 | 0.974 | | - | 13 | 385.489 | 2.364 | 26 | 14.389 | 0.169 | 4.356 | 4.752 | 9.053 | 2.278 | 2.377 | 0.152 | 3.879 | 0.974 |
| mlbicgstab | | 1 | 11 | 332.864 | 2.609 | 22 | 14.402 | 0.169 | 4.364 | 4.813 | 8.925 | 2.347 | 2.403 | 0.150 | 4.816 | 1.208 | | 1 | 11 | 341.875 | 2.541 | 22 | 14.896 | 0.176 | 4.359 | 4.884 | 9.173 | 2.444 | 2.498 | 0.156 | 4.816 | 1.208 |
| mlbicgstab | | 2 | 8 | 355.616 | 2.339 | 23 | 14.441 | 0.169 | 4.396 | 5.145 | 8.817 | 2.393 | 2.436 | 0.155 | 5.565 | 1.395 | | 2 | 8 | 376.232 | 2.211 | 23 | 14.491 | 0.169 | 4.362 | 4.913 | 8.876 | 2.395 | 2.528 | 0.155 | 5.565 | 1.395 |
| mlbicgstab | | 3 | 6 | 341.242 | 2.396 | 22 | 14.386 | 0.169 | 4.410 | 5.004 | 8.868 | 2.310 | 2.444 | 0.154 | 6.314 | 1.583 | | 3 | 6 | 339.511 | 2.409 | 22 | 14.319 | 0.170 | 4.370 | 4.896 | 8.799 | 2.393 | 2.381 | 0.155 | 6.314 | 1.583 |
| mlbicgstab$_{orig}$ | | 1 | 12 | 363.759 | 2.371 | 24 | 14.425 | 0.169 | 4.356 | 5.105 | 8.883 | 2.352 | 2.411 | 0.153 | 4.816 | 1.208 | | 1 | 11 | 332.623 | 2.593 | 22 | 14.378 | 0.170 | 4.360 | 5.015 | 8.886 | 2.316 | 2.424 | 0.151 | 4.816 | 1.208 |
| mlbicgstab$_{orig}$ | | 2 | 8 | 358.343 | 2.368 | 23 | 14.567 | 0.170 | 4.356 | 5.089 | 8.937 | 2.282 | 2.511 | 0.152 | 5.565 | 1.395 | | 2 | 8 | 359.167 | 2.362 | 23 | 14.600 | 0.169 | 4.366 | 5.052 | 8.985 | 2.382 | 2.435 | 0.155 | 5.565 | 1.395 |
| mlbicgstab$_{orig}$ | | 3 | 6 | 352.316 | 2.309 | 22 | 14.381 | 0.168 | 4.392 | 4.824 | 8.943 | 2.323 | 2.430 | 0.153 | 6.314 | 1.583 | | 3 | 6 | 346.059 | 2.350 | 22 | 14.040 | 0.169 | 4.363 | 4.725 | 8.993 | 2.219 | 2.155 | 0.153 | 6.314 | 1.583 |
| mlbicgstab$_{ss}$ | | 1 | 12 | 341.461 | 2.448 | 23 | 14.231 | 0.170 | 4.374 | 4.837 | 8.898 | 2.263 | 2.349 | 0.150 | 4.067 | 1.021 | | 1 | 12 | 347.398 | 2.406 | 23 | 14.482 | 0.170 | 4.395 | 4.848 | 8.866 | 2.380 | 2.515 | 0.150 | 4.067 | 1.021 |
| mlbicgstab$_{ss}$ | | 2 | 8 | 347.064 | 2.339 | 23 | 14.273 | 0.170 | 4.356 | 4.927 | 9.015 | 2.272 | 2.260 | 0.152 | 4.629 | 1.161 | | 2 | 8 | 368.816 | 2.201 | 23 | 15.120 | 0.172 | 4.363 | 4.941 | 9.214 | 2.418 | 2.664 | 0.156 | 4.629 | 1.161 |
| mlbicgstab$_{ss}$ | | 3 | 6 | 369.568 | 2.281 | 23 | 14.987 | 0.175 | 4.364 | 5.029 | 9.097 | 2.470 | 2.656 | 0.156 | 5.190 | 1.302 | | 3 | 6 | 353.170 | 2.387 | 23 | 14.334 | 0.170 | 4.356 | 4.864 | 8.939 | 2.322 | 2.350 | 0.152 | 5.190 | 1.302 |
| qmr | | - | 22 | 1636.350 | 2.264 | 109 | 14.747 | 0.174 | 4.354 | 4.790 | 9.125 | 2.416 | 2.700 | 0.162 | 4.254 | 1.068 | | - | 22 | 1651.395 | 2.243 | 109 | 14.892 | 0.167 | 4.378 | 5.030 | 9.137 | 2.489 | 2.816 | 0.159 | 4.254 | 1.068 |
| qmr$_{sym}$ | | - | 21 | 320.710 | 2.369 | 21 | 14.632 | 0.170 | 4.374 | 4.941 | 9.051 | 2.404 | 2.399 | 0.156 | 3.879 | 0.974 | | - | 21 | 317.597 | 2.392 | 21 | 14.485 | 0.171 | 4.372 | 4.855 | 8.916 | 2.322 | 2.389 | 0.156 | 3.879 | 0.974 |
| rbicgstab | | 1 | 14 | 455.986 | 2.185 | 28 | 15.704 | 0.189 | 4.367 | 4.909 | 9.524 | 2.549 | 2.757 | 0.168 | 3.879 | 0.974 | | 1 | 14 | 418.246 | 2.382 | 28 | 14.450 | 0.168 | 4.373 | 5.106 | 8.918 | 2.393 | 2.455 | 0.156 | 3.879 | 0.974 |
| rbicgstab | | 2 | 7 | 415.454 | 2.384 | 28 | 14.308 | 0.170 | 4.367 | 5.142 | 8.931 | 2.260 | 2.451 | 0.158 | 4.254 | 1.068 | | 2 | 7 | 419.620 | 2.360 | 28 | 14.462 | 0.168 | 4.366 | 4.726 | 8.948 | 2.387 | 2.477 | 0.156 | 4.254 | 1.068 |
| rbicgstab | | 3 | 4 | 357.698 | 2.379 | 24 | 14.336 | 0.169 | 4.394 | 4.787 | 8.874 | 2.294 | 2.318 | 0.155 | 4.629 | 1.161 | | 3 | 4 | 359.119 | 2.370 | 24 | 14.389 | 0.170 | 4.382 | 4.958 | 8.977 | 2.354 | 2.342 | 0.156 | 4.629 | 1.161 |
| tfqmr | | - | 13 | 795.031 | 2.327 | 53 | 14.630 | 0.172 | 4.367 | 5.180 | 9.037 | 2.426 | 2.654 | 0.159 | 4.254 | 1.068 | | - | 13 | 774.019 | 2.391 | 53 | 14.264 | 0.165 | 4.375 | 4.874 | 8.924 | 2.392 | 2.406 | 0.155 | 4.254 | 1.068 |
| bicg | **2** | - | 21 | 648.586 | 2.220 | 42 | 15.044 | 0.181 | 4.377 | 4.785 | 9.332 | 2.458 | 2.690 | 0.165 | 3.883 | 0.975 | **5** | - | 21 | 630.142 | 2.285 | 42 | 14.565 | 0.163 | 4.372 | 5.840 | 9.122 | 2.301 | 2.476 | 0.148 | 3.899 | 0.978 |
| bicg$_{sym}$ | | - | 21 | 310.123 | 2.391 | 21 | 14.328 | 0.170 | 4.376 | 4.740 | 8.767 | 2.266 | 2.382 | 0.151 | 3.321 | 0.834 | | - | 21 | 305.255 | 2.429 | 21 | 13.952 | 0.171 | 4.398 | 5.207 | 8.819 | 2.170 | 2.194 | 0.139 | 3.338 | 0.837 |
| bicgstab | | - | 12 | 358.432 | 2.362 | 24 | 14.475 | 0.169 | 4.372 | 4.976 | 8.995 | 2.391 | 2.372 | 0.153 | 3.883 | 0.975 | | - | 12 | 357.602 | 2.367 | 24 | 14.274 | 0.170 | 4.403 | 5.328 | 8.982 | 2.173 | 2.338 | 0.139 | 3.899 | 0.978 |
| cg | | - | 21 | 314.282 | 2.375 | 21 | 14.524 | 0.170 | 4.364 | 5.054 | 8.894 | 2.421 | 2.388 | 0.155 | 3.321 | 0.834 | | - | 21 | 312.366 | 2.390 | 21 | 14.289 | 0.173 | 4.404 | 5.434 | 9.017 | 2.215 | 2.247 | 0.144 | 3.338 | 0.837 |
| cgs | | - | 13 | 384.278 | 2.371 | 26 | 14.346 | 0.169 | 4.377 | 5.155 | 8.918 | 2.311 | 2.426 | 0.154 | 3.883 | 0.975 | | - | 13 | 386.656 | 2.357 | 26 | 14.338 | 0.169 | 4.389 | 6.440 | 8.934 | 2.288 | 2.383 | 0.144 | 3.899 | 0.978 |
| mlbicgstab | | 1 | 11 | 340.525 | 2.551 | 22 | 14.462 | 0.169 | 4.380 | 5.151 | 9.029 | 2.338 | 2.366 | 0.151 | 4.819 | 1.209 | | 1 | 12 | 354.524 | 2.450 | 24 | 14.050 | 0.169 | 4.364 | 5.250 | 8.893 | 2.126 | 2.194 | 0.143 | 4.836 | 1.212 |
| mlbicgstab | | 2 | 8 | 343.607 | 2.421 | 23 | 14.072 | 0.169 | 4.371 | 4.760 | 8.825 | 2.252 | 2.266 | 0.154 | 5.568 | 1.396 | | 2 | 8 | 345.021 | 2.411 | 23 | 14.011 | 0.169 | 4.391 | 5.238 | 8.748 | 2.180 | 2.309 | 0.145 | 5.585 | 1.399 |
| mlbicgstab | | 3 | 7 | 399.705 | 2.046 | 26 | 14.247 | 0.169 | 4.387 | 4.843 | 8.954 | 2.233 | 2.368 | 0.155 | 6.317 | 1.583 | | 3 | 6 | 358.996 | 2.278 | 22 | 14.408 | 0.169 | 4.409 | 5.306 | 9.249 | 2.105 | 2.316 | 0.144 | 6.334 | 1.586 |
| mlbicgstab$_{orig}$ | | 1 | 12 | 362.714 | 2.378 | 24 | 14.384 | 0.169 | 4.408 | 4.746 | 8.869 | 2.247 | 2.388 | 0.152 | 4.819 | 1.209 | | 1 | 12 | 365.706 | 2.359 | 24 | 14.406 | 0.170 | 4.369 | 5.247 | 8.966 | 2.144 | 2.363 | 0.144 | 4.836 | 1.212 |
| mlbicgstab$_{orig}$ | | 2 | 8 | 350.947 | 2.418 | 23 | 14.241 | 0.170 | 4.384 | 5.246 | 8.882 | 2.221 | 2.396 | 0.153 | 5.568 | 1.396 | | 2 | 8 | 359.094 | 2.363 | 23 | 14.513 | 0.170 | 4.393 | 5.243 | 9.018 | 2.262 | 2.375 | 0.148 | 5.585 | 1.399 |
| mlbicgstab$_{orig}$ | | 3 | 6 | 354.198 | 2.296 | 22 | 14.389 | 0.169 | 4.359 | 4.827 | 8.952 | 2.339 | 2.414 | 0.155 | 6.317 | 1.583 | | 3 | 7 | 422.355 | 1.926 | 26 | 14.898 | 0.169 | 4.406 | 5.489 | 9.164 | 2.305 | 2.700 | 0.150 | 6.334 | 1.586 |
| mlbicgstab$_{ss}$ | | 1 | 12 | 366.039 | 2.284 | 23 | 14.993 | 0.177 | 4.380 | 4.745 | 9.256 | 2.400 | 2.638 | 0.155 | 4.070 | 1.021 | | 1 | 12 | 345.953 | 2.417 | 23 | 14.271 | 0.170 | 4.396 | 5.627 | 8.880 | 2.187 | 2.289 | 0.143 | 4.087 | 1.024 |
| mlbicgstab$_{ss}$ | | 2 | 8 | 346.735 | 2.341 | 23 | 14.257 | 0.170 | 4.378 | 4.774 | 8.980 | 2.275 | 2.283 | 0.151 | 4.632 | 1.162 | | 2 | 8 | 345.924 | 2.346 | 23 | 14.086 | 0.169 | 4.388 | 5.352 | 8.889 | 2.143 | 2.294 | 0.143 | 4.648 | 1.165 |
| mlbicgstab$_{ss}$ | | 3 | 6 | 354.648 | 2.377 | 23 | 14.404 | 0.170 | 4.356 | 4.963 | 8.864 | 2.364 | 2.441 | 0.153 | 5.194 | 1.302 | | 3 | 6 | 350.607 | 2.404 | 23 | 14.076 | 0.170 | 4.369 | 5.220 | 8.901 | 2.167 | 2.251 | 0.142 | 5.210 | 1.305 |
| qmr | | - | 22 | 1645.892 | 2.251 | 109 | 14.830 | 0.173 | 4.375 | 4.933 | 9.384 | 2.428 | 2.593 | 0.162 | 4.257 | 1.068 | | - | 22 | 1608.394 | 2.303 | 109 | 14.447 | 0.166 | 4.382 | 5.265 | 9.130 | 2.306 | 2.596 | 0.152 | 4.274 | 1.071 |
| qmr$_{sym}$ | | - | 21 | 338.905 | 2.241 | 21 | 15.388 | 0.189 | 4.366 | 5.024 | 9.250 | 2.426 | 2.746 | 0.157 | 3.883 | 0.975 | | - | 21 | 317.950 | 2.389 | 21 | 14.400 | 0.171 | 4.390 | 5.331 | 8.855 | 2.170 | 2.369 | 0.142 | 3.899 | 0.978 |
| rbicgstab | | 1 | 14 | 407.339 | 2.446 | 28 | 14.075 | 0.169 | 4.373 | 4.918 | 8.892 | 2.265 | 2.240 | 0.156 | 3.883 | 0.975 | | 1 | 14 | 417.666 | 2.385 | 28 | 14.347 | 0.169 | 4.390 | 5.648 | 8.891 | 2.203 | 2.402 | 0.147 | 3.899 | 0.978 |
| rbicgstab | | 2 | 7 | 424.318 | 2.334 | 28 | 14.635 | 0.168 | 4.371 | 5.042 | 9.069 | 2.265 | 2.488 | 0.156 | 4.257 | 1.068 | | 2 | 7 | 405.125 | 2.445 | 28 | 13.812 | 0.168 | 4.393 | 5.206 | 8.888 | 2.100 | 2.168 | 0.141 | 4.274 | 1.071 |
| rbicgstab | | 3 | 4 | 364.713 | 2.334 | 24 | 14.630 | 0.170 | 4.370 | 4.851 | 8.931 | 2.352 | 2.475 | 0.157 | 4.632 | 1.162 | | 3 | 4 | 352.124 | 2.417 | 24 | 13.995 | 0.170 | 4.387 | 5.230 | 8.856 | 2.154 | 2.166 | 0.143 | 4.648 | 1.165 |
| tfqmr | | - | 13 | 784.391 | 2.359 | 53 | 14.428 | 0.170 | 4.363 | 5.087 | 9.039 | 2.346 | 2.533 | 0.158 | 4.257 | 1.068 | | - | 13 | 765.300 | 2.418 | 53 | 13.999 | 0.163 | 4.398 | 5.471 | 8.935 | 2.205 | 2.356 | 0.145 | 4.274 | 1.071 |

Figure D.1: The performance, using 4 processors on NEMO, of the full MPI-based distributed-memory system as a function of the distributed-transpose algorithm employed to support the DFT kernel, the iterative scheme, and where applicable, the number of starting vectors.

| Scheme | TR | SV | NI | CT (s) | MVPs | MVP (s) | VI (s) | TDFT (s) | TFTR (s) | CORE (s) | VFTR (s) | VRTR (s) | NDDC (s) | Memory Total (GB) | Memory Node (GB) | TR | SV | NI | CT (s) | MVPs | MVP (s) | VI (s) | TDFT (s) | TFTR (s) | CORE (s) | VFTR (s) | VRTR (s) | NDDC (s) | Memory Total (GB) | Memory Node (GB) |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| bicg | **0** | - | 21 | 1500.496 | 42 | 35.294 | 0.255 | 7.244 | 18.172 | 14.869 | 9.506 | 9.831 | 0.228 | 6.690 | 1.678 | **3** | - | 21 | 1133.539 | 42 | 26.548 | 0.254 | 7.239 | 10.069 | 15.150 | 5.191 | 5.280 | 0.227 | 6.690 | 1.678 |
| $bicg_{sym}$ | | - | 21 | 780.744 | 21 | 36.619 | 0.259 | 7.242 | 18.599 | 14.904 | 9.696 | 10.190 | 0.227 | 5.719 | 1.436 | | - | 21 | 573.004 | 21 | 26.743 | 0.261 | 7.223 | 10.107 | 15.113 | 5.115 | 5.202 | 0.227 | 5.719 | 1.436 |
| bicgstab | | - | 12 | 874.830 | 24 | 35.874 | 0.258 | 7.357 | 18.095 | 14.904 | 9.474 | 9.855 | 0.228 | 6.690 | 1.678 | | - | 12 | 653.739 | 24 | 26.662 | 0.256 | 7.278 | 10.110 | 15.107 | 5.076 | 5.208 | 0.227 | 6.690 | 1.678 |
| cg | | - | 21 | 771.419 | 21 | 36.212 | 0.257 | 7.329 | 18.254 | 14.910 | 9.540 | 10.043 | 0.225 | 5.719 | 1.436 | | - | 21 | 573.596 | 21 | 26.795 | 0.258 | 7.273 | 10.114 | 15.140 | 5.125 | 5.212 | 0.226 | 5.719 | 1.436 |
| cgs | | - | 13 | 946.171 | 26 | 35.899 | 0.258 | 7.364 | 18.055 | 14.918 | 9.516 | 9.998 | 0.228 | 6.690 | 1.678 | | - | 13 | 707.863 | 26 | 26.733 | 0.255 | 7.321 | 10.211 | 15.015 | 5.135 | 5.308 | 0.227 | 6.690 | 1.678 |
| mlbicgstab | | 1 | 12 | 889.369 | 24 | 36.313 | 0.261 | 7.421 | 18.279 | 14.926 | 9.658 | 10.113 | 0.227 | 8.308 | 2.083 | | 1 | 12 | 660.414 | 24 | 26.803 | 0.256 | 7.226 | 10.209 | 15.128 | 5.166 | 5.269 | 0.226 | 8.308 | 2.083 |
| mlbicgstab | | 2 | 8 | 862.058 | 23 | 36.394 | 0.257 | 7.323 | 18.191 | 14.911 | 9.632 | 10.122 | 0.226 | 9.602 | 2.407 | | 2 | 8 | 642.143 | 23 | 26.816 | 0.258 | 7.306 | 10.173 | 15.137 | 5.171 | 5.245 | 0.224 | 9.602 | 2.407 |
| mlbicgstab | | 3 | 6 | 837.998 | 22 | 36.266 | 0.257 | 7.382 | 18.432 | 15.059 | 9.726 | 9.999 | 0.228 | 10.897 | 2.730 | | 3 | 7 | 745.327 | 26 | 27.178 | 0.256 | 7.269 | 10.290 | 15.105 | 5.364 | 5.536 | 0.227 | 10.897 | 2.730 |
| $mlbicgstab_{orig}$ | | 1 | 12 | 885.918 | 24 | 36.041 | 0.256 | 7.449 | 18.068 | 14.977 | 9.521 | 9.963 | 0.226 | 8.308 | 2.083 | | 1 | 11 | 615.325 | 22 | 26.856 | 0.258 | 7.241 | 10.160 | 15.160 | 5.124 | 5.339 | 0.225 | 8.308 | 2.083 |
| $mlbicgstab_{orig}$ | | 2 | 8 | 863.017 | 23 | 36.254 | 0.257 | 7.256 | 18.118 | 14.966 | 9.586 | 10.075 | 0.226 | 9.602 | 2.407 | | 2 | 8 | 651.255 | 23 | 27.002 | 0.257 | 7.275 | 10.150 | 15.185 | 5.207 | 5.342 | 0.223 | 9.602 | 2.407 |
| $mlbicgstab_{orig}$ | | 3 | 6 | 834.602 | 22 | 36.261 | 0.258 | 7.365 | 18.133 | 15.133 | 9.560 | 9.879 | 0.227 | 10.897 | 2.730 | | 3 | 7 | 746.441 | 26 | 26.958 | 0.256 | 7.301 | 10.204 | 15.028 | 5.280 | 5.370 | 0.225 | 10.897 | 2.730 |
| $mlbicgstab_{ss}$ | | 1 | 12 | 849.469 | 23 | 36.189 | 0.260 | 7.266 | 18.465 | 14.882 | 9.599 | 10.103 | 0.226 | 7.013 | 1.759 | | 1 | 12 | 631.810 | 23 | 26.717 | 0.254 | 7.237 | 10.088 | 15.143 | 5.110 | 5.221 | 0.224 | 7.013 | 1.759 |
| $mlbicgstab_{ss}$ | | 2 | 8 | 846.749 | 23 | 35.836 | 0.258 | 7.280 | 18.138 | 14.917 | 9.388 | 9.922 | 0.230 | 7.984 | 2.002 | | 2 | 8 | 639.041 | 23 | 26.780 | 0.256 | 7.305 | 10.126 | 15.040 | 5.139 | 5.277 | 0.228 | 7.984 | 2.002 |
| $mlbicgstab_{ss}$ | | 3 | 6 | 868.573 | 23 | 36.183 | 0.257 | 7.456 | 18.205 | 14.919 | 9.609 | 10.132 | 0.225 | 8.955 | 2.245 | | 3 | 6 | 651.539 | 23 | 27.030 | 0.256 | 7.354 | 10.288 | 15.191 | 5.186 | 5.317 | 0.223 | 8.955 | 2.245 |
| qmr | | - | 22 | 3864.717 | 109 | 35.153 | 0.250 | 7.261 | 18.167 | 14.923 | 9.564 | 9.942 | 0.226 | 7.337 | 1.840 | | - | 22 | 2898.344 | 109 | 26.296 | 0.247 | 7.245 | 10.158 | 15.189 | 5.126 | 5.347 | 0.224 | 7.337 | 1.840 |
| $qmr_{sym}$ | | - | 21 | 779.725 | 21 | 36.343 | 0.257 | 7.268 | 18.439 | 14.923 | 9.648 | 9.987 | 0.227 | 6.690 | 1.678 | | - | 21 | 581.214 | 21 | 26.873 | 0.256 | 7.475 | 10.116 | 15.156 | 5.093 | 5.292 | 0.227 | 6.690 | 1.678 |
| rbicgstab | | 1 | 14 | 1023.638 | 28 | 35.950 | 0.254 | 7.264 | 18.251 | 14.929 | 9.523 | 10.024 | 0.226 | 6.690 | 1.678 | | 1 | 14 | 766.286 | 28 | 26.769 | 0.254 | 7.398 | 10.160 | 15.161 | 5.173 | 5.320 | 0.226 | 6.690 | 1.678 |
| rbicgstab | | 2 | 7 | 1028.850 | 28 | 36.075 | 0.257 | 7.234 | 18.267 | 14.963 | 9.566 | 10.148 | 0.228 | 7.337 | 1.840 | | 2 | 7 | 770.280 | 28 | 26.821 | 0.252 | 7.401 | 10.253 | 15.155 | 5.137 | 5.385 | 0.227 | 7.337 | 1.840 |
| rbicgstab | | 3 | 5 | 1092.793 | 30 | 35.701 | 0.254 | 7.336 | 18.310 | 14.833 | 9.594 | 9.935 | 0.230 | 7.984 | 2.002 | | 3 | 5 | 823.544 | 30 | 26.682 | 0.251 | 7.306 | 10.077 | 15.244 | 5.127 | 5.249 | 0.227 | 7.984 | 2.002 |
| tfqmr | | - | 13 | 1900.183 | 53 | 35.383 | 0.255 | 7.479 | 18.335 | 14.956 | 9.566 | 9.877 | 0.227 | 7.337 | 1.840 | | - | 13 | 1424.579 | 53 | 26.468 | 0.249 | 7.360 | 10.245 | 15.022 | 5.178 | 5.384 | 0.224 | 7.337 | 1.840 |
| bicg | **2** | - | 21 | 1206.954 | 42 | 28.230 | 0.246 | 7.195 | 10.821 | 14.786 | 5.404 | 7.136 | 0.228 | 6.695 | 1.679 | **5** | - | 21 | 1380.124 | 42 | 32.045 | 0.247 | 7.233 | 14.200 | 14.854 | 7.940 | 8.223 | 0.227 | 6.718 | 1.684 |
| $bicg_{sym}$ | | - | 21 | 611.491 | 21 | 28.482 | 0.253 | 7.205 | 10.844 | 14.747 | 5.332 | 7.065 | 0.225 | 5.724 | 1.437 | | - | 21 | 689.845 | 21 | 31.893 | 0.252 | 7.220 | 13.023 | 14.667 | 8.168 | 7.475 | 0.224 | 5.748 | 1.441 |
| bicgstab | | - | 12 | 698.778 | 24 | 28.442 | 0.251 | 7.262 | 10.859 | 14.813 | 5.332 | 7.059 | 0.228 | 6.695 | 1.679 | | - | 12 | 799.747 | 24 | 32.177 | 0.248 | 7.225 | 12.996 | 14.708 | 7.993 | 7.942 | 0.227 | 6.718 | 1.684 |
| cg | | - | 21 | 614.930 | 21 | 28.645 | 0.252 | 7.244 | 10.866 | 14.782 | 5.337 | 7.186 | 0.225 | 5.724 | 1.437 | | - | 21 | 682.129 | 21 | 31.536 | 0.251 | 7.232 | 13.151 | 14.720 | 7.703 | 7.368 | 0.225 | 5.748 | 1.441 |
| cgs | | - | 13 | 758.871 | 26 | 28.575 | 0.250 | 7.211 | 10.799 | 14.815 | 5.344 | 7.245 | 0.227 | 6.695 | 1.679 | | - | 13 | 860.573 | 26 | 31.946 | 0.249 | 7.246 | 14.222 | 14.800 | 7.792 | 8.048 | 0.229 | 6.718 | 1.684 |
| mlbicgstab | | 1 | 11 | 653.973 | 22 | 28.890 | 0.250 | 7.201 | 10.770 | 14.841 | 5.506 | 7.251 | 0.225 | 8.313 | 2.084 | | 1 | 11 | 736.563 | 22 | 31.994 | 0.249 | 7.183 | 14.284 | 14.820 | 7.812 | 7.994 | 0.225 | 8.336 | 2.088 |
| mlbicgstab | | 2 | 8 | 692.872 | 23 | 28.875 | 0.249 | 7.220 | 10.774 | 14.834 | 5.505 | 7.281 | 0.225 | 9.607 | 2.407 | | 2 | 8 | 772.851 | 23 | 31.934 | 0.249 | 7.217 | 14.438 | 14.800 | 7.872 | 7.842 | 0.225 | 9.631 | 2.412 |
| mlbicgstab | | 3 | 6 | 674.459 | 22 | 29.061 | 0.250 | 7.223 | 10.871 | 15.025 | 5.590 | 7.110 | 0.225 | 10.901 | 2.731 | | 3 | 7 | 866.690 | 26 | 31.268 | 0.248 | 7.192 | 13.089 | 14.797 | 7.325 | 7.846 | 0.226 | 10.925 | 2.735 |
| $mlbicgstab_{orig}$ | | 1 | 12 | 715.069 | 24 | 28.778 | 0.249 | 7.188 | 10.769 | 14.795 | 5.503 | 7.257 | 0.226 | 8.313 | 2.084 | | 1 | 11 | 744.126 | 22 | 32.210 | 0.251 | 7.521 | 14.473 | 14.811 | 7.751 | 8.131 | 0.227 | 8.336 | 2.088 |
| $mlbicgstab_{orig}$ | | 2 | 8 | 717.513 | 23 | 28.765 | 0.248 | 7.222 | 10.789 | 14.768 | 5.510 | 7.318 | 0.226 | 9.607 | 2.407 | | 2 | 8 | 786.697 | 23 | 32.157 | 0.249 | 7.313 | 13.122 | 14.869 | 7.603 | 8.102 | 0.226 | 9.631 | 2.412 |
| $mlbicgstab_{orig}$ | | 3 | 7 | 798.671 | 26 | 28.884 | 0.249 | 7.234 | 10.896 | 15.021 | 5.548 | 7.075 | 0.225 | 10.901 | 2.731 | | 3 | 6 | 760.802 | 22 | 32.189 | 0.251 | 7.210 | 14.201 | 14.795 | 7.749 | 8.118 | 0.226 | 10.925 | 2.735 |
| $mlbicgstab_{ss}$ | | 1 | 12 | 689.804 | 23 | 28.752 | 0.248 | 7.205 | 10.853 | 14.808 | 5.360 | 7.230 | 0.225 | 7.018 | 1.760 | | 1 | 12 | 770.745 | 23 | 32.074 | 0.250 | 7.191 | 14.226 | 14.741 | 7.651 | 8.277 | 0.223 | 7.042 | 1.765 |
| $mlbicgstab_{ss}$ | | 2 | 8 | 687.744 | 23 | 28.771 | 0.249 | 7.210 | 10.769 | 14.778 | 5.498 | 7.225 | 0.228 | 7.989 | 2.003 | | 2 | 8 | 767.604 | 23 | 31.774 | 0.248 | 7.201 | 14.270 | 14.834 | 7.644 | 7.817 | 0.228 | 8.013 | 2.007 |
| $mlbicgstab_{ss}$ | | 3 | 6 | 699.189 | 23 | 28.965 | 0.249 | 7.281 | 10.868 | 14.797 | 5.518 | 7.387 | 0.223 | 8.960 | 2.246 | | 3 | 6 | 777.458 | 23 | 31.898 | 0.248 | 7.217 | 14.312 | 14.830 | 7.810 | 7.852 | 0.222 | 8.984 | 2.250 |
| qmr | | - | 22 | 3118.214 | 109 | 28.259 | 0.245 | 7.243 | 10.708 | 14.815 | 5.428 | 7.191 | 0.225 | 7.342 | 1.841 | | - | 22 | 3495.266 | 109 | 31.542 | 0.243 | 7.516 | 14.452 | 14.746 | 8.069 | 7.750 | 0.224 | 7.366 | 1.845 |
| $qmr_{sym}$ | | - | 21 | 621.480 | 21 | 28.703 | 0.253 | 7.237 | 10.721 | 14.858 | 5.315 | 7.193 | 0.227 | 6.695 | 1.679 | | - | 21 | 696.339 | 21 | 31.817 | 0.252 | 7.209 | 15.097 | 14.746 | 7.487 | 7.937 | 0.227 | 6.718 | 1.684 |
| rbicgstab | | 1 | 14 | 815.457 | 28 | 28.397 | 0.250 | 7.290 | 10.850 | 14.811 | 5.322 | 7.137 | 0.227 | 6.695 | 1.679 | | 1 | 14 | 927.269 | 28 | 31.837 | 0.248 | 7.238 | 14.217 | 14.743 | 7.640 | 8.187 | 0.226 | 6.718 | 1.684 |
| rbicgstab | | 2 | 7 | 827.341 | 28 | 28.761 | 0.249 | 7.248 | 10.737 | 14.786 | 5.530 | 7.322 | 0.227 | 7.342 | 1.841 | | 2 | 7 | 928.027 | 28 | 31.954 | 0.247 | 7.305 | 14.559 | 14.783 | 7.930 | 7.577 | 0.227 | 7.366 | 1.845 |
| rbicgstab | | 3 | 5 | 883.249 | 30 | 28.586 | 0.248 | 7.256 | 10.825 | 14.797 | 5.432 | 7.199 | 0.229 | 7.989 | 2.003 | | 3 | 5 | 999.106 | 30 | 31.887 | 0.247 | 7.207 | 15.194 | 14.712 | 7.793 | 8.150 | 0.228 | 8.013 | 2.007 |
| tfqmr | | - | 13 | 1535.586 | 53 | 28.507 | 0.257 | 7.438 | 10.958 | 14.895 | 5.462 | 7.299 | 0.229 | 7.342 | 1.841 | | - | 13 | 1704.325 | 53 | 31.270 | 0.244 | 7.228 | 15.419 | 14.749 | 7.608 | 8.013 | 0.224 | 7.366 | 1.845 |

Figure D.2: The performance, using 4 nodes on WALTON, of the full MPI-based distributed-memory system as a function of the distributed-transpose algorithm employed to support the DFT kernel, the iterative scheme, and where applicable, the number of starting vectors.

| Scheme | TR | SV | NI | CT (s) | MVPs | MVP (s) | VI (s) | TDFT (s) | TFTR (s) | CORE (s) | VFTR (s) | VRTR (s) | NDDC (s) | Memory Total (GB) | Memory Node (GB) | TR | SV | NI | CT (s) | MVPs | MVP (s) | VI (s) | TDFT (s) | TFTR (s) | CORE (s) | VFTR (s) | VRTR (s) | NDDC (s) | Memory Total (GB) | Memory Node (GB) |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| bicg | **0** | - | 20 | 1438.244 | 40 | 35.460 | 0.221 | 8.932 | 15.667 | 17.236 | 8.085 | 8.743 | 0.241 | 14.678 | 1.842 | **3** | | 20 | 1279.492 | 40 | 31.485 | 0.221 | 8.958 | 15.524 | 17.686 | 6.098 | 6.615 | 0.240 | 14.678 | 1.842 |
| $bicg_{sym}$ | | - | 20 | 728.726 | 20 | 35.845 | 0.227 | 8.954 | 15.767 | 17.146 | 8.101 | 8.883 | 0.240 | 12.586 | 1.581 | | - | 20 | 646.767 | 20 | 31.740 | 0.227 | 8.923 | 15.250 | 17.691 | 5.986 | 6.383 | 0.241 | 12.586 | 1.581 |
| bicgstab | | - | 12 | 879.820 | 24 | 35.949 | 0.223 | 8.922 | 15.668 | 17.065 | 8.186 | 9.058 | 0.240 | 14.678 | 1.842 | | - | 12 | 779.166 | 24 | 31.746 | 0.223 | 8.948 | 15.474 | 17.644 | 6.047 | 6.434 | 0.239 | 14.678 | 1.842 |
| cg | | - | 20 | 728.871 | 20 | 35.852 | 0.226 | 8.890 | 16.049 | 17.008 | 8.240 | 8.789 | 0.241 | 12.586 | 1.581 | | - | 20 | 647.969 | 20 | 31.805 | 0.228 | 8.910 | 15.371 | 17.306 | 5.995 | 6.485 | 0.242 | 12.586 | 1.581 |
| cgs | | - | 13 | 958.390 | 26 | 36.261 | 0.225 | 9.006 | 16.103 | 17.172 | 8.270 | 9.061 | 0.241 | 14.678 | 1.842 | | - | 13 | 824.561 | 26 | 31.170 | 0.223 | 8.932 | 15.343 | 17.446 | 6.115 | 6.208 | 0.240 | 14.678 | 1.842 |
| mlbicgstab | | 1 | 11 | 817.849 | 22 | 36.302 | 0.224 | 8.948 | 15.736 | 17.240 | 8.254 | 8.812 | 0.242 | 18.165 | 2.278 | | 1 | 11 | 722.385 | 22 | 31.953 | 0.224 | 8.979 | 15.359 | 17.490 | 6.178 | 6.696 | 0.242 | 18.165 | 2.278 |
| mlbicgstab | | 2 | 8 | 854.923 | 23 | 35.911 | 0.224 | 9.101 | 16.340 | 17.164 | 8.202 | 8.968 | 0.242 | 20.954 | 2.627 | | 2 | 8 | 757.498 | 23 | 31.677 | 0.224 | 8.934 | 15.931 | 17.502 | 6.178 | 6.434 | 0.243 | 20.954 | 2.627 |
| mlbicgstab | | 3 | 6 | 842.725 | 22 | 36.653 | 0.224 | 9.017 | 16.445 | 17.365 | 8.365 | 8.973 | 0.239 | 23.744 | 2.976 | | 3 | 6 | 735.410 | 22 | 31.765 | 0.224 | 9.035 | 15.857 | 17.457 | 6.095 | 6.603 | 0.238 | 23.744 | 2.976 |
| $mlbicgstab_{orig}$ | | 1 | 12 | 896.297 | 24 | 36.249 | 0.224 | 9.059 | 16.119 | 17.281 | 8.237 | 8.975 | 0.244 | 18.165 | 2.278 | | 1 | 12 | 793.983 | 24 | 31.988 | 0.223 | 9.152 | 15.528 | 17.793 | 6.109 | 6.543 | 0.242 | 18.165 | 2.278 |
| $mlbicgstab_{orig}$ | | 2 | 8 | 868.017 | 23 | 36.170 | 0.222 | 8.941 | 16.826 | 17.432 | 8.465 | 8.659 | 0.243 | 20.954 | 2.627 | | 2 | 8 | 759.091 | 23 | 31.476 | 0.224 | 8.948 | 15.779 | 17.492 | 6.137 | 6.289 | 0.242 | 20.954 | 2.627 |
| $mlbicgstab_{orig}$ | | 3 | 6 | 857.871 | 22 | 35.963 | 0.225 | 9.018 | 16.959 | 17.499 | 8.454 | 8.546 | 0.241 | 23.744 | 2.976 | | 3 | 6 | 748.467 | 22 | 32.150 | 0.224 | 9.012 | 15.917 | 17.638 | 6.148 | 6.764 | 0.241 | 23.744 | 2.976 |
| $mlbicgstab_{ss}$ | | 1 | 12 | 858.978 | 23 | 36.227 | 0.222 | 8.931 | 16.015 | 17.203 | 8.219 | 9.118 | 0.243 | 15.375 | 1.930 | | 1 | 12 | 750.185 | 23 | 31.782 | 0.222 | 8.973 | 15.368 | 17.480 | 6.056 | 6.535 | 0.243 | 15.375 | 1.930 |
| $mlbicgstab_{ss}$ | | 2 | 8 | 848.492 | 23 | 35.760 | 0.222 | 8.979 | 15.800 | 17.350 | 8.204 | 8.662 | 0.239 | 17.467 | 2.191 | | 2 | 8 | 751.538 | 23 | 31.560 | 0.224 | 9.032 | 16.030 | 17.340 | 6.136 | 6.479 | 0.239 | 17.467 | 2.191 |
| $mlbicgstab_{ss}$ | | 3 | 6 | 875.250 | 23 | 36.624 | 0.223 | 9.029 | 16.664 | 17.423 | 8.397 | 8.993 | 0.238 | 19.559 | 2.453 | | 3 | 6 | 768.921 | 23 | 31.978 | 0.222 | 8.969 | 15.819 | 17.594 | 6.092 | 6.627 | 0.239 | 19.559 | 2.453 |
| qmr | | - | 21 | 3685.538 | 104 | 35.106 | 0.216 | 8.887 | 16.272 | 17.069 | 8.226 | 9.109 | 0.243 | 16.073 | 2.017 | | - | 21 | 3261.049 | 104 | 31.019 | 0.215 | 8.961 | 15.215 | 17.603 | 6.039 | 6.611 | 0.242 | 16.073 | 2.017 |
| $qmr_{sym}$ | | - | 20 | 741.428 | 20 | 36.168 | 0.226 | 8.926 | 15.819 | 17.019 | 8.156 | 9.048 | 0.241 | 14.678 | 1.842 | | - | 20 | 655.426 | 20 | 31.818 | 0.225 | 9.178 | 15.278 | 17.303 | 5.909 | 6.536 | 0.242 | 14.678 | 1.842 |
| rbicgstab | | 1 | 13 | 947.867 | 26 | 35.777 | 0.223 | 8.955 | 15.607 | 17.297 | 8.094 | 8.926 | 0.241 | 14.678 | 1.842 | | 1 | 13 | 842.503 | 26 | 31.704 | 0.223 | 8.998 | 15.192 | 17.567 | 6.020 | 6.612 | 0.243 | 14.678 | 1.842 |
| rbicgstab | | 2 | 7 | 1029.272 | 28 | 36.012 | 0.222 | 8.917 | 15.774 | 17.131 | 8.147 | 9.024 | 0.241 | 16.073 | 2.017 | | 2 | 7 | 889.908 | 28 | 31.013 | 0.222 | 8.907 | 15.378 | 17.480 | 6.016 | 6.160 | 0.242 | 16.073 | 2.017 |
| rbicgstab | | 3 | 5 | 1092.102 | 30 | 35.585 | 0.220 | 8.956 | 16.142 | 17.062 | 8.174 | 8.994 | 0.241 | 17.467 | 2.191 | | 3 | 5 | 950.787 | 30 | 30.863 | 0.220 | 8.900 | 15.371 | 17.497 | 5.978 | 6.111 | 0.242 | 17.467 | 2.191 |
| tfqmr | | - | 13 | 1886.154 | 53 | 35.129 | 0.217 | 8.892 | 15.665 | 17.048 | 8.076 | 8.901 | 0.242 | 16.073 | 2.017 | | - | 13 | 1647.102 | 53 | 30.611 | 0.217 | 8.993 | 15.163 | 17.473 | 6.032 | 6.147 | 0.242 | 16.073 | 2.017 |
| bicg | **2** | - | 20 | 1602.148 | 40 | 39.435 | 0.221 | 8.915 | 16.141 | 17.521 | 7.877 | 12.785 | 0.241 | 14.695 | 1.844 | **5** | | 20 | 2026.824 | 40 | 49.599 | 0.220 | 9.119 | 26.010 | 17.244 | 14.130 | 16.873 | 0.240 | 14.782 | 1.851 |
| $bicg_{sym}$ | | - | 20 | 811.177 | 20 | 39.770 | 0.227 | 8.937 | 15.720 | 17.256 | 7.592 | 13.219 | 0.241 | 12.603 | 1.582 | | - | 20 | 1046.614 | 20 | 50.532 | 0.227 | 8.949 | 27.465 | 17.271 | 13.509 | 17.208 | 0.242 | 12.690 | 1.590 |
| bicgstab | | - | 12 | 990.612 | 24 | 40.366 | 0.224 | 9.077 | 16.241 | 17.414 | 7.914 | 13.470 | 0.241 | 14.695 | 1.844 | | - | 12 | 1239.200 | 24 | 49.713 | 0.223 | 9.112 | 25.600 | 17.147 | 13.367 | 17.256 | 0.240 | 14.782 | 1.851 |
| cg | | - | 20 | 806.475 | 20 | 39.553 | 0.227 | 9.011 | 15.829 | 17.259 | 7.662 | 12.919 | 0.242 | 12.603 | 1.582 | | - | 20 | 1046.005 | 20 | 50.642 | 0.226 | 8.957 | 25.383 | 17.269 | 14.408 | 16.636 | 0.241 | 12.690 | 1.590 |
| cgs | | - | 13 | 1052.698 | 26 | 39.732 | 0.224 | 8.950 | 16.235 | 17.628 | 7.880 | 12.777 | 0.240 | 14.695 | 1.844 | | - | 13 | 1329.894 | 26 | 49.446 | 0.223 | 8.899 | 26.297 | 17.234 | 13.433 | 16.849 | 0.241 | 14.782 | 1.851 |
| mlbicgstab | | 1 | 12 | 1002.661 | 24 | 40.354 | 0.222 | 9.042 | 16.061 | 17.490 | 7.781 | 13.583 | 0.242 | 18.182 | 2.280 | | 1 | 11 | 1145.017 | 22 | 49.575 | 0.222 | 8.893 | 26.068 | 17.326 | 13.165 | 17.090 | 0.243 | 18.269 | 2.287 |
| mlbicgstab | | 2 | 8 | 964.968 | 23 | 40.459 | 0.223 | 9.043 | 16.037 | 17.575 | 7.806 | 13.344 | 0.243 | 20.971 | 2.629 | | 2 | 8 | 1215.407 | 23 | 50.496 | 0.223 | 9.024 | 31.395 | 17.534 | 13.356 | 17.208 | 0.242 | 21.058 | 2.636 |
| mlbicgstab | | 3 | 6 | 933.107 | 22 | 40.550 | 0.225 | 9.253 | 16.235 | 17.602 | 7.791 | 13.504 | 0.240 | 23.761 | 2.977 | | 3 | 6 | 1149.864 | 22 | 49.454 | 0.224 | 9.245 | 26.529 | 17.288 | 13.338 | 16.738 | 0.239 | 23.847 | 2.985 |
| $mlbicgstab_{orig}$ | | 1 | 12 | 997.772 | 24 | 40.262 | 0.224 | 9.090 | 16.056 | 17.312 | 7.823 | 13.604 | 0.244 | 18.182 | 2.280 | | 1 | 11 | 1157.898 | 22 | 49.515 | 0.224 | 8.916 | 27.289 | 17.222 | 13.313 | 16.708 | 0.244 | 18.269 | 2.287 |
| $mlbicgstab_{orig}$ | | 2 | 8 | 969.216 | 23 | 40.381 | 0.225 | 9.024 | 16.333 | 17.635 | 7.821 | 13.202 | 0.243 | 20.971 | 2.629 | | 2 | 8 | 1217.933 | 23 | 50.343 | 0.224 | 9.043 | 31.875 | 17.600 | 13.237 | 17.151 | 0.242 | 21.058 | 2.636 |
| $mlbicgstab_{orig}$ | | 3 | 6 | 934.186 | 22 | 40.367 | 0.224 | 9.072 | 16.535 | 17.638 | 7.860 | 13.142 | 0.241 | 23.761 | 2.977 | | 3 | 6 | 1181.083 | 22 | 50.400 | 0.222 | 9.076 | 30.762 | 17.621 | 13.337 | 17.263 | 0.240 | 23.847 | 2.985 |
| $mlbicgstab_{ss}$ | | 1 | 12 | 943.836 | 23 | 40.001 | 0.222 | 8.969 | 16.169 | 17.409 | 7.681 | 13.338 | 0.243 | 15.393 | 1.931 | | 1 | 12 | 1173.007 | 23 | 48.816 | 0.221 | 8.924 | 26.342 | 17.199 | 13.553 | 15.784 | 0.243 | 15.479 | 1.939 |
| $mlbicgstab_{ss}$ | | 2 | 8 | 929.506 | 23 | 39.067 | 0.225 | 9.124 | 16.067 | 17.328 | 7.558 | 12.621 | 0.240 | 17.485 | 2.193 | | 2 | 8 | 1199.101 | 23 | 49.891 | 0.223 | 9.085 | 26.781 | 17.222 | 13.555 | 16.577 | 0.240 | 17.571 | 2.200 |
| $mlbicgstab_{ss}$ | | 3 | 6 | 949.053 | 23 | 39.658 | 0.223 | 9.062 | 16.398 | 17.615 | 7.658 | 12.811 | 0.239 | 19.577 | 2.454 | | 3 | 6 | 1230.009 | 23 | 50.899 | 0.223 | 9.039 | 31.934 | 17.571 | 13.412 | 17.439 | 0.238 | 19.663 | 2.462 |
| qmr | | - | 21 | 4146.588 | 104 | 39.456 | 0.216 | 9.007 | 16.077 | 17.323 | 7.890 | 13.541 | 0.242 | 16.090 | 2.018 | | - | 21 | 5179.466 | 104 | 49.042 | 0.215 | 9.065 | 25.701 | 17.275 | 13.889 | 16.986 | 0.242 | 16.177 | 2.026 |
| $qmr_{sym}$ | | - | 20 | 828.172 | 20 | 40.216 | 0.226 | 9.015 | 15.885 | 17.251 | 7.699 | 13.525 | 0.242 | 14.695 | 1.844 | | - | 20 | 1033.088 | 20 | 49.686 | 0.226 | 9.003 | 26.514 | 17.157 | 13.270 | 16.832 | 0.241 | 14.782 | 1.851 |
| rbicgstab | | 1 | 13 | 1034.537 | 26 | 38.923 | 0.223 | 8.979 | 15.868 | 17.136 | 7.519 | 12.839 | 0.243 | 14.695 | 1.844 | | 1 | 13 | 1334.005 | 26 | 49.390 | 0.223 | 8.959 | 26.047 | 17.305 | 13.341 | 16.932 | 0.241 | 14.782 | 1.851 |
| rbicgstab | | 2 | 7 | 1141.862 | 28 | 39.823 | 0.222 | 9.123 | 15.883 | 17.294 | 7.747 | 13.397 | 0.242 | 16.090 | 2.018 | | 2 | 7 | 1453.059 | 28 | 49.978 | 0.221 | 8.948 | 27.500 | 17.451 | 13.337 | 17.405 | 0.242 | 16.177 | 2.026 |
| rbicgstab | | 3 | 5 | 1242.260 | 30 | 40.364 | 0.220 | 9.119 | 15.960 | 17.432 | 7.633 | 13.947 | 0.242 | 17.485 | 2.193 | | 3 | 5 | 1532.648 | 30 | 49.060 | 0.220 | 8.900 | 26.523 | 17.184 | 13.191 | 16.848 | 0.243 | 17.571 | 2.200 |
| tfqmr | | - | 13 | 2136.915 | 53 | 39.693 | 0.217 | 8.934 | 15.848 | 17.354 | 7.831 | 13.564 | 0.242 | 16.090 | 2.018 | | - | 13 | 2661.621 | 53 | 48.848 | 0.216 | 9.122 | 26.864 | 17.091 | 13.573 | 17.044 | 0.242 | 16.177 | 2.026 |

Figure D.3: The performance, using 8 nodes on WALTON, of the full MPI-based distributed-memory system as a function of the distributed-transpose algorithm employed to support the DFT kernel, the iterative scheme, and where applicable, the number of starting vectors.

| Scheme | TR | SV | NI | CT (s) | MVPs | MVP (s) | VI (s) | TDFT (s) | TFTR (s) | CORE (s) | VFTR (s) | VRTR (s) | NDDC (s) | Memory Total (GB) | Memory Node (GB) | TR | SV | NI | CT (s) | MVPs | MVP (s) | VI (s) | TDFT (s) | TFTR (s) | CORE (s) | VFTR (s) | VRTR (s) | NDDC (s) | Memory Total (GB) | Memory Node (GB) |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| bicg | **0** | - | 20 | 1286.443 | 40 | 31.658 | 0.203 | 7.903 | 11.653 | 17.951 | 6.044 | 6.749 | 0.221 | 27.461 | 1.730 | **3** | | 20 | 1479.110 | 40 | 36.478 | 0.203 | 7.951 | 21.296 | 18.267 | 7.894 | 9.058 | 0.222 | 27.461 | 1.730 |
| $bicg_{sym}$ | | - | 20 | 641.455 | 20 | 31.443 | 0.209 | 7.884 | 12.055 | 17.730 | 5.891 | 6.390 | 0.222 | 23.608 | 1.489 | | - | 20 | 745.734 | 20 | 36.619 | 0.208 | 7.857 | 21.717 | 18.055 | 7.831 | 8.798 | 0.222 | 23.608 | 1.489 |
| bicgstab | | - | 12 | 792.865 | 24 | 32.330 | 0.207 | 7.835 | 11.604 | 17.900 | 5.899 | 6.800 | 0.222 | 27.461 | 1.730 | | - | 12 | 892.204 | 24 | 36.462 | 0.205 | 7.785 | 21.156 | 18.189 | 7.713 | 8.928 | 0.222 | 27.461 | 1.730 |
| cg | | - | 20 | 653.463 | 20 | 31.995 | 0.209 | 7.800 | 12.115 | 17.818 | 5.961 | 6.774 | 0.224 | 23.608 | 1.489 | | - | 20 | 741.702 | 20 | 36.433 | 0.209 | 7.781 | 21.012 | 18.077 | 7.618 | 8.837 | 0.224 | 23.608 | 1.489 |
| cgs | | - | 13 | 858.978 | 26 | 32.422 | 0.206 | 7.801 | 12.104 | 17.988 | 5.961 | 7.214 | 0.221 | 27.461 | 1.730 | | - | 13 | 960.264 | 26 | 36.341 | 0.206 | 7.867 | 20.965 | 18.582 | 7.782 | 8.278 | 0.222 | 27.461 | 1.730 |
| mlbicgstab | | 1 | 12 | 796.891 | 24 | 32.381 | 0.206 | 7.801 | 11.776 | 17.900 | 6.004 | 7.226 | 0.223 | 33.883 | 2.132 | | 1 | 11 | 822.070 | 22 | 36.493 | 0.206 | 7.840 | 21.218 | 18.472 | 7.592 | 8.666 | 0.223 | 33.883 | 2.132 |
| mlbicgstab | | 2 | 8 | 774.059 | 23 | 32.422 | 0.207 | 7.783 | 11.717 | 17.981 | 5.933 | 7.123 | 0.223 | 39.021 | 2.453 | | 2 | 8 | 864.937 | 23 | 36.338 | 0.206 | 7.889 | 20.094 | 18.359 | 7.638 | 8.207 | 0.223 | 39.021 | 2.453 |
| mlbicgstab | | 3 | 6 | 763.835 | 22 | 32.310 | 0.205 | 7.846 | 12.130 | 18.438 | 6.007 | 6.265 | 0.220 | 44.159 | 2.775 | | 3 | 6 | 838.573 | 22 | 36.508 | 0.207 | 7.809 | 20.377 | 18.903 | 7.642 | 8.040 | 0.221 | 44.159 | 2.775 |
| $mlbicgstab_{orig}$ | | 1 | 12 | 809.814 | 24 | 32.700 | 0.205 | 7.779 | 11.560 | 17.957 | 5.967 | 7.540 | 0.222 | 33.883 | 2.132 | | 1 | 11 | 823.612 | 22 | 36.350 | 0.207 | 7.777 | 20.734 | 18.363 | 7.631 | 8.206 | 0.223 | 33.883 | 2.132 |
| $mlbicgstab_{orig}$ | | 2 | 8 | 782.094 | 23 | 32.519 | 0.206 | 7.777 | 11.406 | 18.115 | 5.898 | 7.162 | 0.224 | 39.021 | 2.453 | | 2 | 8 | 867.676 | 23 | 36.218 | 0.206 | 7.768 | 20.667 | 18.774 | 7.568 | 7.968 | 0.223 | 39.021 | 2.453 |
| $mlbicgstab_{orig}$ | | 3 | 6 | 756.063 | 22 | 32.488 | 0.207 | 7.799 | 11.883 | 18.740 | 5.968 | 6.262 | 0.222 | 44.159 | 2.775 | | 3 | 6 | 853.863 | 22 | 36.740 | 0.206 | 7.815 | 19.955 | 18.456 | 7.758 | 8.365 | 0.222 | 44.159 | 2.775 |
| $mlbicgstab_{ss}$ | | 1 | 12 | 766.979 | 23 | 32.531 | 0.208 | 7.797 | 11.718 | 18.099 | 5.922 | 7.222 | 0.223 | 28.745 | 1.811 | | 1 | 12 | 858.310 | 23 | 36.431 | 0.206 | 7.832 | 20.741 | 18.341 | 7.547 | 8.848 | 0.223 | 28.745 | 1.811 |
| $mlbicgstab_{ss}$ | | 2 | 8 | 767.323 | 23 | 32.300 | 0.205 | 7.761 | 11.707 | 17.962 | 5.973 | 7.095 | 0.219 | 32.599 | 2.052 | | 2 | 8 | 856.650 | 23 | 36.104 | 0.204 | 7.776 | 20.744 | 18.505 | 7.636 | 7.972 | 0.220 | 32.599 | 2.052 |
| $mlbicgstab_{ss}$ | | 3 | 6 | 778.610 | 23 | 32.490 | 0.205 | 7.916 | 11.865 | 17.931 | 5.953 | 7.320 | 0.221 | 36.452 | 2.293 | | 3 | 6 | 866.655 | 23 | 36.275 | 0.206 | 7.862 | 20.615 | 18.493 | 7.639 | 8.468 | 0.221 | 36.452 | 2.293 |
| qmr | | - | 21 | 3427.417 | 104 | 32.617 | 0.198 | 7.806 | 11.676 | 18.109 | 5.982 | 7.559 | 0.222 | 30.030 | 1.891 | | - | 21 | 3753.948 | 104 | 35.743 | 0.199 | 7.849 | 20.948 | 18.290 | 7.722 | 8.959 | 0.222 | 30.030 | 1.891 |
| $qmr_{sym}$ | | - | 20 | 665.106 | 20 | 32.301 | 0.209 | 7.825 | 12.245 | 17.809 | 6.258 | 6.721 | 0.222 | 27.461 | 1.730 | | - | 20 | 752.736 | 20 | 36.693 | 0.211 | 7.821 | 20.955 | 18.272 | 7.750 | 8.786 | 0.225 | 27.461 | 1.730 |
| rbicgstab | | 1 | 13 | 856.379 | 26 | 32.228 | 0.205 | 7.909 | 11.696 | 18.036 | 6.069 | 6.935 | 0.220 | 27.461 | 1.730 | | 1 | 13 | 964.042 | 26 | 36.336 | 0.206 | 7.872 | 21.074 | 18.235 | 7.552 | 8.891 | 0.221 | 27.461 | 1.730 |
| rbicgstab | | 2 | 7 | 943.487 | 28 | 32.955 | 0.205 | 7.882 | 11.744 | 18.148 | 6.037 | 7.225 | 0.221 | 30.030 | 1.891 | | 2 | 7 | 1040.399 | 28 | 36.340 | 0.204 | 7.806 | 20.806 | 18.342 | 7.636 | 8.908 | 0.222 | 30.030 | 1.891 |
| rbicgstab | | 3 | 4 | 815.725 | 24 | 33.148 | 0.206 | 7.791 | 12.004 | 18.070 | 6.084 | 7.421 | 0.222 | 32.599 | 2.052 | | 3 | 4 | 897.006 | 24 | 36.514 | 0.205 | 7.814 | 21.225 | 18.313 | 7.583 | 8.975 | 0.222 | 32.599 | 2.052 |
| tfqmr | | - | 14 | 1857.767 | 56 | 32.695 | 0.200 | 7.754 | 12.114 | 17.902 | 6.172 | 7.492 | 0.222 | 30.030 | 1.891 | | - | 14 | 2045.803 | 56 | 36.022 | 0.200 | 7.814 | 21.414 | 18.212 | 7.694 | 8.989 | 0.222 | 30.030 | 1.891 |
| bicg | **2** | - | 20 | 1774.302 | 40 | 43.715 | 0.203 | 7.798 | 16.261 | 18.181 | 8.099 | 16.220 | 0.222 | 27.512 | 1.733 | **5** | | 20 | 2404.056 | 40 | 58.698 | 0.203 | 7.857 | 37.311 | 18.124 | 16.494 | 22.436 | 0.222 | 27.769 | 1.744 |
| $bicg_{sym}$ | | - | 20 | 879.159 | 20 | 43.068 | 0.208 | 7.930 | 16.035 | 18.093 | 7.840 | 15.505 | 0.223 | 23.659 | 1.492 | | - | 20 | 1227.856 | 20 | 59.179 | 0.208 | 7.868 | 39.781 | 18.059 | 15.350 | 22.793 | 0.225 | 23.916 | 1.503 |
| bicgstab | | - | 12 | 1062.928 | 24 | 43.322 | 0.205 | 7.801 | 15.816 | 18.068 | 7.857 | 15.928 | 0.222 | 27.512 | 1.733 | | - | 12 | 1445.018 | 24 | 57.875 | 0.205 | 7.851 | 38.433 | 18.034 | 15.312 | 22.174 | 0.222 | 27.769 | 1.744 |
| cg | | - | 20 | 879.508 | 20 | 43.122 | 0.208 | 7.851 | 16.281 | 18.125 | 7.907 | 15.446 | 0.223 | 23.659 | 1.492 | | - | 20 | 1214.228 | 20 | 58.703 | 0.209 | 7.963 | 36.784 | 18.133 | 15.699 | 22.040 | 0.224 | 23.916 | 1.503 |
| cgs | | - | 13 | 1166.507 | 26 | 44.032 | 0.206 | 7.794 | 16.086 | 18.241 | 8.137 | 16.265 | 0.221 | 27.512 | 1.733 | | - | 13 | 1577.234 | 26 | 58.677 | 0.205 | 7.888 | 39.053 | 18.084 | 15.645 | 22.352 | 0.222 | 27.769 | 1.744 |
| mlbicgstab | | 1 | 12 | 1072.580 | 24 | 43.619 | 0.206 | 7.807 | 16.479 | 18.080 | 8.069 | 15.988 | 0.223 | 33.935 | 2.134 | | 1 | 12 | 1457.635 | 24 | 58.486 | 0.205 | 7.744 | 38.354 | 18.083 | 15.462 | 22.398 | 0.223 | 34.191 | 2.146 |
| mlbicgstab | | 2 | 8 | 1035.951 | 23 | 43.636 | 0.206 | 7.752 | 16.316 | 18.398 | 8.043 | 15.565 | 0.222 | 39.072 | 2.456 | | 2 | 8 | 1403.261 | 23 | 58.155 | 0.205 | 7.786 | 37.543 | 18.026 | 15.387 | 22.286 | 0.223 | 39.329 | 2.467 |
| mlbicgstab | | 3 | 6 | 1001.553 | 22 | 43.729 | 0.206 | 7.921 | 16.738 | 18.829 | 8.255 | 15.019 | 0.221 | 44.210 | 2.777 | | 3 | 6 | 1369.944 | 22 | 58.907 | 0.207 | 7.935 | 39.301 | 18.484 | 15.532 | 22.018 | 0.221 | 44.467 | 2.788 |
| $mlbicgstab_{orig}$ | | 1 | 11 | 994.108 | 22 | 43.836 | 0.206 | 7.836 | 16.479 | 18.155 | 7.993 | 16.067 | 0.223 | 33.935 | 2.134 | | 1 | 11 | 1368.860 | 22 | 59.534 | 0.207 | 7.820 | 39.349 | 18.134 | 15.556 | 22.699 | 0.223 | 34.191 | 2.146 |
| $mlbicgstab_{orig}$ | | 2 | 8 | 1044.948 | 23 | 43.810 | 0.206 | 7.841 | 16.378 | 18.836 | 8.149 | 15.218 | 0.223 | 39.072 | 2.456 | | 2 | 8 | 1428.651 | 23 | 59.023 | 0.206 | 7.793 | 43.407 | 18.165 | 15.657 | 22.268 | 0.223 | 39.329 | 2.467 |
| $mlbicgstab_{orig}$ | | 3 | 6 | 1001.496 | 22 | 43.568 | 0.207 | 7.967 | 16.621 | 18.551 | 8.074 | 15.384 | 0.221 | 44.210 | 2.777 | | 3 | 6 | 1384.320 | 22 | 59.449 | 0.207 | 7.862 | 40.481 | 18.431 | 15.683 | 22.472 | 0.222 | 44.467 | 2.788 |
| $mlbicgstab_{ss}$ | | 1 | 12 | 1030.532 | 23 | 43.766 | 0.205 | 7.809 | 16.596 | 18.224 | 7.889 | 16.006 | 0.222 | 28.797 | 1.813 | | 1 | 12 | 1407.211 | 23 | 58.675 | 0.205 | 7.812 | 41.363 | 18.082 | 15.524 | 22.448 | 0.222 | 29.054 | 1.824 |
| $mlbicgstab_{ss}$ | | 2 | 8 | 1038.830 | 23 | 43.853 | 0.205 | 7.823 | 16.236 | 18.130 | 7.975 | 16.210 | 0.219 | 32.650 | 2.054 | | 2 | 8 | 1409.882 | 23 | 58.512 | 0.205 | 7.812 | 42.655 | 18.156 | 15.640 | 22.054 | 0.219 | 32.907 | 2.065 |
| $mlbicgstab_{ss}$ | | 3 | 6 | 1041.509 | 23 | 43.546 | 0.205 | 7.937 | 16.594 | 18.607 | 7.981 | 15.416 | 0.220 | 36.503 | 2.295 | | 3 | 6 | 1422.443 | 23 | 58.772 | 0.204 | 7.775 | 39.335 | 18.162 | 15.670 | 22.417 | 0.220 | 36.760 | 2.306 |
| qmr | | - | 21 | 4520.929 | 104 | 43.032 | 0.199 | 7.762 | 15.677 | 18.241 | 8.227 | 15.918 | 0.222 | 30.081 | 1.893 | | - | 21 | 6161.769 | 104 | 58.332 | 0.198 | 7.809 | 41.466 | 18.139 | 16.530 | 22.663 | 0.222 | 30.338 | 1.905 |
| $qmr_{sym}$ | | - | 20 | 891.665 | 20 | 43.378 | 0.208 | 7.821 | 16.261 | 18.154 | 7.812 | 15.713 | 0.223 | 27.512 | 1.733 | | - | 20 | 1235.740 | 20 | 59.259 | 0.208 | 7.861 | 38.312 | 18.237 | 15.579 | 22.300 | 0.222 | 27.769 | 1.744 |
| rbicgstab | | 1 | 13 | 1151.928 | 26 | 43.371 | 0.205 | 7.926 | 16.011 | 18.051 | 7.758 | 15.986 | 0.222 | 27.512 | 1.733 | | 1 | 13 | 1582.244 | 26 | 58.513 | 0.205 | 7.914 | 38.882 | 18.160 | 15.541 | 22.460 | 0.221 | 27.769 | 1.744 |
| rbicgstab | | 2 | 7 | 1246.873 | 28 | 43.484 | 0.204 | 7.953 | 16.795 | 18.184 | 7.998 | 15.913 | 0.222 | 30.081 | 1.893 | | 2 | 7 | 1714.382 | 28 | 58.661 | 0.204 | 7.870 | 40.174 | 18.060 | 15.465 | 22.783 | 0.221 | 30.338 | 1.905 |
| rbicgstab | | 3 | 4 | 1074.237 | 24 | 43.696 | 0.206 | 7.834 | 16.564 | 18.211 | 8.025 | 15.850 | 0.222 | 32.650 | 2.054 | | 3 | 4 | 1491.074 | 24 | 59.475 | 0.206 | 7.828 | 43.858 | 18.174 | 15.449 | 23.090 | 0.223 | 32.907 | 2.065 |
| tfqmr | | - | 14 | 2464.929 | 56 | 43.357 | 0.199 | 7.851 | 16.170 | 18.131 | 8.282 | 15.923 | 0.221 | 30.081 | 1.893 | | - | 14 | 3358.273 | 56 | 58.207 | 0.199 | 7.781 | 37.492 | 18.116 | 16.121 | 22.574 | 0.222 | 30.338 | 1.905 |

Figure D.4: The performance, using 16 nodes on WALTON, of the full MPI-based distributed-memory system as a function of the distributed-transpose algorithm employed to support the DFT kernel, the iterative scheme, and where applicable, the number of starting vectors.

| Scheme | TR | SV | NI | CT (s) | MVPs | MVP (s) | VI (s) | TDFT (s) | TFTR (s) | CORE (s) | VFTR (s) | VRTR (s) | NDDC (s) | Memory Total (GB) | Memory Node (GB) | TR | SV | NI | CT (s) | MVPs | MVP (s) | VI (s) | TDFT (s) | TFTR (s) | CORE (s) | VFTR (s) | VRTR (s) | NDDC (s) | Memory Total (GB) | Memory Node (GB) |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| bicg | **0** | - | 20 | 1415.370 | 40 | 34.928 | 0.203 | 8.279 | 14.808 | 18.144 | 7.316 | 8.371 | 0.221 | 54.402 | 1.740 | **3** | | 20 | 1970.138 | 40 | 48.751 | 0.204 | 8.172 | 37.797 | 18.859 | 13.918 | 14.381 | 0.222 | 54.402 | 1.740 |
| $bicg_{sym}$ | | - | 20 | 716.707 | 20 | 35.242 | 0.209 | 8.315 | 14.674 | 18.031 | 7.412 | 8.180 | 0.221 | 46.636 | 1.497 | | - | 20 | 1019.554 | 20 | 50.331 | 0.209 | 8.253 | 37.762 | 18.180 | 14.009 | 15.351 | 0.222 | 46.636 | 1.497 |
| bicgstab | | - | 12 | 858.664 | 24 | 35.088 | 0.207 | 8.202 | 15.090 | 18.047 | 7.324 | 8.316 | 0.222 | 54.402 | 1.740 | | - | 12 | 1192.354 | 24 | 48.969 | 0.206 | 8.168 | 37.631 | 18.165 | 13.544 | 14.739 | 0.221 | 54.402 | 1.740 |
| cg | | - | 20 | 710.296 | 20 | 34.963 | 0.209 | 8.177 | 14.928 | 18.019 | 7.168 | 8.163 | 0.221 | 46.636 | 1.497 | | - | 20 | 1004.947 | 20 | 49.596 | 0.210 | 8.282 | 37.902 | 18.170 | 13.883 | 14.785 | 0.221 | 46.636 | 1.497 |
| cgs | | - | 13 | 933.546 | 26 | 35.336 | 0.207 | 8.207 | 14.813 | 18.086 | 7.299 | 8.461 | 0.221 | 54.402 | 1.740 | | - | 13 | 1289.947 | 26 | 49.041 | 0.207 | 8.172 | 37.664 | 18.792 | 13.672 | 14.384 | 0.221 | 54.402 | 1.740 |
| mlbicgstab | | 1 | 11 | 877.317 | 22 | 35.725 | 0.207 | 8.224 | 14.076 | 18.098 | 7.348 | 8.887 | 0.223 | 67.345 | 2.146 | | 1 | 11 | 1068.011 | 22 | 47.692 | 0.206 | 8.146 | 36.242 | 18.622 | 12.859 | 13.487 | 0.223 | 67.345 | 2.146 |
| mlbicgstab | | 2 | 8 | 849.447 | 23 | 35.724 | 0.207 | 8.245 | 15.379 | 18.186 | 7.250 | 8.772 | 0.223 | 77.699 | 2.470 | | 2 | 8 | 1136.754 | 23 | 48.189 | 0.207 | 8.154 | 38.251 | 18.706 | 13.253 | 13.440 | 0.223 | 77.699 | 2.470 |
| mlbicgstab | | 3 | 6 | 810.947 | 22 | 35.277 | 0.207 | 8.315 | 14.265 | 18.463 | 7.166 | 7.764 | 0.222 | 88.053 | 2.794 | | 3 | 6 | 1097.191 | 22 | 48.228 | 0.208 | 8.222 | 36.943 | 18.540 | 13.032 | 13.648 | 0.222 | 88.053 | 2.794 |
| $mlbicgstab_{orig}$ | | 1 | 11 | 813.698 | 22 | 35.963 | 0.208 | 8.258 | 14.983 | 18.222 | 7.301 | 8.948 | 0.223 | 67.345 | 2.146 | | 1 | 11 | 1101.862 | 22 | 48.060 | 0.206 | 8.187 | 36.598 | 18.754 | 13.128 | 13.790 | 0.222 | 67.345 | 2.146 |
| $mlbicgstab_{orig}$ | | 2 | 8 | 863.327 | 23 | 36.098 | 0.207 | 8.325 | 15.613 | 18.295 | 7.293 | 8.950 | 0.224 | 77.699 | 2.470 | | 2 | 8 | 1156.702 | 23 | 48.811 | 0.207 | 8.175 | 37.538 | 18.759 | 13.308 | 14.117 | 0.223 | 77.699 | 2.470 |
| $mlbicgstab_{orig}$ | | 3 | 6 | 816.162 | 22 | 35.282 | 0.206 | 8.227 | 13.721 | 18.765 | 7.158 | 7.803 | 0.223 | 88.053 | 2.794 | | 3 | 6 | 1101.082 | 22 | 48.193 | 0.208 | 8.161 | 36.777 | 19.209 | 12.936 | 13.566 | 0.222 | 88.053 | 2.794 |
| $mlbicgstab_{ss}$ | | 1 | 12 | 848.468 | 23 | 36.107 | 0.208 | 8.406 | 14.722 | 18.206 | 7.314 | 8.986 | 0.225 | 56.990 | 1.821 | | 1 | 11 | 1053.535 | 21 | 48.130 | 0.206 | 8.135 | 37.855 | 18.298 | 13.041 | 14.242 | 0.224 | 56.990 | 1.821 |
| $mlbicgstab_{ss}$ | | 2 | 8 | 848.190 | 23 | 35.854 | 0.206 | 8.192 | 13.792 | 18.286 | 7.321 | 8.793 | 0.220 | 64.756 | 2.065 | | 2 | 8 | 1135.345 | 23 | 48.249 | 0.207 | 8.212 | 37.223 | 18.690 | 13.102 | 13.815 | 0.221 | 64.756 | 2.065 |
| $mlbicgstab_{ss}$ | | 3 | 6 | 855.291 | 23 | 35.856 | 0.205 | 8.227 | 14.429 | 18.218 | 7.696 | 8.519 | 0.225 | 72.522 | 2.308 | | 3 | 6 | 1139.855 | 23 | 48.083 | 0.206 | 8.249 | 36.614 | 18.962 | 13.071 | 13.622 | 0.225 | 72.522 | 2.308 |
| qmr | | - | 21 | 3761.405 | 104 | 35.831 | 0.200 | 8.247 | 15.166 | 18.296 | 7.832 | 9.040 | 0.222 | 59.579 | 1.903 | | - | 21 | 5045.642 | 104 | 48.181 | 0.199 | 8.182 | 38.137 | 18.306 | 13.862 | 15.146 | 0.222 | 59.579 | 1.903 |
| $qmr_{sym}$ | | - | 20 | 751.468 | 20 | 36.590 | 0.209 | 8.335 | 14.364 | 18.100 | 7.701 | 9.103 | 0.222 | 54.402 | 1.740 | | - | 20 | 1022.777 | 20 | 50.194 | 0.209 | 8.146 | 36.956 | 18.309 | 13.998 | 15.176 | 0.223 | 54.402 | 1.740 |
| rbicgstab | | 1 | 12 | 899.051 | 24 | 36.794 | 0.207 | 8.296 | 16.707 | 18.137 | 7.901 | 9.135 | 0.223 | 54.402 | 1.740 | | 1 | 12 | 1207.598 | 24 | 49.621 | 0.208 | 8.194 | 36.749 | 18.295 | 13.972 | 14.950 | 0.225 | 54.402 | 1.740 |
| rbicgstab | | 2 | 6 | 913.274 | 24 | 37.304 | 0.208 | 8.184 | 15.640 | 18.218 | 8.103 | 9.451 | 0.224 | 59.579 | 1.903 | | 2 | 6 | 1193.684 | 24 | 48.978 | 0.207 | 8.165 | 37.640 | 18.274 | 13.490 | 14.871 | 0.224 | 59.579 | 1.903 |
| rbicgstab | | 3 | 4 | 925.326 | 24 | 37.752 | 0.207 | 8.192 | 16.300 | 18.210 | 8.218 | 9.824 | 0.224 | 64.756 | 2.065 | | 3 | 4 | 1192.294 | 24 | 48.863 | 0.207 | 8.172 | 36.346 | 18.298 | 13.485 | 14.758 | 0.224 | 64.756 | 2.065 |
| tfqmr | | - | 14 | 2015.864 | 56 | 35.548 | 0.202 | 8.236 | 17.129 | 18.157 | 7.586 | 8.873 | 0.222 | 59.579 | 1.903 | | - | 14 | 2733.836 | 56 | 48.223 | 0.201 | 8.262 | 39.780 | 18.207 | 13.792 | 14.937 | 0.223 | 59.579 | 1.903 |
| bicg | **2** | - | 20 | 2577.107 | 40 | 63.867 | 0.202 | 8.152 | 62.425 | 18.476 | 31.546 | 11.620 | 0.222 | 54.519 | 1.744 | **5** | | 20 | 2712.133 | 40 | 66.759 | 0.204 | 8.233 | 55.856 | 18.481 | 15.936 | 30.202 | 0.223 | 55.104 | 1.762 |
| $bicg_{sym}$ | | - | 20 | 1320.890 | 20 | 65.339 | 0.208 | 8.154 | 63.371 | 18.319 | 31.339 | 11.673 | 0.221 | 46.753 | 1.501 | | - | 20 | 1352.897 | 20 | 66.064 | 0.209 | 8.147 | 57.226 | 18.400 | 15.348 | 28.612 | 0.222 | 47.338 | 1.519 |
| bicgstab | | - | 12 | 1564.233 | 24 | 64.363 | 0.205 | 8.165 | 61.668 | 18.312 | 31.166 | 11.547 | 0.221 | 54.519 | 1.744 | | - | 12 | 1650.968 | 24 | 67.008 | 0.206 | 8.261 | 57.570 | 18.335 | 15.107 | 30.367 | 0.222 | 55.104 | 1.762 |
| cg | | - | 20 | 1331.669 | 20 | 65.883 | 0.207 | 8.130 | 63.650 | 18.298 | 31.754 | 11.810 | 0.221 | 46.753 | 1.501 | | - | 20 | 1369.767 | 20 | 67.026 | 0.210 | 8.334 | 56.863 | 18.299 | 15.209 | 29.797 | 0.222 | 47.338 | 1.519 |
| cgs | | - | 13 | 1685.797 | 26 | 64.163 | 0.205 | 8.234 | 62.021 | 18.379 | 31.311 | 11.320 | 0.221 | 54.519 | 1.744 | | - | 13 | 1785.074 | 26 | 67.114 | 0.207 | 8.216 | 57.239 | 18.451 | 15.505 | 29.891 | 0.222 | 55.104 | 1.762 |
| mlbicgstab | | 1 | 12 | 1548.797 | 24 | 63.555 | 0.205 | 8.130 | 62.924 | 18.341 | 30.392 | 11.408 | 0.222 | 67.462 | 2.149 | | 1 | 11 | 1499.735 | 22 | 66.314 | 0.207 | 8.190 | 58.407 | 18.338 | 15.426 | 28.883 | 0.223 | 68.046 | 2.168 |
| mlbicgstab | | 2 | 8 | 1526.438 | 23 | 64.963 | 0.207 | 8.196 | 63.494 | 18.495 | 31.465 | 11.297 | 0.222 | 77.816 | 2.474 | | 2 | 8 | 1582.510 | 23 | 66.625 | 0.207 | 8.216 | 55.299 | 18.323 | 15.457 | 29.480 | 0.224 | 78.401 | 2.492 |
| mlbicgstab | | 3 | 6 | 1457.464 | 22 | 64.467 | 0.207 | 8.480 | 59.957 | 18.363 | 31.321 | 10.983 | 0.224 | 88.170 | 2.798 | | 3 | 6 | 1515.362 | 22 | 66.266 | 0.207 | 8.225 | 57.282 | 18.555 | 15.376 | 28.763 | 0.223 | 88.755 | 2.816 |
| $mlbicgstab_{orig}$ | | 1 | 11 | 1447.734 | 22 | 64.615 | 0.206 | 8.168 | 62.461 | 18.419 | 30.978 | 11.577 | 0.223 | 67.462 | 2.149 | | 1 | 12 | 1665.866 | 24 | 67.217 | 0.208 | 8.316 | 58.857 | 18.424 | 15.550 | 29.506 | 0.224 | 68.046 | 2.168 |
| $mlbicgstab_{orig}$ | | 2 | 8 | 1525.665 | 23 | 64.667 | 0.206 | 8.323 | 62.946 | 18.480 | 31.236 | 11.213 | 0.222 | 77.816 | 2.474 | | 2 | 8 | 1586.031 | 23 | 66.426 | 0.207 | 8.218 | 57.424 | 18.440 | 15.359 | 29.342 | 0.224 | 78.401 | 2.492 |
| $mlbicgstab_{orig}$ | | 3 | 6 | 1482.136 | 22 | 65.395 | 0.207 | 8.187 | 62.796 | 18.464 | 31.775 | 10.895 | 0.222 | 88.170 | 2.798 | | 3 | 6 | 1526.734 | 22 | 66.460 | 0.208 | 8.176 | 57.916 | 18.667 | 15.314 | 28.832 | 0.223 | 88.755 | 2.816 |
| $mlbicgstab_{ss}$ | | 1 | 12 | 1498.264 | 23 | 64.194 | 0.206 | 8.136 | 62.432 | 18.394 | 31.255 | 11.019 | 0.224 | 57.107 | 1.825 | | 1 | 12 | 1576.212 | 23 | 66.624 | 0.206 | 8.199 | 57.626 | 18.375 | 15.455 | 29.431 | 0.225 | 57.692 | 1.843 |
| $mlbicgstab_{ss}$ | | 2 | 8 | 1509.350 | 23 | 64.387 | 0.206 | 8.227 | 61.920 | 18.417 | 31.284 | 11.208 | 0.221 | 64.873 | 2.068 | | 2 | 8 | 1588.541 | 23 | 66.936 | 0.206 | 8.238 | 58.181 | 18.438 | 15.372 | 29.806 | 0.221 | 65.458 | 2.087 |
| $mlbicgstab_{ss}$ | | 3 | 6 | 1506.039 | 23 | 63.937 | 0.206 | 8.252 | 59.060 | 18.455 | 30.865 | 10.906 | 0.226 | 72.639 | 2.312 | | 3 | 6 | 1596.961 | 23 | 67.014 | 0.206 | 8.185 | 59.225 | 18.441 | 15.696 | 29.458 | 0.225 | 73.224 | 2.330 |
| qmr | | - | 21 | 6566.389 | 104 | 62.751 | 0.197 | 8.308 | 62.246 | 18.432 | 31.566 | 11.619 | 0.221 | 59.696 | 1.906 | | - | 21 | 6877.989 | 104 | 65.400 | 0.199 | 8.251 | 58.404 | 18.450 | 16.187 | 29.680 | 0.222 | 60.281 | 1.925 |
| $qmr_{sym}$ | | - | 20 | 1320.745 | 20 | 65.008 | 0.208 | 8.151 | 61.575 | 18.440 | 31.193 | 11.450 | 0.222 | 54.519 | 1.744 | | - | 20 | 1378.235 | 20 | 67.044 | 0.207 | 8.264 | 56.590 | 18.417 | 15.306 | 29.413 | 0.223 | 55.104 | 1.762 |
| rbicgstab | | 1 | 12 | 1572.078 | 24 | 64.583 | 0.205 | 8.198 | 64.605 | 18.304 | 31.341 | 11.332 | 0.223 | 54.519 | 1.744 | | 1 | 12 | 1626.039 | 24 | 66.134 | 0.207 | 8.182 | 56.731 | 18.448 | 15.283 | 29.197 | 0.225 | 55.104 | 1.762 |
| rbicgstab | | 2 | 6 | 1575.561 | 24 | 64.776 | 0.206 | 8.149 | 62.919 | 18.322 | 31.319 | 11.635 | 0.224 | 59.696 | 1.906 | | 2 | 6 | 1625.443 | 24 | 65.866 | 0.207 | 8.243 | 57.427 | 18.426 | 14.947 | 29.307 | 0.225 | 60.281 | 1.925 |
| rbicgstab | | 3 | 4 | 1581.284 | 24 | 64.908 | 0.205 | 8.101 | 62.708 | 18.474 | 31.616 | 11.406 | 0.224 | 64.873 | 2.068 | | 3 | 4 | 1620.214 | 24 | 65.675 | 0.205 | 8.208 | 57.653 | 18.432 | 15.530 | 28.539 | 0.224 | 65.458 | 2.087 |
| tfqmr | | - | 14 | 3559.857 | 56 | 63.006 | 0.199 | 8.171 | 64.935 | 18.387 | 31.368 | 11.520 | 0.223 | 59.696 | 1.906 | | - | 14 | 3697.447 | 56 | 64.775 | 0.200 | 8.270 | 57.805 | 18.316 | 15.680 | 29.005 | 0.223 | 60.281 | 1.925 |

Figure D.5: The performance, using 32 nodes on WALTON, of the full MPI-based distributed-memory system as a function of the distributed-transpose algorithm employed to support the DFT kernel, the iterative scheme, and where applicable, the number of starting vectors.

| Scheme | TR | SV | NI | CT (s) | MVPs | MVP (s) | VI (s) | TDFT (s) | TFTR (s) | CORE (s) | VFTR (s) | VRTR (s) | NDDC (s) | Memory Total (GB) | Memory Node (GB) | TR | SV | NI | CT (s) | MVPs | MVP (s) | VI (s) | TDFT (s) | TFTR (s) | CORE (s) | VFTR (s) | VRTR (s) | NDDC (s) | Memory Total (GB) | Memory Node (GB) |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| bicg | **0** | - | 20 | 1791.372 | 40 | 44.278 | 0.198 | 8.886 | 28.590 | 16.496 | 12.407 | 14.024 | 0.216 | 108.857 | 1.744 | **3** | | 20 | 3369.317 | 40 | 83.694 | 0.198 | 8.856 | 87.539 | 16.999 | 31.468 | 32.141 | 0.217 | 109.352 | 1.750 |
| $bicg_{sym}$ | | - | 20 | 915.674 | 20 | 45.128 | 0.204 | 8.900 | 29.513 | 16.450 | 12.780 | 13.498 | 0.215 | 93.689 | 1.507 | | - | 20 | 1741.557 | 20 | 86.407 | 0.204 | 8.931 | 87.062 | 16.618 | 32.393 | 31.928 | 0.216 | 94.184 | 1.512 |
| bicgstab | | - | 12 | 1106.207 | 24 | 45.395 | 0.201 | 8.984 | 30.420 | 16.566 | 12.835 | 13.890 | 0.216 | 108.857 | 1.744 | | - | 12 | 2078.162 | 24 | 85.790 | 0.202 | 8.958 | 88.195 | 16.825 | 32.560 | 31.936 | 0.217 | 109.352 | 1.750 |
| cg | | - | 20 | 923.414 | 20 | 45.482 | 0.204 | 8.995 | 29.937 | 16.462 | 12.710 | 13.783 | 0.214 | 93.689 | 1.507 | | - | 20 | 1726.531 | 20 | 85.598 | 0.204 | 8.850 | 87.344 | 16.641 | 31.887 | 31.840 | 0.217 | 94.184 | 1.512 |
| cgs | | - | 12 | 1124.320 | 24 | 46.266 | 0.201 | 8.818 | 29.183 | 16.424 | 13.475 | 14.323 | 0.216 | 108.857 | 1.744 | | - | 12 | 2078.192 | 24 | 85.806 | 0.202 | 8.846 | 86.981 | 16.881 | 32.046 | 32.093 | 0.216 | 109.352 | 1.750 |
| mlbicgstab | | 1 | 11 | 1036.432 | 22 | 45.899 | 0.202 | 8.846 | 28.897 | 16.538 | 12.820 | 14.511 | 0.216 | 134.136 | 2.141 | | 1 | 11 | 1907.015 | 22 | 85.583 | 0.202 | 8.901 | 88.908 | 16.937 | 32.075 | 31.871 | 0.216 | 134.136 | 2.141 |
| mlbicgstab | | 2 | 8 | 1071.030 | 23 | 45.152 | 0.202 | 8.918 | 28.863 | 16.524 | 12.797 | 13.772 | 0.216 | 154.360 | 2.458 | | 2 | 8 | 1984.211 | 23 | 84.833 | 0.201 | 8.834 | 83.373 | 17.036 | 31.658 | 31.705 | 0.216 | 154.360 | 2.458 |
| mlbicgstab | | 3 | 6 | 1022.947 | 22 | 44.890 | 0.203 | 8.896 | 28.257 | 16.839 | 12.344 | 13.592 | 0.218 | 174.584 | 2.775 | | 3 | 6 | 1875.292 | 22 | 83.626 | 0.202 | 8.837 | 77.067 | 17.034 | 31.112 | 31.055 | 0.218 | 175.079 | 2.780 |
| $mlbicgstab_{orig}$ | | 1 | 11 | 1035.631 | 22 | 46.015 | 0.202 | 9.010 | 30.017 | 16.523 | 13.046 | 14.205 | 0.216 | 134.136 | 2.141 | | 1 | 11 | 1900.736 | 22 | 85.332 | 0.201 | 8.895 | 88.956 | 16.970 | 32.239 | 31.243 | 0.216 | 134.136 | 2.141 |
| $mlbicgstab_{orig}$ | | 2 | 8 | 1081.135 | 23 | 45.550 | 0.201 | 8.805 | 30.832 | 16.492 | 12.814 | 14.100 | 0.216 | 154.360 | 2.458 | | 2 | 8 | 1980.338 | 23 | 84.580 | 0.202 | 8.841 | 81.304 | 17.064 | 31.408 | 31.759 | 0.217 | 154.360 | 2.458 |
| $mlbicgstab_{orig}$ | | 3 | 6 | 1037.554 | 22 | 45.304 | 0.200 | 8.844 | 28.604 | 16.847 | 12.689 | 13.668 | 0.217 | 174.584 | 2.775 | | 3 | 6 | 1883.790 | 22 | 83.788 | 0.204 | 8.916 | 77.784 | 17.029 | 31.196 | 31.175 | 0.219 | 175.079 | 2.780 |
| $mlbicgstab_{ss}$ | | 1 | 12 | 1054.097 | 23 | 45.018 | 0.202 | 8.926 | 29.582 | 16.485 | 12.495 | 13.945 | 0.217 | 113.913 | 1.824 | | 1 | 12 | 1985.421 | 23 | 85.275 | 0.201 | 8.912 | 85.667 | 16.875 | 32.152 | 31.718 | 0.217 | 113.913 | 1.824 |
| $mlbicgstab_{ss}$ | | 2 | 8 | 1089.979 | 23 | 46.217 | 0.201 | 8.848 | 31.168 | 16.457 | 13.091 | 14.433 | 0.217 | 129.080 | 2.061 | | 2 | 8 | 1952.675 | 23 | 83.754 | 0.200 | 8.869 | 81.721 | 16.944 | 31.355 | 30.914 | 0.217 | 129.080 | 2.061 |
| $mlbicgstab_{ss}$ | | 3 | 6 | 1062.063 | 23 | 44.723 | 0.200 | 8.860 | 29.339 | 16.530 | 12.448 | 13.627 | 0.217 | 144.248 | 2.299 | | 3 | 6 | 1970.937 | 23 | 84.259 | 0.200 | 8.866 | 77.889 | 17.047 | 31.589 | 31.345 | 0.218 | 144.743 | 2.305 |
| qmr | | - | 21 | 4655.116 | 104 | 44.417 | 0.195 | 8.826 | 28.669 | 16.457 | 12.919 | 14.268 | 0.217 | 118.969 | 1.903 | | - | 21 | 8693.455 | 104 | 83.227 | 0.194 | 8.840 | 89.393 | 16.876 | 32.256 | 32.725 | 0.217 | 119.464 | 1.909 |
| $qmr_{sym}$ | | - | 20 | 937.008 | 20 | 45.833 | 0.204 | 8.881 | 29.310 | 16.413 | 12.841 | 14.160 | 0.216 | 108.857 | 1.744 | | - | 20 | 1737.189 | 20 | 85.840 | 0.204 | 8.905 | 89.501 | 16.640 | 32.137 | 31.723 | 0.215 | 109.352 | 1.750 |
| rbicgstab | | 1 | 14 | 1293.468 | 28 | 45.449 | 0.200 | 8.806 | 29.373 | 16.430 | 12.912 | 14.231 | 0.217 | 108.857 | 1.744 | | 1 | 14 | 2384.962 | 28 | 84.410 | 0.200 | 8.818 | 88.290 | 16.892 | 31.663 | 31.934 | 0.217 | 108.857 | 1.744 |
| rbicgstab | | 2 | 6 | 1112.964 | 24 | 45.400 | 0.201 | 8.826 | 30.341 | 16.451 | 12.823 | 14.007 | 0.217 | 118.969 | 1.903 | | 2 | 6 | 2057.166 | 24 | 84.906 | 0.201 | 8.878 | 87.372 | 16.865 | 31.688 | 31.916 | 0.218 | 118.969 | 1.903 |
| rbicgstab | | 3 | 5 | 1369.739 | 30 | 44.804 | 0.199 | 8.826 | 29.526 | 16.451 | 12.574 | 14.071 | 0.217 | 129.080 | 2.061 | | 3 | 5 | 2561.496 | 30 | 84.511 | 0.199 | 8.855 | 88.661 | 16.862 | 31.543 | 32.412 | 0.217 | 129.576 | 2.067 |
| tfqmr | | - | 14 | 2515.782 | 56 | 44.445 | 0.195 | 8.883 | 29.374 | 16.463 | 12.768 | 14.004 | 0.216 | 118.969 | 1.903 | | - | 14 | 4733.170 | 56 | 84.018 | 0.196 | 8.941 | 84.685 | 16.787 | 32.632 | 32.504 | 0.216 | 119.464 | 1.909 |
| bicg | **2** | - | 20 | 3401.904 | 40 | 84.487 | 0.199 | 8.820 | 94.778 | 16.694 | 52.140 | 12.557 | 0.218 | 109.352 | 1.750 | **5** | | 20 | 3515.997 | 40 | 84.582 | 0.199 | 8.884 | 96.245 | 16.841 | 14.913 | 49.781 | 0.218 | 111.829 | 1.779 |
| $bicg_{sym}$ | | - | 20 | 1784.728 | 20 | 88.549 | 0.204 | 8.838 | 100.753 | 16.624 | 53.098 | 12.914 | 0.216 | 94.184 | 1.512 | | - | 20 | 1785.086 | 20 | 82.722 | 0.204 | 8.875 | 96.419 | 16.825 | 12.013 | 48.001 | 0.217 | 96.661 | 1.541 |
| bicgstab | | - | 12 | 2059.838 | 24 | 85.067 | 0.201 | 8.858 | 96.717 | 16.669 | 50.795 | 12.772 | 0.216 | 109.352 | 1.750 | | - | 12 | 2108.224 | 24 | 81.190 | 0.201 | 8.905 | 112.882 | 16.801 | 12.241 | 45.895 | 0.218 | 111.829 | 1.779 |
| cg | | - | 20 | 1772.861 | 20 | 87.888 | 0.204 | 8.819 | 95.827 | 16.638 | 52.495 | 13.067 | 0.215 | 94.184 | 1.512 | | - | 20 | 1794.172 | 20 | 83.700 | 0.204 | 8.900 | 115.167 | 16.830 | 12.230 | 47.656 | 0.218 | 96.661 | 1.541 |
| cgs | | - | 12 | 2073.389 | 24 | 85.738 | 0.204 | 8.869 | 97.719 | 16.741 | 51.362 | 12.767 | 0.217 | 109.352 | 1.750 | | - | 12 | 2128.581 | 24 | 82.932 | 0.202 | 8.888 | 95.555 | 16.843 | 12.242 | 48.183 | 0.218 | 111.829 | 1.779 |
| mlbicgstab | | 1 | 11 | 1938.669 | 22 | 87.177 | 0.201 | 8.862 | 101.290 | 16.713 | 52.193 | 12.523 | 0.216 | 134.632 | 2.146 | | 1 | 11 | 2213.788 | 22 | 86.213 | 0.203 | 8.930 | 96.846 | 16.889 | 12.278 | 52.165 | 0.217 | 137.108 | 2.175 |
| mlbicgstab | | 2 | 8 | 1991.931 | 23 | 85.298 | 0.202 | 8.838 | 92.628 | 16.972 | 51.140 | 12.171 | 0.216 | 154.855 | 2.463 | | 2 | 8 | 2051.016 | 23 | 82.611 | 0.202 | 8.845 | 96.798 | 16.849 | 12.190 | 47.646 | 0.217 | 157.332 | 2.492 |
| mlbicgstab | | 3 | 6 | 1933.922 | 22 | 86.226 | 0.202 | 8.879 | 97.317 | 16.930 | 51.742 | 12.300 | 0.218 | 175.079 | 2.780 | | 3 | 6 | 2075.538 | 22 | 86.615 | 0.202 | 8.880 | 104.831 | 16.899 | 12.179 | 51.705 | 0.219 | 177.556 | 2.809 |
| $mlbicgstab_{orig}$ | | 1 | 11 | 1950.946 | 22 | 86.467 | 0.202 | 8.911 | 104.071 | 16.727 | 51.843 | 12.695 | 0.216 | 134.632 | 2.146 | | 1 | 11 | 2009.854 | 22 | 84.129 | 0.201 | 8.928 | 99.064 | 16.867 | 12.197 | 48.733 | 0.217 | 137.108 | 2.175 |
| $mlbicgstab_{orig}$ | | 2 | 8 | 2033.678 | 23 | 86.944 | 0.202 | 8.846 | 98.160 | 16.934 | 52.222 | 12.578 | 0.217 | 154.855 | 2.463 | | 2 | 8 | 2147.696 | 23 | 85.313 | 0.201 | 8.819 | 101.188 | 16.845 | 12.201 | 51.037 | 0.218 | 157.332 | 2.492 |
| $mlbicgstab_{orig}$ | | 3 | 6 | 1935.146 | 22 | 86.063 | 0.203 | 8.846 | 91.972 | 16.898 | 51.777 | 12.199 | 0.218 | 175.079 | 2.780 | | 3 | 6 | 1943.250 | 22 | 81.459 | 0.202 | 8.919 | 98.987 | 16.891 | 12.261 | 46.979 | 0.219 | 177.556 | 2.809 |
| $mlbicgstab_{ss}$ | | 1 | 12 | 2013.920 | 23 | 86.558 | 0.201 | 8.907 | 98.863 | 16.723 | 51.448 | 12.797 | 0.217 | 114.408 | 1.829 | | 1 | 12 | 2070.494 | 23 | 83.759 | 0.200 | 8.896 | 97.188 | 16.859 | 12.373 | 48.728 | 0.219 | 116.885 | 1.858 |
| $mlbicgstab_{ss}$ | | 2 | 8 | 2025.697 | 23 | 86.886 | 0.201 | 8.821 | 98.919 | 16.743 | 52.363 | 12.498 | 0.217 | 129.576 | 2.067 | | 2 | 8 | 2052.418 | 23 | 83.598 | 0.200 | 8.853 | 100.031 | 16.813 | 12.187 | 48.797 | 0.219 | 132.052 | 2.096 |
| $mlbicgstab_{ss}$ | | 3 | 6 | 2002.217 | 23 | 85.552 | 0.201 | 8.863 | 93.419 | 16.958 | 51.320 | 12.265 | 0.218 | 144.743 | 2.305 | | 3 | 6 | 2089.017 | 23 | 84.270 | 0.201 | 8.858 | 117.716 | 16.822 | 12.208 | 49.209 | 0.220 | 147.220 | 2.333 |
| qmr | | - | 21 | 8736.704 | 104 | 83.644 | 0.195 | 8.870 | 98.841 | 16.715 | 52.531 | 12.930 | 0.217 | 119.464 | 1.909 | | - | 21 | 8717.898 | 104 | 81.599 | 0.195 | 8.994 | 104.556 | 16.862 | 14.914 | 48.312 | 0.218 | 121.941 | 1.937 |
| $qmr_{sym}$ | | - | 20 | 1756.348 | 20 | 86.812 | 0.204 | 8.932 | 96.464 | 16.656 | 51.637 | 12.796 | 0.217 | 109.352 | 1.750 | | - | 20 | 1819.886 | 20 | 84.930 | 0.203 | 8.874 | 96.443 | 16.843 | 12.321 | 50.080 | 0.217 | 111.829 | 1.779 |
| rbicgstab | | 1 | 14 | 2407.719 | 28 | 85.256 | 0.200 | 8.777 | 97.908 | 16.723 | 51.668 | 12.618 | 0.218 | 109.352 | 1.750 | | 1 | 14 | 2443.222 | 28 | 80.980 | 0.200 | 8.913 | 105.318 | 16.882 | 12.161 | 47.437 | 0.219 | 111.829 | 1.779 |
| rbicgstab | | 2 | 6 | 2060.978 | 24 | 85.020 | 0.202 | 8.822 | 96.408 | 16.674 | 50.932 | 12.542 | 0.219 | 119.464 | 1.909 | | 2 | 6 | 2137.917 | 24 | 83.323 | 0.201 | 8.844 | 100.795 | 16.849 | 12.151 | 48.684 | 0.220 | 121.941 | 1.937 |
| rbicgstab | | 3 | 5 | 2569.935 | 30 | 84.787 | 0.199 | 8.808 | 100.244 | 16.663 | 51.399 | 12.654 | 0.217 | 129.576 | 2.067 | | 3 | 5 | 2693.992 | 30 | 83.376 | 0.198 | 8.847 | 102.480 | 16.810 | 12.116 | 49.534 | 0.219 | 132.052 | 2.096 |
| tfqmr | | - | 14 | 4760.389 | 56 | 84.502 | 0.196 | 8.961 | 97.565 | 16.699 | 52.303 | 12.936 | 0.216 | 119.464 | 1.909 | | - | 14 | 4981.104 | 56 | 84.016 | 0.196 | 8.898 | 109.934 | 16.879 | 13.477 | 51.089 | 0.218 | 121.941 | 1.937 |

Figure D.6: The performance, using 64 nodes on WALTON, of the full MPI-based distributed-memory system as a function of the distributed-transpose algorithm employed to support the DFT kernel, the iterative scheme, and where applicable, the number of starting vectors.