Development and Application of Numerical Methods in Biomolecular Solvation

by

Leighton W. Wilson

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Applied and Interdisciplinary Mathematics and Scientific Computing)
in the University of Michigan
2021

Doctoral Committee:

Professor Robert Krasny, Co-Chair
Professor Paul Zimmerman, Co-Chair
Professor Silas Alben
Professor Shravan Veerapaneni

Leighton W. Wilson

lwwilson@umich.edu

ORCID iD: 0000-0003-1676-8156

© Leighton W. Wilson 2021

DEDICATION

Dedicated to Leighton Clyde and Mary Sula Wilson, my grandparents.

"The sun's a desperate star that burns like every single one before."

ACKNOWLEDGMENTS

This work would not have been possible without the instruction, mentorship, support, and friendship of countless people. First, I would like to thank Robert Krasny, whose support and guidance has been critical in my professional and personal development over the last half decade. I will forever be grateful that he took a chance on me as a student. I am also grateful to Paul Zimmerman, for warmly inviting me into his group and affording me the opportunity to deepen my knowledge of computational chemistry. I also thank my committee members, Silas Alben and Shravan Veerapaneni for their feedback and input on this project. I have deeply valued the expertise of my external collaborators Nathan Baker, Weihua Geng, and Tyler Luchko. Karen Smith, Teresa Stokes, and the administrative staff of the Math Department have always been caring, encouraging, and helpful, and I thank them for their support. The Advanced Research Computing and Michigan Institute for Computational Discovery and Engineering support staff has also been deeply helpful.

Several funding sources have supported me over the course of this research, including National Science Foundation grant DMS-1819094, Extreme Science and Engineering Discovery Environment (XSEDE) grants ACI-1548562 and ASC-190062. I have additionally been supported by the Department of Defense through the National Defense Science and Engineering Graduate Fellowship and the Rackham Graduate School through the Rackham Predoctoral Fellowship.

Nathan Vaughn has been my chief academic partner on this expedition, and I'm deeply appreciative of the hours and hours we spent coding, barbecuing, and drinking coffee together. Our graduate school journeys have been deeply linked and our dissertations tied at the hip. I'm very proud of what we accomplished together. Khoi Dang has also been an important part of my time in graduate school. I've deeply enjoyed not only our collaborative projects, but our many conversations about science, technology, and our research,

My undergraduate advisor Shan Zhao, in my freshmen year of college, gave me an opportunity to explore research in mathematics. I (almost) never looked back. Many teachers throughout the years cultivated my interest in science and mathematics, including Thomas Abney, Marilyn Niemann, Debbie Anderson, Rosemary Ham, and Mary Clyde Prichard. Last but not least, I thank Shilpa, Rooney, and my family for their love and support.

TABLE OF CONTENTS

Dedication . . . . **ii**

Acknowledgments . . . . **iii**

List of Figures . . . . **vii**

List of Tables . . . . **xiv**

List of Algorithms . . . . **xvi**

List of Appendices . . . . **xvii**

List of Acronyms . . . . **xviii**

List of Symbols . . . . **xx**

Abstract . . . . **xxiii**

Chapter

1 Introduction **1**

1.1 Overview of fast summation methods and early results 1
1.2 BLDTT: GPU-accelerated barycentric Lagrange dual tree traversal 3
1.3 A Poisson–Boltzmann equation solver 3
1.4 Treecode acceleration of 3D reference interaction site model 4

2 Overview of Fast Summation Methods and Early Results **6**

2.1 Background 6
2.2 Treecodes 7
2.2.1 Tree construction 8
2.2.2 Particle-cluster treecodes 9
2.2.3 Cluster-particle treecodes 10
2.2.4 Recurrence relations for Taylor coefficients 13
2.3 Early projects: Computational improvements to treecodes 15
2.3.1 Tree virtualization and boundary grids 15
2.3.2 Flattening coefficient arrays 15
2.3.3 Results 16

3 BLDTT: GPU-Accelerated Barycentric Lagrange Dual Tree Traversal . . **23**

3.1 Background 23

3.1.1 Dual tree traversal methods 23
3.1.2 Kernel-independent methods 24
3.1.3 GPU-based compute for fast summation methods 25
3.1.4 Barycentric Lagrange interpolation 28
3.1.5 Present work 30
3.2 Description of BLDTT fast summation method 30
3.2.1 Algorithm overview 30
3.2.2 Tree building 31
3.2.3 Four types of interactions 32
3.2.4 Upward pass 34
3.2.5 Dual tree traversal 35
3.2.6 Downward pass 37
3.2.7 Derivation of upward and downward passes 38
3.2.8 Description of BLTC 41
3.3 BLDTT implementation 41
3.3.1 Computing interaction lists 41
3.3.2 MPI distributed memory parallelization 42
3.3.3 GPU details 42
3.4 Methodology 43
3.5 Results 44
3.5.1 Problem size scaling 44
3.5.2 GPU acceleration of BLDTT 45
3.5.3 Non-uniform particle distributions 46
3.5.4 Non-cubic particle domains 49
3.5.5 Unequal targets and sources 49
3.5.6 Other interaction kernels 50
3.5.7 MPI strong scaling 51

4 A Poisson–Boltzmann Equation Solver 55

4.1 Background 55
4.1.1 The Poisson–Boltzmann equation 55
4.1.2 Treecode-accelerated boundary integral PB solver (TABI-PB) . . . 57
4.2 Project 1: Comparison of molecular surface triangulation codes 58
4.2.1 Project description 58
4.2.2 Methodology 59
4.2.3 Results 61
4.3 Project 2: Implementation of node patch method 69
4.3.1 Project description 69
4.3.2 Methodology 69
4.3.3 Results 69
4.4 Project 3: GPU-accelerated BLDTT TABI-PB 72
4.4.1 Project description 72
4.4.2 Methodology 73
4.4.3 Results 74
4.5 Application: Electrostatic binding free energy calculation 79

4.5.1 Project description 79
4.5.2 Methodology 81
4.5.3 Results 83

5 Treecode Acceleration of the 3D Reference Interaction Site Model 88

5.1 Background 88
5.1.1 Overview 88
5.1.2 Liquid integral equation theory 89
5.1.3 1D reference interaction site model (1D-RISM) 91
5.1.4 3D reference interaction site model (3D-RISM) 92
5.1.5 Long range components of correlation functions 92
5.1.6 The RISM procedure and its implementation 94
5.2 Project 1: Implementing treecodes for asymptotic correlation functions . . 95
5.2.1 Project description 95
5.2.2 Methodology 97
5.2.3 Results 100
5.3 Project 2: GPU-accelerated BLDTT 3D-RISM 113
5.3.1 Project description 113
5.3.2 Methodology and results 113

6 Conclusion 116

6.1 BLDTT 117
6.1.1 Present work 117
6.1.2 Future work 117
6.2 TABI-PB 118
6.2.1 Present work 118
6.2.2 Future work 119
6.3 RISM 123
6.3.1 Present work 123
6.3.2 Future work 123

Appendices 125

Bibliography 142

LIST OF FIGURES

FIGURE

2.1 A particle-cluster interaction between a target particle $\mathbf{x}_i$ and a source cluster C with center $\mathbf{y}_c$ and radius r containing particles $\mathbf{y}_j$ with charge q_j. The particle-cluster distance is $R = |\mathbf{x}_i - \mathbf{y}_c|$. 9

2.2 A cluster-particle interaction between a target cluster C with center $\mathbf{x}_c$ and radius r containing particles $\mathbf{x}_i$ and a source particle $\mathbf{y}_j$ with charge q_j. The cluster-particle distance is $R = |\mathbf{y}_j - \mathbf{x}_c|$. 12

2.3 Volume grid test case, 1E5 sources with 64E6 targets on a regular grid, relative ℓ_2 (solid, $\circ$) and ℓ_∞ (dotted, $\triangledown$) errors from exact solution versus CPU time for original explicit treecode (green), virtual treecode (blue), and virtual treecode with flattened coefficient arrays (red), for various treecode expansion orders from 1 to 15, increasing from right to left, and multipole acceptance criterion of (a) $\theta = 0.3$ and (b) $\theta = 0.8$. Corresponding direct calculation time was 9.43E4 s. Simulations ran in serial on Intel Xeon CPU. 17

2.4 Volume grid test case, 1E5 sources with 64E6 targets on a regular grid, relative ℓ_2 (solid, $\circ$) and ℓ_∞ (dotted, $\triangledown$) errors versus total memory usage for original explicit treecode (green), virtual treecode (blue), and virtual treecode with flattened coefficient arrays (red), for treecode expansion order from 1 to 15, increasing from right to left, and multipole acceptance criterion of (a) $\theta = 0.3$ and (b) $\theta = 0.8$. Corresponding direct calculation memory usage was 2.01 GB with real targets (black, dash-dotted) and 0.51 GB with virtual targets (black, dashed). Simulations ran in serial on Intel Xeon CPU. 18

2.5 Boundary grid test case, 1E5 sources with targets on the six faces of a $1000 \times 1000 \times 1000$ point grid, L_2 (solid, $\circ$) and L_∞ (dotted, $\triangledown$) errors from exact solution versus CPU time for original explicit treecode (green), virtual 2D treecode (blue), and virtual 2D treecode with flattened coefficient arrays (red), for treecode expansion order from 1 to 15, increasing from right to left, and multipole acceptance criterion of (a) $\theta = 0.3$ and (b) $\theta = 0.8$. Corresponding direct calculation time (black, dash-dotted) was 8.83E3 s. Simulations ran in serial on Intel Xeon CPU. 18

2.6 Boundary grid test case, 1E5 sources with targets on the six faces of a $1000 \times 1000 \times 1000$ point grid, L_2 (solid, $\circ$) and L_∞ (dotted, $\triangledown$) errors from exact solution versus total memory usage for original explicit treecode (green), virtual 2D treecode (blue), and virtual 2D treecode with flattened coefficient arrays (red), for treecode expansion order from 1 to 15, increasing from right to left, and multipole acceptance criterion of (a) $\theta = 0.3$ and (b) $\theta = 0.8$. Corresponding direct calculation memory usage was 0.196 GB with real targets (black, dash-dotted) and 0.056 GB with virtual targets (black, dashed). Simulations ran in serial on Intel Xeon CPU. 19
2.7 Partition strategies for randomly distributed sources and grid-placed targets. Sources are represented by red markers, targets by blue markers. (a) depicts an initial uniform random distribution of sources and uniform grid of targets, (b) depicts a disjoint partitioning strategy across four processors, and (c) depicts a global partitioning strategy across four processors. 20
2.8 Volume grid test case, 1E5 sources with 64E6 targets on a regular grid, number of processors versus (a) total time and (b) speedup for direct calculation (black, dashed, $\circ$), original explicit treecode with real targets and 3D coefficient arrays (green), and virtual treecode with flattened coefficient arrays (red), using parallelization strategies depicted in Fig. 2.7(b) (dashed, $\circ$) and Fig. 2.7(c) (dash-dotted, $\triangledown$), treecode parameters $\theta = 0.3$ and $p = 12$. Relative ℓ_2 error approximately 4E–10. Simulations ran on Intel Xeon CPUs. 21
2.9 Boundary grid test case, 1E5 sources with targets on the six faces of a $1000 \times 1000 \times 1000$ point grid, number of processors versus (a) total time and (b) speedup for direct calculation (black, dashed, $\circ$), original explicit treecode with real targets and 3D coefficient arrays (green), and 2D virtual treecode with flattened coefficient arrays (red), using parallelization strategies depicted in Fig. 2.7(b) (dashed, $\circ$) and Fig. 2.7(c) (dash-dotted, $\triangledown$), using parallelization strategies depicted in Fig. 2.7(b) (dashed, $\circ$) and Fig. 2.7(c) (dash-dotted, $\triangledown$), treecode parameters $\theta = 0.3$ and $p = 12$. Simulations ran on Intel Xeon CPUs. 22

3.1 Structure of the standard configuration Kepler GK110. 15 SMs have access to a device-wide L2 cache and global GPU memory through memory controllers. The GPU communicates with the host machine through a PCI interface. 27
3.2 Structure of an SM on the Kepler GK110. The SM contains 192 cores which each have access to SM-wide shared memory, L1 cache, and read-only memory. . . . 28
3.3 Basic memory hierarchy of the Kepler GK110. The device-wide global memory is accessed by individual threads through the hierarchy of device-wide L2 cache and SM-wide L1 cache. The SMs can be configured to adjust the split between the shared memory and L1 cache. Greater shared memory gives more user control over memory optimization, but at the expense of bandwidth for global memory fetches. 29

3.4 Four types of interactions are used, in each case the target cluster C_t on the left interacts with the source cluster C_s on the right, (a) direct particle-particle interaction (PP), (b) particle-cluster approximation (PC), (c) cluster-particle approximation (CP), (d) cluster-cluster approximation (CC), dots are target/source particles $\mathbf{x}_i, \mathbf{y}_j$, crosses are target/source proxy particles $\mathbf{t}_{\boldsymbol{\ell}}, \mathbf{s}_{\mathbf{k}}$, target/source cluster radii r_t, r_s, target-source cluster distance R. 33
3.5 Comparison of BLTC and BLDTT, compute time (s) versus number of particles N=1E5, 1E6, 1E7, 1E8, random uniformly distributed particles in $[-1,1]^3$ interacting by the Coulomb kernel, MAC parameter $\theta = 0.7$, degree $n = 8$ yielding 7-8 digit accuracy, direct sum (green), BLTC (red), BLDTT (blue), (a) linear scale, (b) logarithmic scale, scaling lines $O(N^2)$ (dash-dotted), $O(N \log N)$ (dotted), $O(N)$ (dashed), simulations ran on one NVIDIA P100 GPU. 45
3.6 Sample random distributions with N=4E5 particles, (a) uniform, (b) Gaussian, (c) Plummer. 46
3.7 Different particle distributions, compute time (s) versus relative ℓ_2 error, N=2E7 random particles, (a) uniform, (b) Gaussian, (c) Plummer, BLTC (red, dashed), BLDTT (blue, solid), connected curves represent constant MAC θ (0.5 $\times$; 0.7 $\circ$; 0.9 $*$), interpolation degree $n = 1, 2, 4, 6, 8, 10$ increases from right to left on each curve, simulations ran on one NVIDIA P100 GPU. 47
3.8 Different particle distributions, compute time (s) versus relative ℓ_2 N=2E7 random particles, (a) uniform, (b) Gaussian, (c) Plummer, BLDTT with only CC and PP interactions (red, dashed), BLDTT with PP, PC, CP, and CC interactions (blue, solid), connected curves represent constant MAC θ (0.5 $\times$; 0.7 $\circ$; 0.9 $*$), interpolation degree $n = 1, 2, 4, 6, 8, 10$ increases from right to left on each curve, simulations ran on one NVIDIA P100 GPU. 48
3.9 Different particle distributions, number of pointwise interactions (kernel evaluations $G(\mathbf{x}, \mathbf{y})$), N=2E7 random particles, (a) uniform, (b) Gaussian, (c) Plummer, MAC $\theta = 0.9$, interpolation degree $n = 1, 2, 4, 6, 8, 10$, each pair of bars shows BLDTT with CC and PP only (left) and BLDTT with PP, PC, CP, CC (right), direct sum calculation would use 4E14 PP interactions, simulations ran on one NVIDIA P100 GPU. 48
3.10 Non-cubic domains, N=4E5 random uniformly distributed particles, (a) thin slab of dimensions $1 \times 10 \times 10$, (b) square rod of dimensions $1 \times 1 \times 10$, (c) spherical surface of radius 1. 49
3.11 Non-cubic domains, N=2E7 random uniformly distributed particles, (a) thin slab of dimensions $1 \times 10 \times 10$, (b) square rod of dimensions $1 \times 1 \times 10$, (c) sphere surface of radius 1, compute time (s) versus relative ℓ_2 error, BLTC (red, dashed), BLDTT (blue, solid), connected curves represent constant MAC θ (0.5 $\times$; 0.7 $\circ$; 0.9 $*$), interpolation degree $n = 1, 2, 4, 6, 8, 10$ increases from right to left on each curve, simulations ran on one NVIDIA P100 GPU. 50

3.12 Unequal targets and sources, (a) M=2E7 targets, N=2E6 sources, (b) M=2E6 targets, N=2E7 sources, random uniformly distributed particles, compute time (s) versus relative ℓ_2 error, BLTC (red, dashed), CP-BLTC (green, dash-dotted), BLDTT (blue, solid), connected curves represent constant MAC θ (0.5 $\times$; 0.7 $\circ$; 0.9 $*$), interpolation degree $n = 1, 2, 4, 6, 8, 10$, simulations ran on one NVIDIA P100 GPU. 51
3.13 Different interaction kernels, N=2E7 random uniformly distributed particles in the cube $[-1, 1]^3$, (a) oscillatory, $\sin(\pi r)/r$, (b) Yukawa, $\exp(-0.5r)/r$, (c) regularized Coulomb, $1/(r^2 + 0.005^2)^{1/2}$, compute time (s) versus relative ℓ_2 error, BLTC (red, dashed), BLDTT (blue, solid), connected curves represent constant MAC θ (0.5 $\times$; 0.7 $\circ$; 0.9 $*$), interpolation degree $n = 1, 2, 4, 6, 8, 10$ increases from right to left on each curve, simulations ran on one NVIDIA P100 GPU. . . 52
3.14 Examples of domain decomposition, N=1.6E6 random uniformly distributed particles in the cube $[-1, 1]^3$, (a) 8 ranks with 2E5 particles per rank, (b) 16 ranks with 1E5 particles per rank, colors represent particles residing on different ranks, partitioning by Trilinos Zoltan. 53
3.15 MPI strong scaling, N=64E6 particles, (a) random uniform, (b) Gaussian, (c) Plummer distributions, MAC $\theta = 0.7$, interpolation degree $n = 8$ yielding 7-8 digit accuracy, compute time (s) versus number of GPUs, BLTC (red), BLDTT (blue), ideal scaling (dashed lines), parallel efficiency (boxed numbers). 53
3.16 Component breakdown of run time across 1 to 32 NVIDIA P100 GPUs, 64E6 random uniformly distributed particles in the cube $[-1, 1]^3$, MAC $\theta = 0.7$, interpolation degree $n = 8$, error $\approx$ 1E–8, (a) BLTC, (b) BLDTT, upward pass (blue), compute due to local sources and source clusters (orange), compute due to remote sources and source clusters (yellow), downward pass (purple), LET construction and communication (green), and other (light blue), which includes tree building and interaction list building, breakdown is based on timing results for most expensive MPI rank in each case. 54

4.1 Triangle aspect ratio versus number of elements N for each generated surface, (a) average aspect ratio, r_{avg}, (b) maximum aspect ratio, r_{max}, MSMS (black, $\circ$), NanoShaper (red, $\triangledown$). 62
4.2 Protein 1AIE, SES triangulation and electrostatic potential, (a) MSMS, density $d = 6$, $N = 31480$ triangles, (b) NanoShaper, scaling parameter $s = 2$, $N = 32208$ triangles. 63
4.3 Protein 1AIE, zoom of SES triangulation, (a) MSMS, density $d = 6$, $N = 31480$ triangles, green boxes enclose *stitches* formed by high aspect ratio triangles, white box encloses a *cusp* formed by neighboring triangles that meet at an acute angle, (b) NanoShaper, scaling parameter $s = 2$, $N = 32208$ triangles, yellow box encloses a possible irregular feature. 64
4.4 Surface area S_a versus N^{-1} for four representative proteins, where N is the number of triangles, MSMS (black, solid, $\circ$), NanoShaper (red, dashed, $\triangledown$). . . 65
4.5 Solvation energy E_{sol} versus N^{-1} for four representative proteins, where N is the number of triangles, MSMS (black, solid, $\circ$), NanoShaper (red, dashed, $\triangledown$). . . 66

4.6 NanoShaper versus MSMS results for entire set of 38 biomolecules using values extrapolated to the limit $N \to \infty$, (a) surface area S_a, (b) solvation energy ΔG_{solv}, black lines indicate perfect correspondence. 67
4.7 Computational efficiency, total run time of TABI-PB for computing solvation energy E_{sol} using MSMS (black, $\circ$) and NanoShaper (red, $\times$) versus number of triangles N, (a) run time (s), solid lines are least squares fits, (b) number of GMRES iterations (maximum 110). Simulations ran in serial on Intel Xeon CPU. 68
4.8 Construction of a node patch from faces of a surface mesh. (a) Five faces meet at a vertex. A patch shown in (b) is formed by taking one third of the area of each triangle surrounding the vertex. 69
4.9 $1/N$ versus solvation energy for four representative proteins, where N is number of computational elements (faces for collocation, vertices for node patch), collocation (black, solid, $\circ$), node patch (red, dashed, $\triangledown$). 70
4.10 Solvation energy relative error versus CPU time for four representative proteins, collocation in original C rewrite of TABI-PB (black, solid, $\circ$), collocation in code refactored TABI-PB (blue, dashed, $\times$), node patch (red, dashed, $\triangledown$). Note that the GMRES tolerance is 1E–4. Simulations ran in serial on Intel Xeon CPU. . . 71
4.11 Computational performance of TABI-PB using collocation (black, $\circ$) and node patch (red, $\triangledown$), over all runs from test set, (a) $1/N$ versus solvation energy error, (b) solvation energy error versus run time. Simulations ran in serial on Intel Xeon CPU. 72
4.12 Number of surface elements versus run time (s) for four representative proteins, Taylor treecode TABI-PB (red, $\triangledown$), BLDTT TABI-PB (blue, $\circ$), NanoShaper scaling parameters 1.0, 1.5, 2.0, 2.5, 3.0, 3.5, 4.0, DS run as BLDTT TABI-PB with $\theta = 0.0$, TC run with $\theta = 0.8$, $p = 3$, $N_0 = 500$, DTT run with $\theta = 0.8$, $n = 3$, $N_0 = 50$. Simulations ran in serial on Intel Xeon CPU. 75
4.13 Solvation energy relative error versus run time (s) for four representative proteins, Taylor treecode TABI-PB (red, $\triangledown$), BLDTT TABI-PB (blue, $\circ$), NanoShaper scaling parameters 1.0, 1.5, 2.0, 2.5, 3.0, 3.5, 4.0, DS run as BLDTT TABI-PB with $\theta = 0.0$, TC run with $\theta = 0.8$, $p = 3$, $N_0 = 500$, DTT run with $\theta = 0.8$, $n = 3$, $N_0 = 50$. Simulations ran in serial on Intel Xeon CPU. 77
4.14 Number of surface elements versus run time (s) for a single Intel Xeon CPU core (red, $\circ$) versus NVIDIA P100 GPU (blue, $\circ$), across NanoShaper scaling parameters 1.0, 1.5, 2.0, 2.5, 3.0, 3.5, 4.0, PDB ID 1A63, (a) all components of run time including surface meshing with NanoShaper, (b) surface meshing time excluded. Note that the NanoShaper surface meshing software is CPU only, and is the only part of the code that has not been GPU-accelerated. 78
4.15 Component breakdown of run time across NanoShaper scaling parameters 1.0, 1.5, 2.0, 2.5, 3.0, 3.5, 4.0, PDB ID 1A63, (a) single Intel Xeon CPU core, (b) NVIDIA P100 GPU, build NanoShaper surface mesh (blue), compute source terms (orange), upward pass (yellow), downward pass (purple), compute interactions (green), compute ΔG_{solv}(light blue), other (burgundy) which includes tree building, interaction list building, and all parts of GMRES other than the matrix-vector product. 78

4.16 (a) The Poisson–Boltzmann model of biomolecular solvation, where Ω_1 is the solute domain with dielectric constant ε_1, Ω_2 is the solvent domain with dielectric constant ε_2, and Γ is the molecular surface. (b) The binding model for two biomolecular monomers. If the electrostatic solvation energy for monomer A is $G^{\mathrm{A}}_{\mathrm{solv}}$, for monomer B is $G^{\mathrm{B}}_{\mathrm{solv}}$, and for the bound complex is $G^{\mathrm{AB}}_{\mathrm{solv}}$, then the binding solvation energy ΔG_{solv} is the difference between $G^{\mathrm{AB}}_{\mathrm{solv}}$ and $G^{\mathrm{B}}_{\mathrm{solv}} + G^{\mathrm{B}}_{\mathrm{solv}}$ 79
4.17 Thermodynamic loop illustrating the binding of two monomers A and B, only electrostatic interactions considered. Subscripts s, v denote solvent and vacuum, respectively. Electrostatic binding free energy $\Delta\Delta G_{\mathrm{bind}}$ is determined by the change in vacuum electrostatic energy ΔE_{coul} and electrostatic binding solvation energy $\Delta\Delta G_{\mathrm{solv}} = \Delta G^{\mathrm{AB}}_{\mathrm{solv}} - \Delta G^{\mathrm{A}}_{\mathrm{solv}} - \Delta G^{\mathrm{B}}_{\mathrm{solv}}$. 80
4.18 Features of the target biomolecules across all three datasets, (a) number of atoms, (b) total net charge. 81
4.19 (a) MIBPB-calculated binding energy versus TABI-PB-calculated binding energy, (b) relative deviation in TABI-PB-calculated binding energy from MIBPB-calculated binding energy for entire test set, for MSMS and NanoShaper $d_{1/2}$ extrapolations and NanoShaper d_5 results. 84
4.20 Average relative deviation of TABI-PB results from MIBPB results versus total CPU time (s) for computing entire test set, numbers next to data points are corresponding densities. Simulations ran in serial on Intel Xeon CPU. 85
4.21 (a) Relative error and (b) absolute error in kcal/mol of NanoShaper binding energy in comparison to NanoShaper $d_{1/2}$ extrapolation, (c) relative error and (d) absolute error in kcal/mol of NanoShaper complex solvation energy in comparison to NanoShaper $d_{1/2}$ extrapolation. 86

5.1 (a) Radial distribution functions (RDF) of water, depicting all three site-site interactions, O–O, O–H, and H–H. (b) Cross section of solute site-solvent site RDFs of a hydroxymethyl group surrounded by water, showing relative density of oxygen and hydrogen molecules in solution around the hydroxymethyl group. Figures courtesy of Tyler Luchko. 93
5.2 Stick representations of solutes used in this work. 98
5.3 Dependence of the relative numerical error of the solvation free energy (SFE) and partial molar volume (PMV) on the 3D-RISM residual tolerance. Relative errors are calculated against a reference calculation converged to a residual tolerance of 1E−13. 100
5.4 Relative speedup of treecode TCF LRA compared to direct summation versus relative error in $\mu_{\mathrm{ex,kh}}$ for tubulin, adhiron, CB7, and phenol, Taylor series order $p = 2k$, $k = 1, \ldots, 10$, increasing from right to left for each line. Simulations ran in serial on Metropolis cluster. 102
5.5 Relative speedup of treecode DCF LRA compared to direct summation versus relative error in $\mu_{\mathrm{ex,kh}}$ for tubulin, adhiron, CB7, and phenol, Taylor series order $p = 2k$, $k = 1, \ldots, 10$, increasing from right to left for each line. Simulations ran in serial on Metropolis cluster. 103

5.6 Relative speedup of treecode Coulomb potential energy compared to direct summation versus relative error in $\mu_{\text{ex,kh}}$ for tubulin, adhiron, CB7, and phenol, Taylor series order $p = 2k$, $k = 1, \ldots, 10$, increasing from right to left for each line. Simulations ran in serial on Metropolis cluster. 104
5.7 Relative error in $\mu_{\text{ex,kh}}$ of 3D-RISM calculations with treecode parameters $\theta = 0.3$ and $N_0 = 500$ versus Taylor series order, p, for tubulin, adhiron, CB7, and phenol. 106
5.8 Total runtime of 3D-RISM converged to a tolerance of 1E−6, with potential and asymptotics calculated using direct and treecode summation. Required runtime is shown for setting up the calculations (potential and asymptotics) and iterating to a converged solution. Treecode and cut-off parameters can be found in Table 5.2. Simulations ran in serial on Metropolis cluster. 108
5.9 Runtime for different components of the potential and asymptotics calculation for Fig. 5.8 using direct and treecode summation. Calculations were solved to a residual tolerance of 1E−6. Simulations ran in serial on Metropolis cluster. . . 109
5.10 Speedup over multiple cores of the total calculation time for direct and treecode summation 3D-RISM calculations converged to a tolerance of 1E−6 on Metropolis and Stampede2 clusters. Treecode and cut-off parameters can be found in Table 5.2. 110
5.11 Computation time over multiple cores of the total calculation time and various components for direct and treecode summation 3D-RISM calculations on a tubulin dimer converged to a tolerance of 1E−6 on Metropolis and Stampede2 clusters. Treecode and cut-off parameters can be found in Table 5.2. 112
5.12 1Z7Q TCF LRA calculation, original BaryTree CP-BLTC (**blue**), heavily specialized CP-BLTC for 3D-RISM (**green**), connected curves represent constant MAC θ (0.7 $\times$, solid; 0.9 $\circ$, dashed), interpolation degree $n = 1, 2, 4, 6, 8, 10$ increases from right to left on each curve (original CP-BLTC run only to $n = 8$), simulations ran on one NVIDIA P100 GPU. 115

6.1 Thermodynamic loop illustrating the binding of two monomers A and B. Subscripts s, v denote solvent and vacuum, respectively. Binding free energy $\Delta\Delta G_{\text{bind}}$ is determined by the change in vacuum molecular mechanics energy ΔE_{MM}, containing van der Waals, electrostatic, and bonding terms, an entropic contribution $T\Delta S$, and binding solvation energy $\Delta\Delta G_{\text{solv}} = \Delta G^{\text{AB}}_{\text{solv}} - \Delta G^{\text{A}}_{\text{solv}} - \Delta G^{\text{B}}_{\text{solv}}$, containing polar and non-polar terms. 121
B.1 1Z7Q TCF LRA calculation, original BaryTree CP-BLTC (blue), heavily specialized CP-BLTC for 3D-RISM (green), connected curves represent constant MAC θ (0.7 $\times$, solid; 0.9 $\circ$, dashed), interpolation degree $n = 1, 2, 4, 6, 8, 10$ increases from right to left on each curve (first three versions run only to $n = 8$), simulations ran on one NVIDIA P100 GPU. 137
B.2 Run time component breakdown of the five codes (labeled by color) shown in Fig. B.1 for MAC $\theta = 0.9$, across interpolation degrees $n = 1, 2, 4, 6, 8$, (a) absolute time breakdown, (b) proportional percentage breakdown. 139

LIST OF TABLES

TABLE

3.1 Comparison of BLTC and BLDTT, number of particles $N = 1E5, 1E6, 1E7, 1E8$, random uniformly distributed particles in $[-1, 1]^3$ interacting by the Coulomb kernel, MAC parameter $\theta = 0.7$, degree $n = 8$, compute time (s) from Fig. 3.5, ℓ_2 error, simulations ran on one NVIDIA P100 GPU. 45

3.2 Comparison of BLDTT running on 8 CPU cores and on one NVIDIA P100 GPU, number of particles $N = 1E5, 1E6, 1E7, 1E8$, random uniformly distributed particles in $[-1, 1]^3$ interacting by the Coulomb kernel, MAC parameter $\theta = 0.7$, degree $n = 8$, compute time (s), speedup, same errors as in Table 3.1. 46

4.1 PDB ID and number of atoms for test set of 38 biomolecules comprising proteins, peptides, and nucleic acid fragments. 60

4.2 Triangulation filter results showing percent of deleted triangles and zero-area triangles, values shown are averaged over all triangulations using MSMS and NanoShaper, zero-area triangles are a subset of deleted triangles. 61

4.3 Average run time (s) and average number of GMRES iterations per triangle for MSMS and NanoShaper meshes over entire set of 38 biomolecules. Simulations ran in serial on Intel Xeon CPU. 67

4.4 (a) Target and source charges $p_{i\ell}$ and $q_{j\ell}$ for computing the matrix-vector product in GMRES, (b) terms for computing the matrix-vector product in GMRES. For a given source particle $\mathbf{x}_j$ and target particle $\mathbf{x}_i$, the input vector values for $\mathbf{x}_j$ are $\phi_1(\mathbf{x}_j)$ and $\frac{\partial \phi_1(\mathbf{x}_j)}{\partial n}$. The contribution to the output value $\phi_1(\mathbf{x}_i)$ is $p_{i0}V_0$, and the contribution to $\frac{\partial \phi_2(\mathbf{x}_i)}{\partial n}$ is $p_{i1}V_1 + p_{i2}V_2 + p_{i3}V_3$. 73

4.5 Solvation energy (kcal/mol), solvation energy relative error, and run time (s) for direct sum boundary integral PB (DS), Taylor treecode TABI-PB (TC), and BLDTT TABI-PB (DTT), for PDB ID 1A63, NanoShaper scaling parameters 1.0, 1.5, 2.0, 2.5, 3.0, 3.5, 4.0, DS run as BLDTT TABI-PB with $\theta = 0.0$, TC run with $\theta = 0.8$, $p = 3$, $N_0 = 500$, DTT run with $\theta = 0.8$, $n = 3$, $N_0 = 50$. Simulations ran in serial on Intel Xeon CPU. 76

4.6 Corresponding densities for MSMS and NanoShaper. 82

4.7 Top section: average % deviation of TABI-PB calculated binding energy for all three test sets from MIBPB-calculated binding energy. Bottom section: total CPU time (in kiloseconds); TABI-PB comparison is given for NanoShaper (NS) in densities $d_1 - d_5$ and extrapolated in $d_{1/2}$ and $d_{2/3}$, and for MSMS in densities $d_1 - d_3$ and extrapolated in $d_{1/2}$ and $d_{2/3}$. Simulations ran in serial on Intel Xeon CPU. 83

5.1 Solutes used in this work. 97
5.2 Optimized 3D-RISM parameter settings. Treecode parameters MAC θ, order p. All LJ cutoffs were adjusted to fit inside the solvation box. 99
5.3 Guide to selecting treecode parameters for a given residual tolerance. Recommended parameters should be tested before production use. 105

LIST OF ALGORITHMS

ALGORITHM

2.1 The particle-cluster treecode with Taylor expansions. 11
2.2 The cluster-particle treecode with Taylor expansions. 14

3.1 High-level overview of the barycentric Lagrange dual tree traversal (BLDTT) fast summation method. 31
3.2 The dual tree traversal approach used in the BLDTT. 36

5.1 An example of the RISM procedure with removal of asymptotic components using Picard iteration. 94

A.1 BaryTree BLDTT implementation, with special attention paid to MPI calls, GPU data transfers, and GPU compute kernels. 126
A.2 Computing potential and proxy potential contributions due to the interaction lists $\mathcal{L}$. 129
A.3 The upward pass computes proxy charges of source clusters. 134
A.4 The downward pass increments the local potential with proxy potentials of target clusters. 135

LIST OF APPENDICES

APPENDIX

A Implementation Details of BaryTree **125**

B Implementation Details of GPU-Accelerated TCF LRA Treecodes **136**

C Advice for Scientific Computing Software Projects **140**

LIST OF ACRONYMS

MAC multipole acceptance criterion

FMM fast multipole method

DTT dual tree traversal

BLTC barycentric Lagrange treecode, specifically, the particle-cluster form

CP-BLTC cluster-particle barycentric Lagrange treecode

BLDTT barycentric Lagrange dual tree traversal fast summation method

PP particle-particle

PC particle-cluster

CP cluster-particle

CC cluster-cluster

PB Poisson–Boltzmann

SES solvent excluded surface

TABI-PB treecode accelerated boundary integral Poisson–Boltzmann equation solver

RISM reference interaction site model

1D-RISM one dimensional reference interaction site model

3D-RISM three dimensional reference interaction site model

MDIIS modified direct inversion of the iterative subspace

KH Kovalenko–Hirata closure

SFE solvation free energy

PMV partial molar volume

LRA long-range asymptotic function

DCF direct correlation function

TCF total correlation function

LIST OF SYMBOLS

M number of target particles

N number of source particles

$\mathbf{x}_i$ target particle in 3D

x_i target particle in 1D

$\mathbf{y}_j$ source particle in 3D

y_j source particle in 1D

q_j source charge of y_j

$G(x, y)$ interaction kernel

$\phi(x_i)$ potential at target x_i

C_t cluster of target particles

C_s cluster of source particles

$\mathbf{y}_C$ center of cluster C

r_C radius of cluster C

R distance between cluster and a particle, or distance between two clusters

p Taylor series expansion order parameter

θ multipole acceptance criterion, or MAC

M_0 maximum number of target particles per leaf

N_0 maximum number of source particles per leaf

$a_{\mathbf{k}}(\mathbf{x}_i, \mathbf{y}_c)$ $\mathbf{k}$th Taylor coefficient of the particle-cluster interaction between $\mathbf{x}_i$ and C

$M_{\mathbf{k}}(C)$ $\mathbf{k}$th Taylor moment of cluster C

n interpolation degree

$L_k(x)$ kth Lagrange polynomial evaluated at x

$\mathbf{t}_{\boldsymbol{\ell}}$ target proxy particle in 3D

t_ℓ target proxy particle in 1D

$\mathbf{s_k}$ source proxy particle in 3D

s_k source proxy particle in 1D

$\widehat{q}_k$ proxy charge of s_k

$\widehat{C}_t$ proxy particles in target cluster

$\widehat{C}_s$ proxy particles in source cluster

$\phi_{PP}(x_i, C_t, C_s)$ PP potential at target x_i in C_t due to sources in C_s

$\phi_{PC}(x_i, C_t, \widehat{C}_s)$ PC potential at target x_i in C_t due to proxy sources in $\widehat{C}_s$

$\phi_{CP}(t_\ell, \widehat{C}_t, C_s)$ CP proxy potential at proxy target t_ℓ in $\widehat{C}_t$ due to sources in C_s

$\phi_{CC}(t_\ell, \widehat{C}_t, \widehat{C}_s)$ CC proxy potential at proxy target t_ℓ in $\widehat{C}_t$ due to proxy sources in $\widehat{C}_s$

Ω_1 interior domain containing solute biomolecule

Ω_2 exterior domain containing ionic solvent

Γ interface between Ω_1 and Ω_2

N_c number of atoms in solute biomolecule

$\mathbf{y}_k$ position of kth solute atom

q_k partial charge of kth solute atom, in units of fundamental charge e_c

ε dielectric constant

ε_1 dielectric constant in solute domain

ε_2 dielectric constant in solvent domain

$\overline{\kappa}$ modified Debye-Hückel parameter in units of Å^{-2}

N_A Avogadro's number

k_B Boltzmann's constant

T temperature

I_s molar concentration of ionic solvent

ΔG_{solv} electrostatic solvation energy

$\phi_{\text{reac}}(\mathbf{y}_k)$ reaction field potential at atomic position $\mathbf{y}_k$

$\Delta\Delta G_{\text{bind}}$ electrostatic binding free energy

a RISM solute site

$\mathbf{R}_a$ position of solute site a

Q_a^U partial charge of solute site a

$c_\gamma(\mathbf{r})$ translationally invariant direct correlation function (DCF) on solvent site γ

$h_\gamma(\mathbf{r})$ translationally invariant total correlation function (TCF) on solvent site γ

$c_\gamma^{(\mathbf{lr})}(\mathbf{r})$ asymptotic long range direct correlation function on solvent site γ in real space

$h_j^{(\mathbf{lr})}(\mathbf{r})$ asymptotic long range total correlation function of ionic species j in real space

$\hat{c}_\gamma^{(\mathbf{lr})}(\mathbf{k})$ asymptotic long range direct correlation function on solvent site γ in reciprocal space

$\hat{h}_j^{(\mathbf{lr})}(\mathbf{k})$ asymptotic long range total correlation function of ionic species j in reciprocal space

η charge smearing parameter

κ_D contribution to the inverse Debye length of an ionic species

ABSTRACT

This work addresses the development of fast summation methods for long range particle interactions and their application to problems in biomolecular solvation, which describes the interaction of proteins or other biomolecules with their solvent environment. At the core of this work are treecodes, tree-based fast summation methods which, for N particles, reduce the cost of computing particle interactions from $O(N^2)$ to $O(N \log N)$. Background on fast summation methods and treecodes in particular, as well as several treecode improvements developed in the early stages of this work, are presented.

Building on treecodes, dual tree traversal (DTT) methods are another class of tree-based fast summation methods which reduce the cost of computing particle interactions for N particles to $O(N)$. The primary result of this work is the development of an $O(N)$ dual tree traversal fast summation method based on barycentric Lagrange polynomial interpolation (BLDTT). This method is implemented to run across multiple GPU compute nodes in the software package BaryTree. Across different problem sizes, particle distributions, geometries, and interaction kernels, the BLDTT shows consistently better performance than the previously developed barycentric Lagrange treecode (BLTC).

The first major biomolecular solvation application of fast summation methods presented is to the Poisson–Boltzmann implicit solvent model, and in particular, the treecode-accelerated boundary integral Poisson–Boltzmann solver (TABI-PB). The work on TABI-PB consists of three primary projects and an application. The first project investigates the impact of various biomolecular surface meshing codes on TABI-PB, and integrated the NanoShaper software into the package, resulting in significantly better performance. Second, a node patch method for discretizing the system of integral equations is introduced to replace the previous centroid collocation scheme, resulting in faster convergence of solvation energies. Third, a new version of TABI-PB with GPU acceleration based on the BLDTT is developed, resulting in even more scalability. An application investigating the binding of biomolecular complexes is undertaken using the previous Taylor treecode-based version of TABI-PB. In addition to these projects, work performed over the course of this thesis integrated TABI-PB into the popular Adaptive Poisson–Boltzmann Solver (APBS) developed at Pacific Northwest National Laboratory.

The second major application of fast summation methods is to the 3D reference interaction site model (3D-RISM), a statistical-mechanics based continuum solvation model. This work applies cluster-particle Taylor expansion treecodes to treat long-range asymptotic Coulomb-like potentials in 3D-RISM, and results in significant speedups and improved scalability to the 3D-RISM package implemented in AmberTools. Additionally, preliminary work on specialized GPU-accelerated treecodes based on BaryTree for 3D-RISM long-range asymptotic functions is presented.

CHAPTER 1

Introduction

This thesis addresses the development of fast summation methods for long range particle interactions and their application to problems in biomolecular solvation, which describes the interaction of proteins or other biomolecules with their solvent environment. At the core of this work are treecodes, tree-based fast summation methods which, for N particles, reduce the cost of computing particle interactions from $O(N^2)$ to $O(N \log N)$. Background on fast summation methods and treecodes in particular, as well as several treecode improvements developed in the early stages of this work, are presented in Chapter 2. Dual tree traversal (DTT) methods are a class of tree-based fast summation methods which reduce the cost of computing particle interactions for N particles to $O(N)$. The development of the GPU-accelerated barycentric Lagrange dual tree traversal (BLDTT) algorithm, which forms the primary result of this thesis, is detailed in Chapter 3. The first major biomolecular solvation application of fast summation methods presented is to the Poisson–Boltzmann implicit solvent model, and in particular, the treecode-accelerated boundary integral Poisson–Boltzmann solver (TABI-PB), detailed in Chapter 4. The second major application of fast summation methods is to the reference interaction site model (RISM), a statistical-mechanics based continuum solvation model, detailed in Chapter 5. We summarize the primary contents of each chapter below.

1.1 Overview of fast summation methods and early results

This chapter provides a brief survey of fast summation methods for particle interactions, focusing on tree-based methods and treecodes in particular. Additionally, several early improvements to treecode methods developed in the course of this work are described.

Long-range particle interactions are essential in many areas of computational physics, including calculation of electrostatic or gravitational potentials, as well as discrete convolution

sums in boundary element methods. Consider the potential due to a set of N particles,

$$\phi(\mathbf{x}_i) = \sum_{j=1}^{N} G(\mathbf{x}_i, \mathbf{x}_j) q_j, \quad i = 1 : N, \tag{1.1}$$

where $\mathbf{x}_i$ is a target particle, $\mathbf{x}_j$ is a source particle with strength q_j, and $G(\mathbf{x}, \mathbf{y})$ is an interaction kernel. The cost of evaluating the potentials $\phi(\mathbf{x}_i)$ by direct summation scales like $O(N^2)$, which is prohibitively expensive for large systems, but several methods are available to reduce the cost. Among these are tree-based methods in which the particles are partitioned into clusters with a hierarchical tree structure.

In particular, treecodes are a class of tree-based fast summation methods that employ a hierarchical tree structure to efficiently approximate the interaction of N particles. Originally developed for use in gravitational N-body simulations [1], they can be applied to a wide variety of problems involving the interaction between source and and target sites which may or may not be coincident. For the case of coincident particles described above, treecode methods reduce the computational cost to $O(N \log N)$. The essential idea behind these methods is the replacement of direct particle-particle interactions between some particle and a spatially well-separated group of particles with a single particle-cluster interaction. This requires the construction of a hierarchical oct-tree structure and a criterion for determining whether a particle and a cluster are well-separated.

There are two treecode implementations described in this chapter for the general case of M target particles and N source particles, in which case a direct computation scales like $O(MN)$. The first described method, the particle-cluster treecode, which builds a tree on the sources, is $O\left((M+N)\log N\right)$. The second method, the cluster-particle treecode, which builds a tree on the targets, is $O\left((M+N)\log M\right)$. While particle-cluster treecodes are generally more efficient when $M < N$, cluster-particle treecodes are a better choice when $M > N$. To approximate the interaction between a particle and a cluster, the methods described employ Taylor expansions. This chapter describes recursion relations for calculating Taylor expansion coefficients for certain kernels to arbitrary order developed by Krasny and coworkers [2, 3, 4].

This chapter additionally describes several computational improvements to treecodes, particularly in their cluster-particle form, developed in the early stages of this work. These improvements produced significant memory savings for the application of treecodes to the RISM problem, detailed in Chapter 5.

1.2 BLDTT: GPU-accelerated barycentric Lagrange dual tree traversal

Joint work with Nathan Vaughn developed the GPU-accelerated barycentric Lagrange treecode (BLTC), a treecode method based on polynomial interpolation. This work is described in Nathan's thesis [5] and in [6]. Building on the BLTC, this chapter, following closely the work presented in [7], details the development of the $O(N)$ barycentric Lagrange dual tree traversal fast summation method (BLDTT), and its MPI and OpenACC implementation for running on multiple GPU nodes in the software package BaryTree.

The scheme replaces well-separated particle-particle interactions by adaptively chosen particle-cluster, cluster-particle, and cluster-cluster approximations given by barycentric Lagrange interpolation at proxy particles on a Chebyshev grid in each cluster. The BLDTT is kernel-independent and the approximations can be efficiently mapped onto GPUs, where target particles provide an outer level of parallelism and source particles provide an inner level of parallelism. We present an OpenACC GPU implementation of the BLDTT with MPI remote memory access for distributed memory parallelization. The performance of the GPU-accelerated BLDTT is demonstrated for calculations with different problem sizes, particle distributions, geometric domains, and interaction kernels, as well as for unequal target and source particles, and compared with the earlier BLTC. In addition, MPI strong scaling results are presented for the BLTC and BLDTT using N=64E6 particles on up to 32 GPUs. Further implementation details are presented in Appendix A. The BaryTree package implementing the BLDTT is available on GitHub at `github.com/Treecodes/BaryTree`.

1.3 A Poisson–Boltzmann equation solver

Implicit solvent models play an important role in computational modeling of electrostatic interactions between biomolecules and their solvent environment [8, 9, 10]. Of particular importance is the Poisson–Boltzmann (PB) model [11, 12]. Consider an interior domain $\Omega_1 \subset \mathbb{R}^3$ containing the solute biomolecule, and an exterior domain $\Omega_2 = \mathbb{R}^3 \setminus \overline{\Omega}_1$ containing the ionic solvent. In a 1:1 electrolyte at low ionic concentrations, one can utilize the linearized PB equation for the electrostatic potential ϕ,

$$-\nabla \cdot (\varepsilon(\mathbf{x})\nabla\phi(\mathbf{x})) + \overline{\kappa}^2(\mathbf{x})\phi(\mathbf{x}) = \sum_{k=1}^{N_c} q_k \delta\left(\mathbf{x} - \mathbf{y}_k\right), \tag{1.2}$$

where ε is the dielectric constant, $\overline{\kappa}$ is the modified Debye-Hückel parameter in units of Å^{-2}, N_c is the number of atoms in the solute biomolecule, $\mathbf{y}_k$ is the position of the kth atom of the solute, and q_k is the associated partial charge in units of fundamental charge e_c.

A variety of numerical approaches have been applied to the Poisson–Boltzmann model, including finite-difference [13, 14, 15, 16, 17, 18, 19, 20], finite-element [12, 21, 22], and boundary integral [23, 24, 25, 26, 27] schemes. In particular, a treecode-accelerated boundary integral scheme for the linearized Poisson–Boltzmann equation (TABI-PB) was recently developed [26, 28]. Boundary integral schemes for the PB equation solve for the surface potential on a triangulated discretization of the interface. These schemes generally benefit from rigorous enforcement of the interface conditions and the boundary condition at infinity. However, these schemes face the expense of solving a dense linear system, and hence TABI-PB has traditionally used a treecode algorithm to reduce the computational cost from $O(N^2)$ to $O(N \log N)$, where N is the number of triangles representing the interface.

The work in this thesis regarding TABI-PB consists of three primary projects and an application. The first project investigates the impact of surface triangulation codes on the performance of TABI-PB. In the Poisson–Boltzmann model, the solute-solvent interface is often taken to be the molecular surface or solvent-excluded surface (SES), and the quality of the SES triangulation is critical in boundary element simulations of the PB model. This project compares the MSMS and NanoShaper surface triangulation codes for a set of 38 biomolecules.

The second project implements a node patch method for forming the underlying linear system. Previously, TABI-PB has used a constant element centroid collocation scheme for discretizing the boundary integral problem. In the node patch scheme [12], the boundary integral equations are discretized so that the problem is computed on the vertices of the surface mesh elements instead of at the centroids of the faces.

Note that the previous two projects utilized the Taylor treecodes detailed in Chapter 2; the third project describes the development of a new BLDTT-based GPU-accelerated TABI-PB solver, applying the work detailed in Chapter 3 to TABI-PB. We then describe the application of TABI-PB to the calculation of electrostatic free energies of binding between proteins and ligands, comparing with a popular finite-difference based Poisson–Boltzmann software.

In addition to these projects, work performed over the course of this thesis integrated TABI-PB into the popular Adaptive Poisson–Boltzmann Solver (APBS) developed at Pacific Northwest National Laboratory [29]. TABI-PB is available at `github.com/Treecodes/TABI-PB` and as an APBS submodule at `github.com/Electrostatics/APBS`.

1.4 Treecode acceleration of 3D reference interaction site model

Beyond the Poisson–Boltzmann model detailed in Chapter 4, another approach to implicit solvent models are integral equation theories, based on the Ornstein-Zernike equation [30].

The 3D-reference interaction site model of molecular solvation (3D-RISM) [31, 32] is one such integral equation, which has been shown to provide solvation thermodynamics in good agreement with experiment and explicit solvent calculations [33, 34, 35, 36].

However, 3D-RISM can be computationally expensive, especially for large molecules. 3D-RISM calculations consist of three sequential steps: initialization (calculating potential energy and long-range electrostatic interactions on a 3D grid), iteration to convergence, and integration of the solvent distribution to calculate thermodynamics. For small molecules, iteration time dominates the calculation, which scales with the number of grid points, N_{grid}, as $O\left(N_{\text{grid}} \log N_{\text{grid}}\right)$. Initialization time dominates for typical proteins, scaling with both the number of solute atoms, N_{atom}, and grid points as $O\left(N_{\text{atom}} N_{\text{grid}}\right)$. Integrating solvent thermodynamics is typically 1% or less of the total computation time. Depending on the precision of the calculation, initialization becomes the most expensive part of the calculation for solutes of 1000 atoms or more and is a major barrier to the practical application of 3D-RISM to large molecules.

Limited work has been done to address the computational cost of initialization for open boundary conditions. Because there is no periodic structure, the entire potential energy is calculated for a real-space grid. In addition, to capture contributions beyond the size of the solvent box, analytic long-range asymptotic (LRA) expressions of the solvent correlation functions must also be computed in real- and reciprocal-space. So far, little has been done to address the cost of computing these expressions.

The work detailed in this chapter addresses the use of fast-summation methods to accelerate computing potential energy and LRA functions, which take the form of Coulomb-like potentials. In the evaluation of LRA functions in 3D-RISM, the solute is represented by N source particles and the solvent grid by M target sites. Typically $M \gg N$ in the case of the 3D-RISM solvent grid, making cluster-particle treecodes described in Chapter 2 appropriate to consider [37].

This work consists of two primary projects. The primary project related to this topic investigates the development of cluster-particle Taylor treecodes for accelerating the calculation of the Coulomb potential and LRA functions in 3D-RISM. The work presented largely follows [38], and the resulting code is available in AmberTools 2019 and later [39]. The second project documents the early development of BaryTree-based GPU-accelerated treecodes applied to 3D-RISM, contributing to the goal of a future fully GPU-accelerated 3D-RISM implementation, with further details of its development given in Appendix B.

CHAPTER 2

Overview of Fast Summation Methods and Early Results

This chapter provides a brief survey of fast summation methods for particle interactions, focusing on tree-based methods and treecodes in particular. §2.1 provides background on particle interactions and tree-based fast summation methods. §2.2 describes treecode algorithms as well as recursion relations for computing Taylor coefficients to arbitrary order for approximating particle and cluster interactions. §2.3 describes several computational improvements to treecodes developed in the early stages of this work that in particular produced significant memory savings for their application to 3D-RISM, detailed in Chapter 5.

2.1 Background

Long-range particle interactions are essential in many areas of computational physics, including calculation of electrostatic or gravitational potentials, as well as discrete convolution sums in boundary element methods. In this context consider the potential due to a set of N particles,

$$\phi(\mathbf{x}_i) = \sum_{j=1}^{N} G(\mathbf{x}_i, \mathbf{x}_j) q_j, \quad i = 1 : N, \tag{2.1}$$

where $\mathbf{x}_i$ is a target particle, $\mathbf{x}_j$ is a source particle with strength q_j, and $G(\mathbf{x}, \mathbf{y})$ is an interaction kernel. In electrostatics applications, for example, q_j is a charge and $G(\mathbf{x}, \mathbf{y})$ is the Coulomb kernel,

$$G(\mathbf{x}, \mathbf{y}) = \frac{1}{|\mathbf{x} - \mathbf{y}|}, \tag{2.2}$$

or the screened Coulomb kernel,

$$G(\mathbf{x}, \mathbf{y}) = \frac{\exp\left(-\kappa\, |\mathbf{x} - \mathbf{y}|\right)}{|\mathbf{x} - \mathbf{y}|}, \tag{2.3}$$

where κ is a screening parameter.

The cost of evaluating the potentials $\phi(\mathbf{x}_i)$ by direct summation scales like $O(N^2)$, which is prohibitively expensive for large systems, but several methods are available to reduce the cost.

Among these are mesh-based methods in which the particles are projected onto a regular mesh, such as particle-particle/particle-mesh (P3M) [40] and particle-mesh Ewald (PME) [41], and tree-based methods in which the particles are partitioned into clusters with a hierarchical tree structure, such as Appel's dual tree traversal (DTT) method [42], the Barnes-Hut treecode (TC) [1], and the Greengard-Rokhlin fast multipole method (FMM) [43, 44]. Other related methods for fast summation of particle interactions include panel clustering [45], hierarchical matrices [46], and multilevel summation [47].

Tree-based fast summation methods such as the DTT, TC, and FMM can be described as having two phases. In the precompute phase, the particles are partitioned into a hierarchical tree of clusters, and assuming the particle distribution is homogeneous, this phase generally scales like $O(N \log N)$. In the compute phase, well-separated particle-particle interactions are approximated using for example monopole approximations [42, 1] or higher order multipole expansions [43, 48, 49]. Some differences in the compute phase of these methods are noted as follows. The TC traverses the tree for each target particle to identify well-separated particle-cluster pairs, while the DTT traverses two copies of the tree simultaneously to identify well-separated cluster-cluster pairs; both methods use a multipole acceptance criterion (MAC) for this purpose. The FMM passes information from one level to the next, with a uniform definition of the interaction list at each level; an upward pass computes cluster moments, and a downward pass computes potentials using multipole-to-local and local-to-local translations. The compute phase of the TC scales like $O(N \log N)$, while the compute phase of the FMM [43, 44] and DTT [50, 49] scale like $O(N)$.

2.2 Treecodes

Treecodes are a particular class of tree-based fast summation methods that employ a hierarchical tree structure to efficiently approximate the interaction of N particles. Originally developed for use in gravitational N-body simulations [1], in which the force of N particle bodies on each other must be computed at each time step of a dynamics simulation, they can be applied to a wide variety of problems involving the interaction between N source particles on M target sites which may or may not be coincident with the sources. For the case of coincident sources and targets, direct sum scales like $O(N^2)$, while treecode methods are $O(N \log N)$. The essential idea behind these methods is the replacement of direct particle-particle interactions between some particle and a spatially well-separated group of particles with a single particle-cluster interaction. This requires the construction of a hierarchical oct-tree structure and a criterion for determining if a particle and a cluster are well-separated.

The simplest interaction between a particle and cluster is a monopole approximation in which, for a well-separated interaction, the cluster is replaced with a single particle at the center whose charge (or mass, or other interaction parameter) is equal to the sum of all particles contained within the cluster. This is similar to the strategy of the original gravitational Barnes–Hut treecode, in which the monopole was placed at the center of mass. To achieve higher accuracy, the interaction between a particle and a cluster can be approximated with a Taylor expansion (in which case, the Barnes–Hut implementation can be viewed as a zero-order expansion). However, hard coded Taylor expansions to high order can be costly and result in an inefficient implementation of the treecode. Krasny and coworkers [2, 3, 4] instead implemented recursion relations for calculating Taylor expansion coefficients to arbitrary order.

We note that there are two primary applications of the treecode in the work described in this document. The first is for the calculation of a boundary integral form of the Poisson-Boltzmann equation, described in Chapter 4. In this case, the targets and sources are coincident particles, representing elements of a discretized biomolecular surface with interactions described by integral kernels. The second is for the calculation of long range functions on a set of regular target grid points from a set of charged source particles in the Reference Interaction Site Model, described in Chapter 5. In this case, the targets and sources are clearly not generally coincident particles.

After describing the tree construction process, we describe here two implementations of treecodes for the general case of M target particles and N source particles. A direct computation would be $O(MN)$ in this case. The first described method, the particle-cluster treecode, which builds a tree on the sources, is $O\left((M+N)\log N\right)$. The second method, the cluster-particle treecode, which builds a tree on the targets, is $O\left((M+N)\log M\right)$.

2.2.1 Tree construction

Consider a set of either M target or N source particles. The tree is constructed as follows: The smallest box with sides parallel to the Cartesian coordinate axes containing all particles is formed. This top level domain is denoted the root cluster. This root cluster is divided uniformly through its center into eight child clusters with three cuts parallel to the Cartesian axes. Each child cluster is shrunk to the smallest box containing all particles within the cluster. The cluster is then further divided into eight more clusters until the resulting clusters contain less than some user specified value (referred to as M_0 for targets, and N_0 for sources), at which point the cluster is divided no further. The clusters at the lowest level of the tree (i.e., those with no children), are referred to as leaves.

2.2.2 Particle-cluster treecodes

We describe the particle-cluster treecode after the work of Boateng and Krasny in [37]. In the particle-cluster treecode, we apply the tree construction procedure described above to the N source particles. We depict a particle-cluster interaction in Fig. 2.1 between a target particle $\mathbf{x}_i$ and a source cluster C with center $\mathbf{y}_c$ and radius r containing particles $\mathbf{y}_j$ with charge q_j. The particle-cluster distance is $R = |\mathbf{x}_i - \mathbf{y}_c|$. The far-field Taylor approximation used by the treecode is valid when the target is well-separated from the cluster.

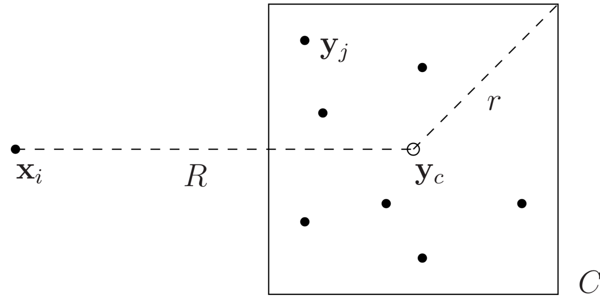


Figure 2.1: A particle-cluster interaction between a target particle $\mathbf{x}_i$ and a source cluster C with center $\mathbf{y}_c$ and radius r containing particles $\mathbf{y}_j$ with charge q_j. The particle-cluster distance is $R = |\mathbf{x}_i - \mathbf{y}_c|$.

We can write the potential expression given in Eq. 2.1 as

$$\phi(\mathbf{x}_i) = \sum_{C} \sum_{\mathbf{y}_j \in C} q_j G(\mathbf{x}_i, \mathbf{y}_j) \tag{2.4}$$

where C represents the source clusters with which target $\mathbf{x}_i$ interacts.

To compute a particle-cluster interaction, we Taylor expand $G(\mathbf{x}_i, \mathbf{y}_j)$ about the cluster center $\mathbf{y}_c$, giving the expression

$$\sum_{\mathbf{y}_j \in C} q_j G\left(\mathbf{x}_i, \mathbf{y}_j\right) \approx \sum_{\mathbf{y}_j \in C} q_j \sum_{\|\mathbf{k}\|=0}^{p} \frac{1}{\mathbf{k}!} \partial_{\mathbf{y}}^{\mathbf{k}} G\left(\mathbf{x}_i, \mathbf{y}_c\right) \left(\mathbf{y}_j - \mathbf{y}_c\right)^{\mathbf{k}} \tag{2.5}$$

$$= \sum_{\|\mathbf{k}\|=0}^{p} a_{\mathbf{k}}\left(\mathbf{x}_i, \mathbf{y}_c\right) M_{\mathbf{k}}\left(C\right), \tag{2.6}$$

where the Taylor coefficients are given by

$$a_{\mathbf{k}}\left(\mathbf{x}_i, \mathbf{y}_c\right) = \frac{1}{\mathbf{k}!} \partial_{\mathbf{y}}^{\mathbf{k}} G\left(\mathbf{x}_i, \mathbf{y}_c\right), \tag{2.7}$$

and the cluster moments are

$$M_{\mathbf{k}}(C) = \sum_{\mathbf{y}_j \in C} q_j (\mathbf{y}_j - \mathbf{y}_c)^{\mathbf{k}}. \tag{2.8}$$

Note that this is a Taylor series in three dimensions, where $\|\mathbf{k}\| = k_1+k_2+k_3$, $\mathbf{k}! = k_1!k_2!k_3!$, $\partial_{\mathbf{y}}^{\mathbf{k}} = \partial_{y_1}^{k_1}\partial_{y_2}^{k_2}\partial_{y_3}^{k_3}$, $(\mathbf{x}_i - \mathbf{x}_c)^{\mathbf{k}} = (x_{i1} - x_{c1})^{k_1}(x_{i2} - x_{c2})^{k_2}(x_{i3} - x_{c3})^{k_3}$, and $1, 2, 3$ denote the three respective Cartesian directions.

The particle-cluster treecode procedure is outlined in Algorithm 2.1. The treecode requires three parameters: a multipole acceptance criterion (MAC) θ, a Taylor series expansion order parameter p, and a maximum source leaf size M_0. The parameter M_0 determines the maximum number of sources allowable in a leaf cluster, and is used during the tree construction procedure.

The procedure in Algorithm 2.1 iterates over all target sites $\mathbf{x}_i$, computing the potential ϕ at each site. For each target site, the recursive procedure **compute-pc** computes the interaction between the target and a source cluster. The MAC θ determines if a particle-cluster interaction is evaluated, or if further children in the tree of source clusters are traversed. If the radius r of the cluster of sources and the distance R between the cluster center and a target particle R is such that $r/R \leq \theta$, then we evaluate the interaction. Otherwise, we traverse the children clusters of the source cluster.

The Taylor series expansion order parameter p specifies the order of the Taylor expansion for evaluating the cluster-particle interaction. A recurrence relation is used to calculate the Taylor coefficients, described below. If a source leaf-target particle interaction fails the MAC, then the interactions are evaluated directly.

Note that the source tree has $O(\log N)$ levels. At every tree level, each source particle contributes to the moment of one cluster. Thus, there are $O(N \log N)$ operations to compute the moments. When computing the potential, the tree is descended M times, once for each target particle, leading to an operation count of $O(M \log N)$. The particle-cluster treecode is thus $O((M + \alpha N) \log N)$, where α is some constant used here to emphasize the different leading coefficients in the asymptotic operation counts of moment computation and potential computation.

2.2.3 Cluster-particle treecodes

The cluster-particle treecode was initially introduced in the form given here by [37]. In the cluster-particle treecode, we apply the tree construction procedure described above to the M target particles. We depict a cluster-particle interaction in Fig. 2.2 between a target

Algorithm 2.1 The particle-cluster treecode with Taylor expansions.

```
1: procedure PARTICLE-CLUSTER
2:     Input: targets x_i, sources y_j, q_j, order p, MAC θ, max sources N_0
3:     construct tree of source clusters
4:     for i = 1, M do
5:         compute-pc(root, x_i)
6:     end for
7:     return potentials φ(x_i)
8: end procedure
1: procedure COMPUTE-PC(C, x)
2:     if MAC is satisfied then
3:         compute and store moments M_k(C) by Eq. 2.8, if not already available
4:         update Taylor coefficients a_k(x_c) by Eq. 2.7
5:         compute interaction by Taylor series approximation Eq. 2.5
6:     else if C is a leaf then
7:         compute interaction by direct summation
8:     else
9:         for each child of C do
10:            compute-pc(child, x)
11:        end for
12:    end if
13: end procedure
```

cluster C with center $\mathbf{x}_c$ and radius r containing particles $\mathbf{x}_i$ and a source particle $\mathbf{y}_j$ with charge q_j. The cluster-particle distance is $R = |\mathbf{y}_j - \mathbf{x}_c|$. The Taylor expansion used for a cluster-particle interaction can be considered a near-field approximation for all target particles in a cluster well-separated from the source.

Consider a tree of target clusters with L levels, where level L is the root cluster and level 1 are the leaves. A target site $\mathbf{x}_i$ will then belong to a nested sequence of clusters $\mathbf{x}_i \in C_1 \subseteq ... \subseteq C_L$, where cluster C_l is at level l. For each cluster C_l, denote its geometric center by $\mathbf{x}_c^l$. Let I_l denote the list of all source particles $\mathbf{y}_j$ that are well-separated from cluster C_l but not from cluster $C_1, ..., C_{l-1}$, and let D denote the list of all source particles $\mathbf{y}_j$ that are not well-separated from any cluster containing $\mathbf{x}_i$. Then we may partition Eq. 2.1 into not well-separated and well-separated sources using these interaction lists by

$$\phi(\mathbf{x}_i) = \sum_{\mathbf{y}_j \in D} q_j G(\mathbf{x}_i, \mathbf{y}_j) + \sum_{l=1}^{L} \sum_{\mathbf{y}_j \in I_l} q_j G(\mathbf{x}_i, \mathbf{y}_j). \tag{2.9}$$

The first term is calculated by direct summation, but the second term is handled by a Taylor expansion in three dimensions for each interaction list. Expanding the second term

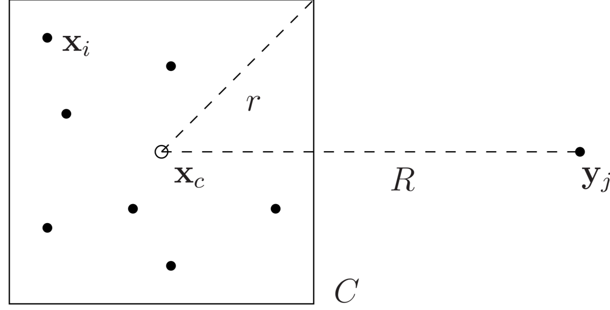


Figure 2.2: A cluster-particle interaction between a target cluster C with center $\mathbf{x}_c$ and radius r containing particles $\mathbf{x}_i$ and a source particle $\mathbf{y}_j$ with charge q_j. The cluster-particle distance is $R = |\mathbf{y}_j - \mathbf{x}_c|$.

$G(\mathbf{x}_i, \mathbf{y}_j)$ about $\mathbf{x}_c^l$, the center of cluster l, gives

$$\begin{aligned}\sum_{\mathbf{y}_j \in I_l} q_j G\left(\mathbf{x}_i, \mathbf{y}_j\right) &\approx \sum_{\mathbf{y}_j \in I_l} q_j \sum_{\|\mathbf{k}\|=0}^{p} \frac{1}{\mathbf{k}!} \partial_{\mathbf{x}}^{\mathbf{k}} G\left(\mathbf{x}_c^l, \mathbf{y}_j\right) \left(\mathbf{x}_i - \mathbf{x}_c^l\right)^{\mathbf{k}} \\ &= \sum_{\|\mathbf{k}\|=0}^{p} m_{\mathbf{k}}\left(\mathbf{x}_c^l\right) \left(\mathbf{x}_i - \mathbf{x}_c^l\right)^{\mathbf{k}},\end{aligned} \tag{2.10}$$

where the coefficients $m_{\mathbf{k}}$ are

$$m_{\mathbf{k}}\left(\mathbf{x}_c^l\right) = \sum_{\mathbf{y}_j \in I_l} q_j (-1)^{\|\mathbf{k}\|} a_{\mathbf{k}}\left(\mathbf{x}_c^l, \mathbf{y}_j\right), \tag{2.11}$$

and the Taylor coefficients $a_{\mathbf{k}}$ are

$$a_{\mathbf{k}}\left(\mathbf{x}_i, \mathbf{y}_j\right) = \frac{1}{\mathbf{k}!} \partial_{\mathbf{y}}^{\mathbf{k}} G\left(\mathbf{x}_i, \mathbf{y}_j\right). \tag{2.12}$$

The cluster-particle treecode procedure is described in Algorithm 2.2. This procedure also requires three parameters: a multipole acceptance criterion (MAC) θ, a Taylor series expansion order parameter p, and a maximum target leaf size N_0. The parameter N_0 determines the maximum number of targets allowable in a leaf cluster, and is used during tree construction.

The cluster-particle procedure occurs in two stages. In the first stage, we iterate over all sources, and the recursive routine **compute-cp1** updates the power series coefficients $m_{\mathbf{k}}(\mathbf{x}_c)$ for each target cluster. For any source-target leaf interactions which fail the MAC, we directly compute the contribution of the source to each target in the leaf. In the second stage, the **compute-cp2** routine evaluates for each target cluster the power series using the coefficients computed in the first stage for all targets contained within the cluster.

The target tree has $O(\log M)$ levels. In the first stage, the tree is traversed for each of the N source particles, leading to an $O(N \log M)$ operation count. In the second stage, we again traverse the tree, but for each of the M target particles, leading to an $O(M \log M)$ operation count. The cluster-particle treecode is thus $O\left((N + \beta M) \log M\right)$, where β is some constant again used to emphasize the different leading coefficients in the first and second stages of the computation.

2.2.4 Recurrence relations for Taylor coefficients

For certain interactions, the Taylor coefficients can be calculated efficiently to arbitrary order by using a recurrence relation. Previous work [2, 4] established recurrence relations for Coulomb and screened Coulomb interactions.

The Taylor coefficients $a_{\mathbf{k}}(\mathbf{x}, \mathbf{y})$ of the Coulomb interaction given in Eq. 2.2 can be calculated by the recurrence relation

$$a_{\mathbf{k}} = \frac{1}{|\mathbf{x} - \mathbf{y}|^2} \left[\left(2 - \frac{1}{\|\mathbf{k}\|}\right) \sum_{i=1}^{3} (x_i - y_i) a_{\mathbf{k}-\mathbf{e}_i} - \left(1 - \frac{1}{\|\mathbf{k}\|}\right) \sum_{i=1}^{3} a_{\mathbf{k}-2\mathbf{e}_i} \right], \tag{2.13}$$

where $\mathbf{e}_i$ is the ith Cartesian basis vector, and x_i represents the ith Cartesian component of $\mathbf{x}$. After explicitly computing the coefficients for $\|\mathbf{k}\| = 0, 1$, the rest may be computed using Eq. 2.13. Furthermore, if any index of $\|\mathbf{k}\|$ is negative, $a_{\mathbf{k}} = 0$.

The screened Coulomb interaction Taylor coefficients $a_{\mathbf{k}}(\mathbf{x}, \mathbf{y})$ are given by the recurrence relation

$$\begin{aligned} a_{\mathbf{k}} = \frac{1}{|\mathbf{x} - \mathbf{y}|^2} \Bigg[&\left(2 - \frac{1}{\|\mathbf{k}\|}\right) \sum_{i=1}^{3} (x_i - y_i) a_{\mathbf{k}-\mathbf{e}_i} - \left(1 - \frac{1}{\|\mathbf{k}\|}\right) \sum_{i=1}^{3} a_{\mathbf{k}-2\mathbf{e}_i} \\ &+ \kappa \left(\sum_{i=1}^{3} (x_i - y_i) b_{\mathbf{k}-\mathbf{e}_i} - \sum_{i=1}^{3} b_{\mathbf{k}-2\mathbf{e}_i} \right) \Bigg], \end{aligned} \tag{2.14}$$

where the $b_{\mathbf{k}}(\mathbf{x}, \mathbf{y})$ are coefficients to an auxiliary function $G(\mathbf{x}, \mathbf{y}) = \exp\left(-\kappa\, |\mathbf{x} - \mathbf{y}|\right)$ whose recurrence is given by

$$b_{\mathbf{k}} = \frac{\kappa}{\|\mathbf{k}\|} \left(\sum_{i=1}^{3} (x_i - y_i) a_{\mathbf{k}-\mathbf{e}_i} - \sum_{i=1}^{3} a_{\mathbf{k}-2\mathbf{e}_i} \right). \tag{2.15}$$

Algorithm 2.2 The cluster-particle treecode with Taylor expansions.

```
1: procedure CLUSTER-PARTICLE
2:     Input: targets x_i, sources y_j, q_j, order p, MAC θ, max targets N_0
3:     construct tree of target clusters
4:     for j = 1, N do
5:         compute-cp1(root, y_j)
6:     end for
7:     compute-cp2(root)
8:     return potentials φ(x_i)
9: end procedure
```

```
1: procedure COMPUTE-CP1(C, y)
2:     if MAC is satisfied then
3:         update power series coefficients m_k(x_c) by Eq. 2.11
4:     else if C is a leaf then
5:         compute first term in Eq. 2.9 by direct summation
6:     else
7:         for each child of C do
8:             compute-cp1(child, y)
9:         end for
10:    end if
11: end procedure
```

```
1: procedure COMPUTE-CP2(C)
2:     if C interacted with a source by Taylor approximation then
3:         for each target site x_i in C do
4:             compute second term in Eq. 2.9 using power series Eq. 2.10
5:         end for
6:     end if
7:     for each child of C do
8:         compute-cp2(child)
9:     end for
10: end procedure
```

2.3 Early projects: Computational improvements to treecodes

In the course of applying and optimizing treecode methods to the problems presented in this document, we have introduced a few minor computational improvements to the algorithms over their original implementation in [37]. These improvements have been particularly important to the treecodes used for the RISM problem, so we present them in the language of cluster-particle treecodes.

2.3.1 Tree virtualization and boundary grids

The original version of the cluster-particle treecode as implemented in [37] requires explicit storage of all target sites. When targets are located on a grid, this can result in using unnecessary memory to explicitly store the targets and unnecessary computation time in generating files of and reading in explicit target positions.

We instead propose a virtual treecode for grid targets, in which the tree is built using only the dimensions and the bounds of the grid, and for direct target-source interactions, the grid point positions are generated on the fly as needed. The grid is uniquely determined by nine numbers: the number of grid points, or dimension, in the x, y, and z directions, and the upper and lower limits of the grid in the x, y, and z directions. The implicit storage of targets also presents an advantage for distributed parallelization around targets for the cluster-particle treecode, since there is no need to direct the reading-in or communication of explicit target positions between multiple processors.

We can further extend the idea of virtual target grids to treecode computations on boundary grids. In this case, instead of constructing a 3D tree from the target points, we construct six 2D trees along each face of the computational domain, each of which is constructed from virtual targets.

2.3.2 Flattening coefficient arrays

In our initial implementation, for each tree leaf, the Taylor coefficients for the treecode were stored in a 3D array of dimension $(p+1) \times (p+1) \times (p+1)$, where p is the order of the treecode. However, the expansions are implemented in such a way that, for a given order p, only the Taylor coefficients such that $p_1 + p_2 + p_3 \leq p$, where p_i represents the expansion order in the ith direction, were used. Previous testing had demonstrated better computational performance at a given level of accuracy for this "pyramidal" order condition over the cubic condition $p_1, p_2, p_3 \leq p$.

To improve the memory usage of our treecode, we instead introduce a flat array of size $(p+1)(p+2)(p+3)/6$. Note that this expression is the sum of the first $p+1$ triangular

numbers, which is how many unique combinations of nonnegative integers p_1, p_2, p_3 satisfy the pyramidal order condition.

2.3.3 Results

We test the computational performance of our new methods on two test cases, one a full volume grid and one a boundary grid, and then consider their parallelization. All computations were performed in serial on the University of Michigan Flux cluster, with 2.5-2.8GHz Intel Xeon CPUs and 16GB of available memory. Timing results were averaged over multiple runs.

The relative ℓ_2 and ℓ_∞ errors, denoted by L_2 and L_∞, are defined by

$$L_2 = \left(\sum_{i=1}^{M} (\phi_i^{ds} - \phi_i^{tc})^2 \Big/ \sum_{i=1}^{M} (\phi_i^{ds})^2 \right)^{1/2}, \qquad L_\infty = \max_i \left|\phi_i^{ds} - \phi_i^{tc}\right| \Big/ \max_i \left|\phi_i^{ds}\right|, \tag{2.16}$$

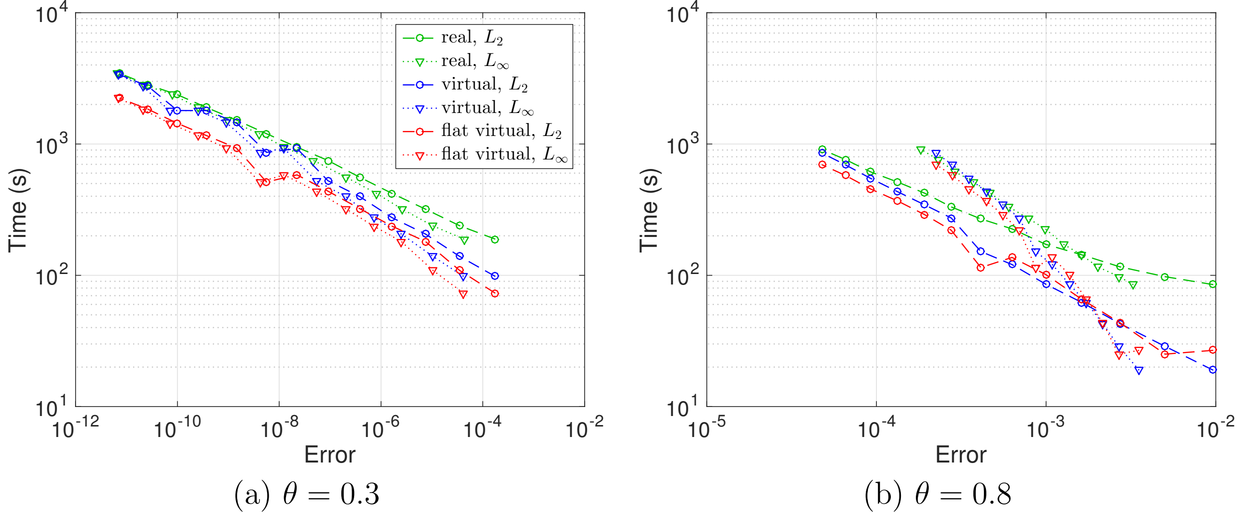
where ϕ_i^{ds} and ϕ_i^{tc} are the target potentials computed by direct summation and treecode approximation, respectively, at target point i.

2.3.3.1 Volume grid test case

We consider 1E5 sources distributed uniformly and randomly, and $400 \times 400 \times 400 = 64\text{E}6$ targets on an evenly spaced grid, with both sources and targets in a $1 \times 1 \times 1$ unit cube centered at the origin. The Coulomb potential is used in this test case.

In Fig. 2.3, we display the relative ℓ_2 and ℓ_∞ errors versus CPU time for three versions of the treecode: the original explicit treecode, denoted 'real', the virtual treecode, denoted 'virtual', and the virtual treecode with flattened coefficient arrays, denoted 'flat virtual.' Plotted results were produced by varying the treecode expansion order parameter from 1 to 15. Additionally, we display the results for both a low and a high value of the multipole acceptance criterion, $\theta = 0.3, 0.8$.

Note that, for low order values (corresponding to higher error), the virtual treecode has a CPU time advantage over the real treecode, primarily because read-in of targets and sources and tree setup dominate computational cost at this level. At higher order values, corresponding to lower error, this advantage disappears for the virtual treecode as actual potential computation begins to dominate the computational cost. However, in the $\theta = 0.3$ case, the virtual tree with flattened coefficient arrays maintains a roughly 30% CPU time advantage over the original treecode, because accessing a flat array in order (contiguous memory accesses) is significantly less costly than accessing a dynamically allocated 3D array out of order (non-contiguous memory accesses).



(a) $\theta = 0.3$ (b) $\theta = 0.8$

Figure 2.3: Volume grid test case, 1E5 sources with 64E6 targets on a regular grid, relative ℓ_2 (solid, $\circ$) and ℓ_∞ (dotted, $\triangledown$) errors from exact solution versus CPU time for original explicit treecode (green), virtual treecode (blue), and virtual treecode with flattened coefficient arrays (red), for various treecode expansion orders from 1 to 15, increasing from right to left, and multipole acceptance criterion of (a) $\theta = 0.3$ and (b) $\theta = 0.8$. Corresponding direct calculation time was 9.43E4 s. Simulations ran in serial on Intel Xeon CPU.

In Fig. 2.4, we display the relative ℓ_2 and ℓ_∞ errors versus total memory usage for the three treecode versions shown in Fig. 2.3. For low order values, the storage of sources and targets dominate memory use, and thus the virtual tree presents a significant memory advantage. As the order of the treecode increases, the memory allocated to store the Taylor coefficients dominates total memory usage, lessening the memory advantage of the virtual tree. The virtual tree with flattened coefficient arrays, however, displays a significant memory use advantage at higher orders.

2.3.3.2 Boundary grid test case

We consider 1E5 sources, distributed uniformly and randomly in a $1 \times 1 \times 1$ cube centered at the origin, and targets located on the six faces of a $1000 \times 1000 \times 1000$ point grid in the same domain as the sources. The Coulomb potential is used in this test case.

In Fig. 2.5, we display the relative ℓ_2 and ℓ_∞ errors versus CPU time for three versions of the treecode: the original explicit 3D treecode, denoted 'real', the 2D virtual treecode, denoted 'virtual', and the 2D virtual treecode with flattened coefficient arrays, denoted 'flat virtual.' Plotted results were produced by varying the treecode expansion order parameter from 1 to 15. Again, we display the results for both a low and a high value of the multipole acceptance criterion, $\theta = 0.3, 0.8$.

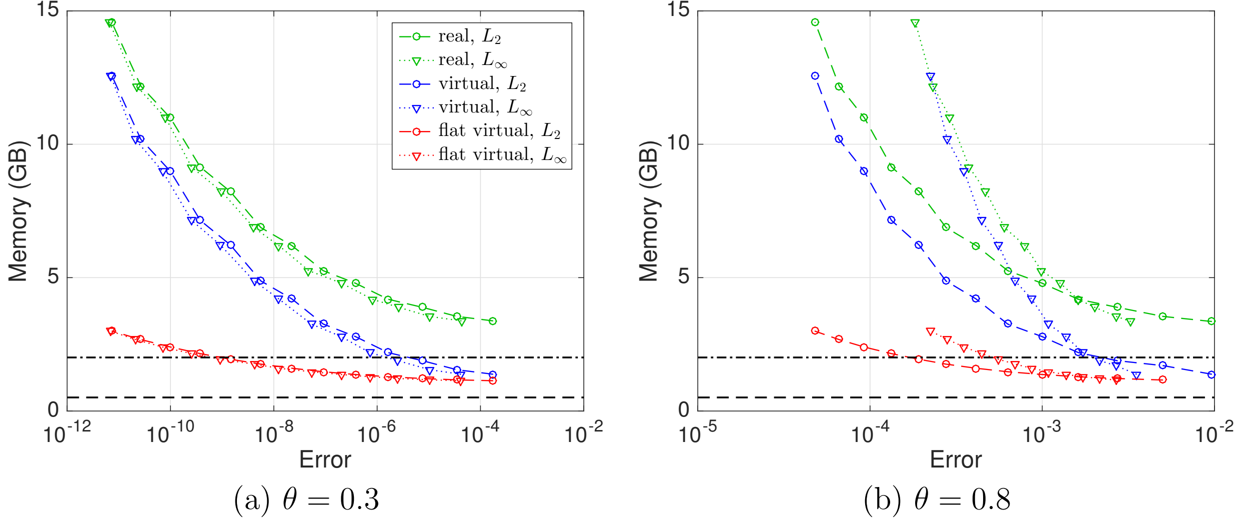


Figure 2.4: Volume grid test case, 1E5 sources with 64E6 targets on a regular grid, relative ℓ_2 (solid, $\circ$) and ℓ_∞ (dotted, $\triangledown$) errors versus total memory usage for original explicit treecode (green), virtual treecode (blue), and virtual treecode with flattened coefficient arrays (red), for treecode expansion order from 1 to 15, increasing from right to left, and multipole acceptance criterion of (a) $\theta = 0.3$ and (b) $\theta = 0.8$. Corresponding direct calculation memory usage was 2.01 GB with real targets (black, dash-dotted) and 0.51 GB with virtual targets (black, dashed). Simulations ran in serial on Intel Xeon CPU.

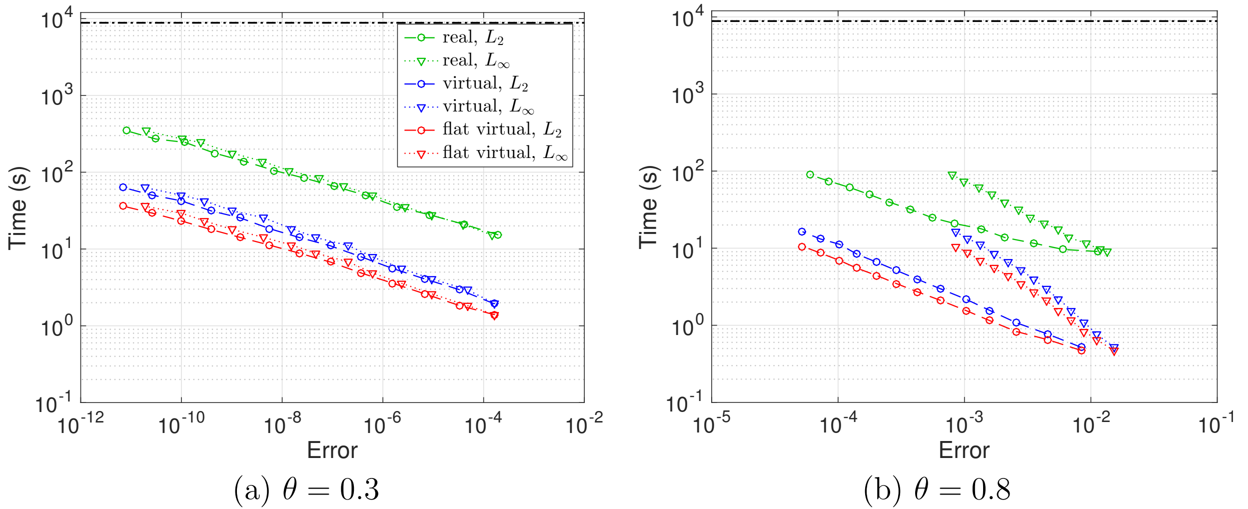


Figure 2.5: Boundary grid test case, 1E5 sources with targets on the six faces of a $1000 \times 1000 \times 1000$ point grid, L_2 (solid, $\circ$) and L_∞ (dotted, $\triangledown$) errors from exact solution versus CPU time for original explicit treecode (green), virtual 2D treecode (blue), and virtual 2D treecode with flattened coefficient arrays (red), for treecode expansion order from 1 to 15, increasing from right to left, and multipole acceptance criterion of (a) $\theta = 0.3$ and (b) $\theta = 0.8$. Corresponding direct calculation time (black, dash-dotted) was 8.83E3 s. Simulations ran in serial on Intel Xeon CPU.

Note that the virtual 2D treecodes perform significantly better than the original treecode at all orders. A significant portion of this performance gain is a result of using six individual 2D treecodes for each face instead of a 3D treecode where all targets are clustered at the boundary. The use of flattened coefficient arrays also gives a small performance gain over the original coefficient storage scheme.

In Fig. 2.6, we display the relative ℓ_2 and ℓ_∞ errors versus memory usage for the three treecode versions shown in Fig. 2.5. Again, we observe the same memory usage behavior as seen in Fig. 2.4, in which the use of flattened coefficient arrays significantly decreases memory use at high treecode expansion orders.

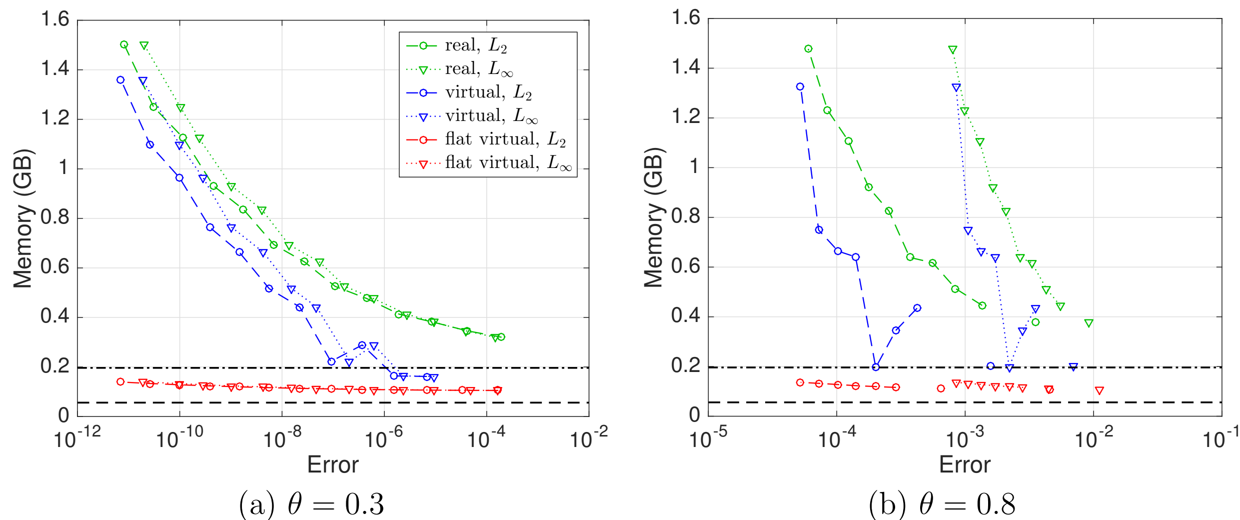


Figure 2.6: Boundary grid test case, 1E5 sources with targets on the six faces of a $1000 \times 1000 \times 1000$ point grid, L_2 (solid, $\circ$) and L_∞ (dotted, ∇) errors from exact solution versus total memory usage for original explicit treecode (green), virtual 2D treecode (blue), and virtual 2D treecode with flattened coefficient arrays (red), for treecode expansion order from 1 to 15, increasing from right to left, and multipole acceptance criterion of (a) $\theta = 0.3$ and (b) $\theta = 0.8$. Corresponding direct calculation memory usage was 0.196 GB with real targets (black, dash-dotted) and 0.056 GB with virtual targets (black, dashed). Simulations ran in serial on Intel Xeon CPU.

2.3.3.3 Parallel performance

We additionally investigate the parallel performance of our improved treecodes to demonstrate their strong scaling in comparison to the direct calculation case. In the current setting, in which we consider a much greater number of targets than sources, we distribute the targets among the processors and replicate all sources and their associated trees on all processors. Figure 2.7(a) shows a sample problem with randomly distributed sources and

targets on a grid, while Figs. 2.7(b) and (c) shows two variations of the parallelization strategy. Figure 2.7(b) depicts a “disjoint strips” partitioning strategy in which the targets on each processor are compact. Figure 2.7(c) depicts a “global” partitioning strategy in which targets on each processor are dispersed across the entire domain. We note that Fig. 2.7(b) is the implied parallelization strategy for the RISM problem discussed in Chapter 5. For both of our previous test cases, we consider the total time and parallel speedup for n number of processors, where speedup is defined as the ratio of the total time for 1 processor to the total time for n processors. We consider the performance on up to 16 processors.

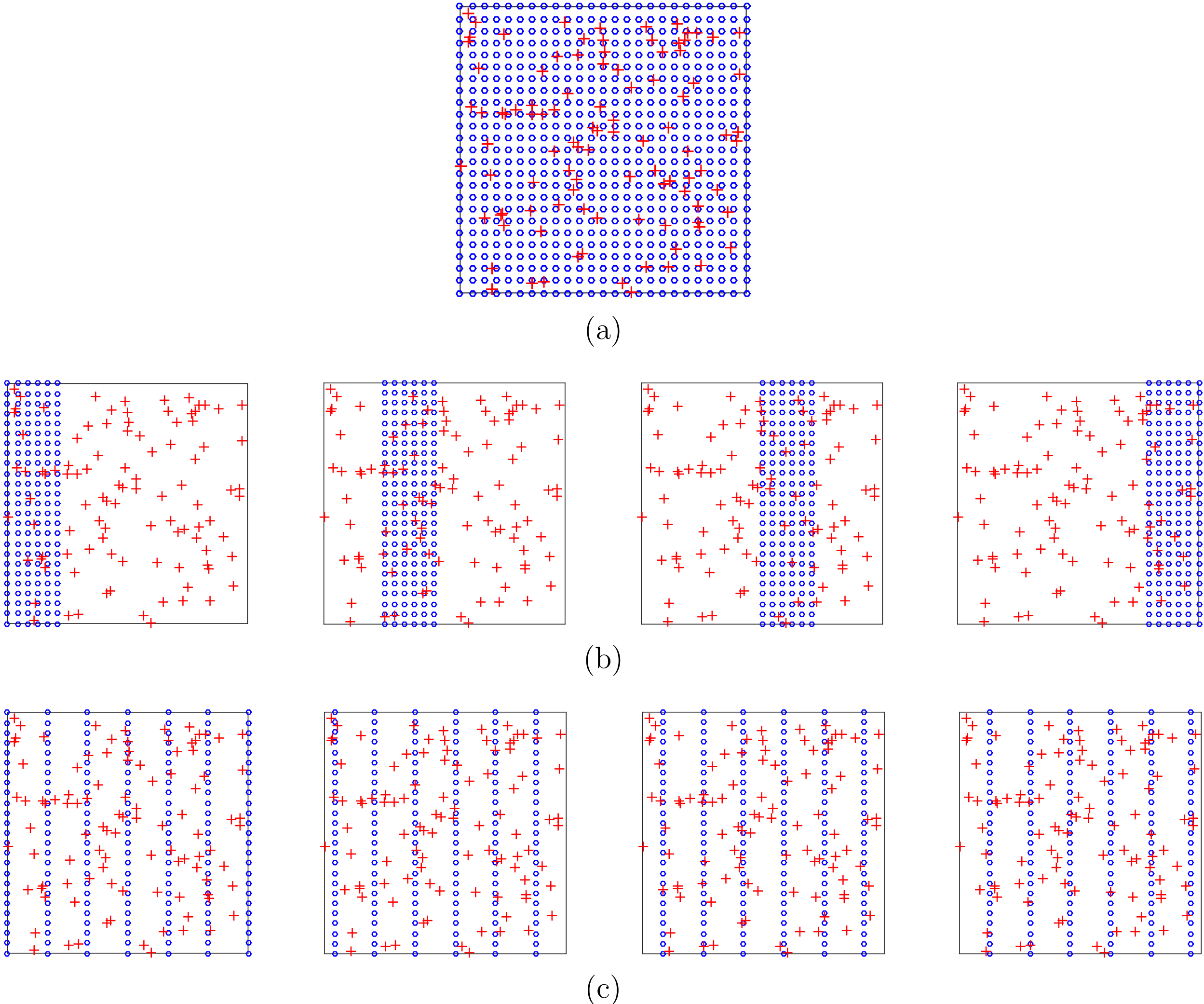


Figure 2.7: Partition strategies for randomly distributed sources and grid-placed targets. Sources are represented by red markers, targets by blue markers. (a) depicts an initial uniform random distribution of sources and uniform grid of targets, (b) depicts a disjoint partitioning strategy across four processors, and (c) depicts a global partitioning strategy across four processors.

In Fig. 2.8, we demonstrate parallelization of the volume grid test case. We display here the number of processors versus (a) total time and (b) speedup for direct calculation (black, dashed, ◦), the original explicit treecode with real targets and 3D coefficient arrays (green), and the 2D virtual treecode with flattened coefficient arrays (red). Both treecodes in this example were calculated with parameters $\theta = 0.3$ and $p = 12$, and exhibited a relative ℓ_2 error of approximately 4E–10. Note that, for all number of processors, the flat virtual treecode takes significantly less time than either the direct calculation or real treecode. Furthermore, the speedup of the flat virtual treecode is very close to the near perfect speedup of the direct calculation, while the speedup of the real treecode trails off significantly. Additionally note that the "global" partitioning strategy (dash-dotted, ▽) depicted in Fig. 2.7(c) exhibits better strong scaling than the "disjoint strips" strategy (dashed, ◦) depicted in Fig. 2.7(b). We see very similar behavior in Fig. 2.9 for the boundary grid test case.

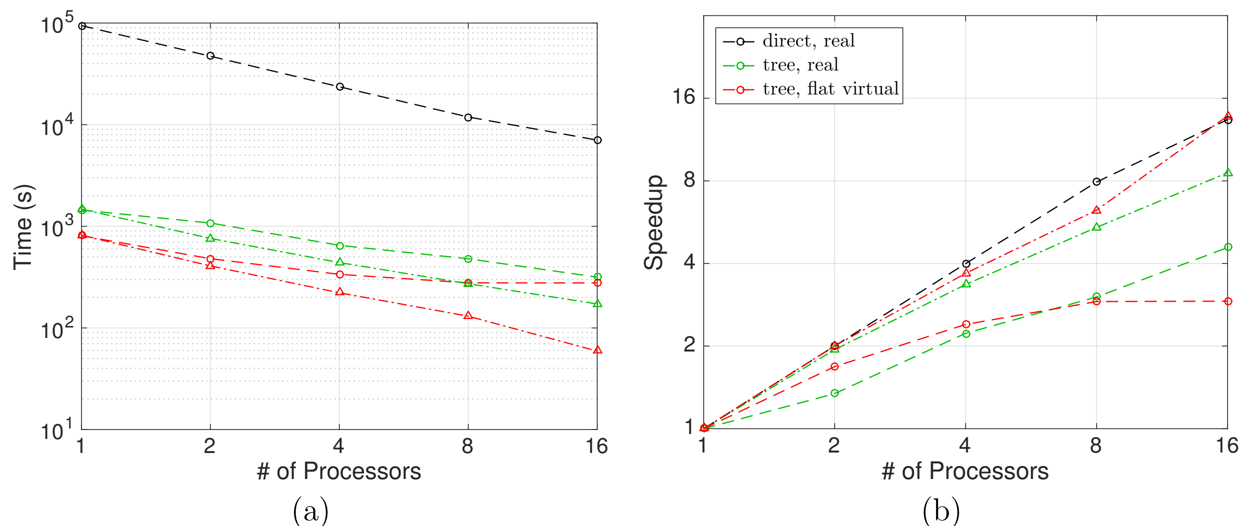


Figure 2.8: Volume grid test case, 1E5 sources with 64E6 targets on a regular grid, number of processors versus (a) total time and (b) speedup for direct calculation (black, dashed, ◦), original explicit treecode with real targets and 3D coefficient arrays (green), and virtual treecode with flattened coefficient arrays (red), using parallelization strategies depicted in Fig. 2.7(b) (dashed, ◦) and Fig. 2.7(c) (dash-dotted, ▽), treecode parameters $\theta = 0.3$ and $p = 12$. Relative ℓ_2 error approximately 4E–10. Simulations ran on Intel Xeon CPUs.

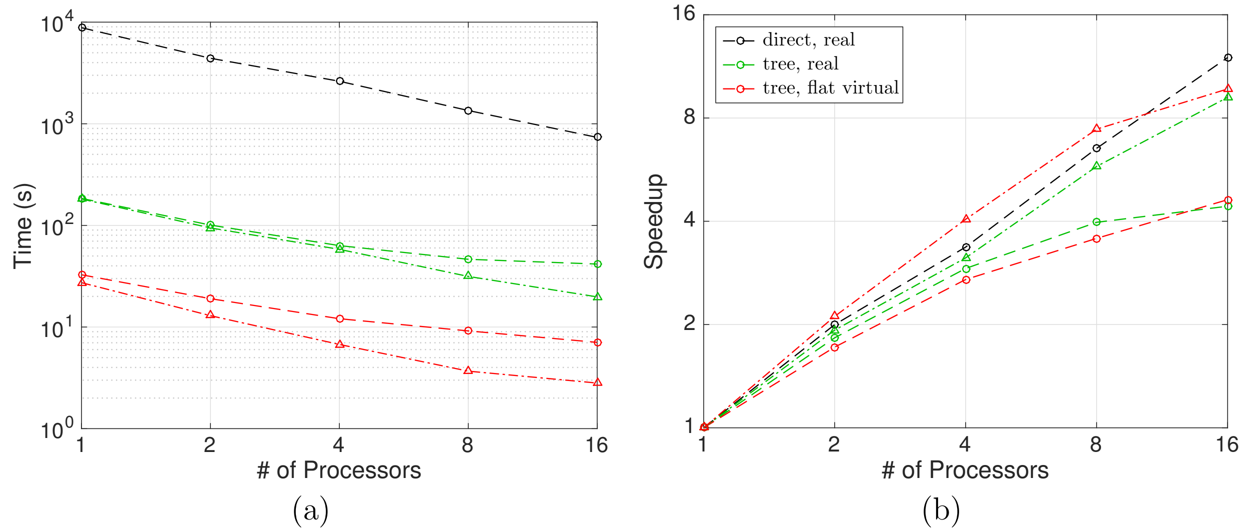


Figure 2.9: Boundary grid test case, 1E5 sources with targets on the six faces of a $1000 \times 1000 \times 1000$ point grid, number of processors versus (a) total time and (b) speedup for direct calculation (black, dashed, $\circ$), original explicit treecode with real targets and 3D coefficient arrays (green), and 2D virtual treecode with flattened coefficient arrays (red), using parallelization strategies depicted in Fig. 2.7(b) (dashed, $\circ$) and Fig. 2.7(c) (dash-dotted, $\triangledown$), using parallelization strategies depicted in Fig. 2.7(b) (dashed, $\circ$) and Fig. 2.7(c) (dash-dotted, $\triangledown$), treecode parameters $\theta = 0.3$ and $p = 12$. Simulations ran on Intel Xeon CPUs.

CHAPTER 3

BLDTT: GPU-Accelerated Barycentric Lagrange Dual Tree Traversal

This chapter details the development of the GPU-accelerated barycentric Lagrange dual tree traversal (BLDTT) algorithm, which forms the primary result of this thesis. The BLDTT builds on previous joint work with Nathan Vaughn which previously developed the GPU-accelerated barycentric Lagrange treecode (BLTC), a treecode method based on polynomial interpolation; this work is described in Nathan's thesis [5] and in our joint paper [6]. §3.1 provides background on dual tree traversal fast summation methods, kernel-independent approaches to fast summation, and GPU computing. §3.2 describes the BLDTT algorithm. §3.3 details the implementation of the BLDTT in BaryTree, for multi-node GPU systems. §3.5 provides results comparing the BLDTT to the BLTC across a wide range of problems. Further implementation details are presented in Appendix A. The most recent version of the BaryTree software package is available at `github.com/Treecodes/BaryTree`. The content of this chapter follows the work of [7], which is in revision and submitted to Comput. Phys. Commun.

3.1 Background

3.1.1 Dual tree traversal methods

Dual tree traversal (DTT) methods are a class of tree-based fast summation method which reduce the cost of computing particle interactions for N particles to $O(N)$. Early work by Appel developed a DTT-style method using monopole approximations [42]. Appel's algorithm was written in the context of computing acceleration of gravitational bodies in astrophysical simulations. In the algorithm, the particles are partitioned into a binary tree, whose leaves were single particles and whose internal nodes contained clumps of particles. The mass of an internal node is the sum of the masses of all particles in its two child nodes, and the position of a node is the center of mass of its child nodes. Each internal node also has an associated radius. To compute the acceleration on one node A caused by another node B

(and vice versa), one first determines if the nodes are well-separated by some criterion. If so, then the acceleration on A due to B and B due to A is approximated using the associated masses and positions of the nodes. If A and B are not well-separated, then the node with the largest radius is opened further. For instance, if the radius of B is larger than the radius of A, then the interaction between A and $B_{\text{left child}}$ and the interaction between A and $B_{\text{right child}}$ are computed by the procedure described above. The algorithm was later shown to be $O(N)$ [50].

Dehnen introduced a similar algorithm [49], again in the context of stellar dynamics. Dehnen positioned the algorithm as an alternative to Barnes–Hut treecodes, arguing that the DTT approach had the robustness to highly heterogeneous particle distributions of a Barnes–Hut treecode with the $O(N)$ scaling of an FMM. Unlike the spherical harmonics of FMMs at the time, Dehnen used low order Taylor expansions targeting the lower accuracy requirements of stellar simulations. This algorithm and its successors are often referred to as DTTs or DTT FMMs.

DTT treecode/ FMM approaches have been shown to adapt well to non-uniform particle distributions relative to other FMM approaches [51, 52, 53]. Successive work introduced parallel DTTs on many core architectures by implementing a task-based programming approach [54, 55, 56]. Of particular note in astrophysics are implementation in exaFMM of Taura [54] and the pfalcON code of Lange and Fortin [56]. DTTs with Taylor expansions have additionally been applied to molecular dynamics, including the work of Lorenzen et al. on periodic condensed phase systems [57] and the work of Coles and Masella on polarizable force fields [58]. Fortin and Touche presented an implementation of a DTT on integrated GPUs [59].

3.1.2 Kernel-independent methods

Tree-based fast summation methods originally relied on analytic series expansions specific to a given kernel $G(\mathbf{x}, \mathbf{y})$. For example, in the case of the Coulomb and Yukawa kernels, multipole expansions were used in the FMM [44, 60], and Cartesian Taylor expansions were used in the TC [2, 4]. Alternative approximations for the Coulomb kernel were introduced involving numerical discretization of the Poisson integral formula [61] and multipole expansions at pseudoparticles [62]. Eventually kernel-independent methods were developed that require only kernel evaluations and are suitable for a large class of kernels. Among these, the kernel-independent FMM (KIFMM) uses equivalent densities defined on proxy surfaces [63], the black-box FMM (bbFMM) uses polynomial interpolation and SVD compression [64], and the barycentric Lagrange treecode (BLTC) uses barycentric Lagrange interpolation [65, 66]. A number of related proxy point methods have recently been developed using skeletonized

interpolation [67] and interpolative decomposition [68, 69]. Several of the kernel-independent fast summation methods have been parallelized for multi-core CPU systems [70, 71, 72, 73, 74, 75, 69].

3.1.3 GPU-based compute for fast summation methods

Graphics Processing Units (GPUs) were initially developed to handle the high throughput, highly parallelizable task of computing and updating the contents of a frame buffer intended for display on an output device. This task involves the execution of relatively simple, uniform computations on millions of frame buffer entries, and thus the architecture of GPUs has been specifically developed for a quintessentially SIMD (single instruction, multiple data) task. In recent years, computational scientists have increasingly used this special architecture of GPUs for a wider range of computational tasks, a practice often referred to as general-purpose computing on graphics processing units (GPGPUs).

Programming for GPUs require a significantly different approach to algorithm design from parallel paradigms for CPUs. GPUs have an incredibly fine-grained parallelism, with thousands of concurrent threads, but have very limited memory organized in an architecturally-dependent hierarchy with high communication costs. Copying data from main memory to the GPU is in particular very expensive. Divergent threads on the GPU can utterly demolish any computational speedups.

In this section, we overview the architecture of GPUs and the typical GPU programming model, and present previous work on GPU acceleration of fast summation methods.

3.1.3.1 Overview of GPU architecture

A GPU is a processor array consisting of Streaming Multiprocessors (SMs), each of which contains many Stream Processors (SPs), the fundamental computing unit of the GPU. The basic execution unit of the programming abstraction for GPUs is the thread, which is mapped to a single SP. A thread-block, or block, is a group of SPs, all of which are contained on the same SM. Within a block, threads are divided into groups of 32 called warps. Warps are the actual execution unit on the GPU, so instructions for threads within a given block are executed warp-by-warp.

The programmer sends work to the GPU by a function known as a compute kernel. The programmer arranges the parallelism for kernel execution by declaring the number of blocks and the number of threads per block that the kernel will utilize. Because each thread can access its thread ID and block ID, parallel operations on, for instance, a 1D array, are typically

performed by indexing the array as a function of thread and block IDs. Note that, because instructions are executed by warp, in general, a declared block size should be a multiple of 32.

The GPU memory is similarly hierarchical in nature. At the top level, the GPU has its own global memory, separate from the CPU's memory and accessible by all threads and blocks. The GPU also has a faster L2 cache available to all threads. Each SM additionally has several types of memory: an L1 cache, shared memory, and a read-only cache. These SM-local memory types can only be used by threads within the SM. The L1 cache in each SM can transfer data to and from the L2 cache. The read-only cache can only be read by threads, but its judicious use can decrease bandwidth pressure on the L1 cache and shared memory. Note that these forms of SM memory can be loaded into GPU thread registers much faster than the global GPU memory or L2 cache. Additionally, each thread has its own local memory, which can load memory from any location on the GPU. Of course, memory loads from the memory of the thread's SM are the fastest.

To give some context to the scale and size of these structures, consider the Kepler GK110 NVIDIA GPU available on Michigan's Flux computing cluster, whose architecture is shown in Figs. 3.1 and 3.2. The Flux configuration of this GPU has a global memory of size 6GB and L2 cache of 1.5MB. Each of the 14 SMs contain 192 SPs, for a total of 2688 cores. Each SM can share 64KB between L1 cache and shared memory configurable into splits of 16/48, 32/32, or 48/16 KB. The read-only cache of each SM is 48KB. The local memory of each SP contains 255 32-bit registers. The memory hierarchy of the Kepler GK110 is shown in Fig. 3.3.

3.1.3.2 Previous work

The direct sum in Eq. 2.1 is well suited for GPU computing; this is because the kernel evaluations $G(\mathbf{x}_i, \mathbf{x}_j)$ can be computed concurrently, where the targets $\mathbf{x}_i$ provide an outer level of parallelism, while the sources $\mathbf{x}_j$ provide an inner level of parallelism. Early GPU implementations of direct summation achieved a 25× speedup over an optimized CPU implementation [77] and a 250× speedup over a portable C implementation [78]. However, these codes still scale like $O(N^2)$ and there is great interest in implementing the sub-quadratic scaling tree-based methods on GPUs, although this is challenging due to the complexity of these methods in comparison with direct summation.

Several previous attempts to implement a range of treecodes on GPUs are worth mentioning. Many early attempts have focused on Barnes–Hut Cartesian treecodes, i.e., treecodes that use a zero order Taylor approximation for the particle-cluster interactions. Zero order treecodes avoid some of the memory management difficulties that come with higher order treecodes;

Figure 3.1: Structure of the standard configuration Kepler GK110. 15 SMs have access to a device-wide L2 cache and global GPU memory through memory controllers. The GPU communicates with the host machine through a PCI interface. Source: NVIDIA Kepler GK110/120 Whitepaper [76].

because every tree cluster has an associated set of moments, and the number of these moments is $O(p^3)$ where p is the Taylor expansion order, memory requirements grow rapidly.

In 2010, Jiang and coworkers developed a Barnes–Hut treecode that achieved speedups of over 100× for computing forces between gravitational bodies [79]. In 2012, Bédorf and coworkers introduced Bonsai, a fully GPU treecode for gravitational simulations, in which even the tree-building was performed on the GPU [80]. This implementation, however, was limited to small-problem sizes because the entire problem had to fit totally within GPU memory. In 2014, Bédorf introduced a multi-GPU implementation of Bonsai that, in a simulation of the galactic evolution of the Milky Way on 18,600 GPUs, achieved a peak performance of 24.77 petaflops [81].

Additional efforts in this area include a GPU treecode for gravitational simulations and GPU FMM for turbulence simulations [82], a GPU treecode that replaced the pointer-chasing

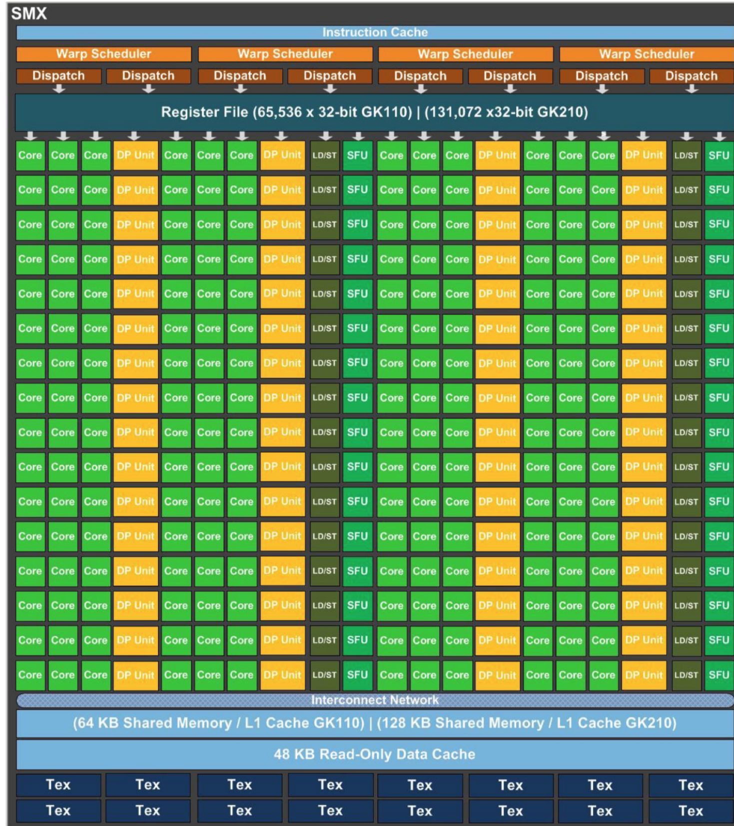


Figure 3.2: Structure of an SM on the Kepler GK110. The SM contains 192 cores which each have access to SM-wide shared memory, L1 cache, and read-only memory. Source: NVIDIA Kepler GK110/120 Whitepaper [76].

recursion often used in CPU treecodes by an iteration over arrays [83], a GPU treecode and GPU FMM for vortex ring dynamics [84], a GPU bbFMM with optimized multipole-to-local and direct short range computations [75], More recently, Fortin and Touche implemented a dual tree traversal scheme on integrated GPUs for gravitational simulations [59].

3.1.4 Barycentric Lagrange interpolation

We briefly review the barycentric Lagrange form of polynomial interpolation in 1d [65]. Given a function $f(x)$ and $n+1$ points $s_k, k = 0 : n$, the Lagrange form of the interpolating polynomial is

$$p_n(x) = \sum_{k=0}^{n} f(s_k) L_k(x), \tag{3.1}$$

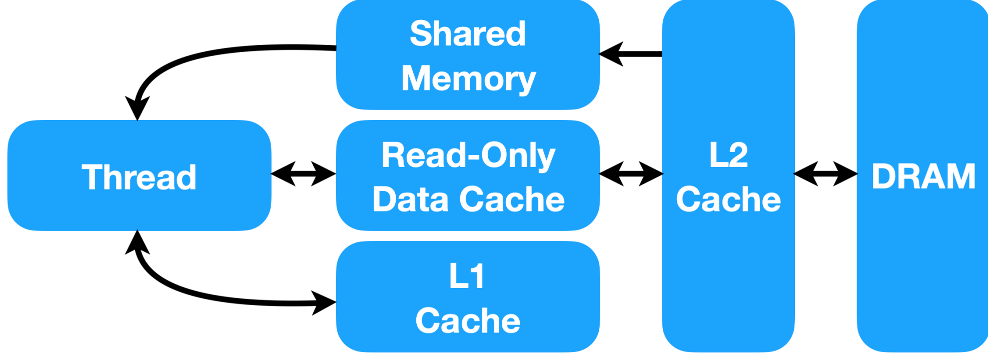


Figure 3.3: Basic memory hierarchy of the Kepler GK110. The device-wide global memory is accessed by individual threads through the hierarchy of device-wide L2 cache and SM-wide L1 cache. The SMs can be configured to adjust the split between the shared memory and L1 cache. Greater shared memory gives more user control over memory optimization, but at the expense of bandwidth for global memory fetches. Figure based on documentation of memory hierarchy in [76].

where the Lagrange polynomial $L_k(x)$ in barycentric form is

$$L_k(x) = \frac{\dfrac{w_k}{x - s_k}}{\displaystyle\sum_{k'=0}^{n} \frac{w_k}{x - s'_k}}, \quad w_k = \frac{1}{\prod_{j=0, j\neq k}^{n}(s_k - s_j)}, \quad k = 0 : n. \tag{3.2}$$

This work employs Chebyshev points of the second kind,

$$s_k = \cos\theta_k, \quad \theta_k = \pi k/n, \quad k = 0 : n, \tag{3.3}$$

in which case the interpolation weights are given by

$$w_k = (-1)^k \delta_k, \quad k = 0 : n, \tag{3.4}$$

where $\delta_k = 1/2$ if $k = 0$ or n, and $\delta_k = 1$ otherwise [85, 65]. The algorithms described below use barycentric Lagrange interpolation in 3D rectangular boxes with interpolation points $\mathbf{s_k} = (s_{k_1}, s_{k_2}, s_{k_3})$ given by a Cartesian tensor product grid of Chebyshev points adapted to the box. Depending on the context, the interpolation points $\mathbf{s_k}$ may also be referred to as proxy particles.

3.1.5 Present work

The present work contributes a GPU-accelerated tree-based fast summation method called barycentric Lagrange dual tree traversal (BLDTT). The BLDTT employs several techniques from previous tree-based methods, including (1) the DTT algorithmic structure [42, 49], (2) barycentric Lagrange interpolation [65, 66, 6], and (3) upward and downward passes similar to those in the FMM, but adapted to the context of barycentric Lagrange interpolation. The BLDTT has several additional features that should be noted. The algorithm replaces well-separated particle-particle interactions by adaptively chosen particle-cluster, cluster-particle, and cluster-cluster approximations, where the clusters are represented by proxy particles at Chebyshev grid points. The approximations are done with barycentric Lagrange interpolation and require only kernel evaluations, hence the BLDTT is kernel-independent. Similar to other tree-based fast summation methods, the BLDTT has a precompute phase and a compute phase; the precompute phase scales like $O(N \log N)$ with a small prefactor, while the compute phase scales like $O(N)$, so the observed scaling of the BLDTT is essentially $O(N)$.

As will be shown, the barycentric Lagrange approximations resemble the direct sum in Eq. 2.1 and they can be efficiently mapped onto GPUs. Based on this observation we present an OpenACC GPU implementation of the BLDTT with MPI remote memory access for distributed memory parallelization. The performance of the BLDTT is documented for calculations with different problem sizes, particle distributions, geometric domains, and interaction kernels, unequal target and source particles. Comparison with our earlier particle-cluster barycentric Lagrange treecode (BLTC) shows the superior performance of the BLDTT. In particular, on a single GPU for problem sizes ranging from N=1E5 to 1E8, the BLTC has $O(N \log N)$ scaling, while the BLDTT has $O(N)$ scaling. In addition, MPI strong scaling results are presented for the BLTC and BLDTT using N=64E6 particles on up to 32 GPUs.

The remainder of this chapter is organized as follows. §3.2 describes the barycentric Lagrange dual tree traversal (BLDTT) fast summation method. §3.3 describes our implementation of the BLDTT using MPI remote memory access for distributed memory parallelization and OpenACC for GPU acceleration. §3.5 presents numerical results for several test cases.

3.2 Description of BLDTT fast summation method

3.2.1 Algorithm overview

The BLDTT fast summation method computes the potential at a set of M target particles due to interactions with a set of N source particles; Eq. 2.1 is a special case in which the targets and sources refer to the same set of particles. First, two hierarchical trees of particle

clusters are built, one for the target particles and one for the source particles, where each cluster is a rectangular box; clusters in the target tree are denoted C_t and clusters in the source tree are denoted C_s. The computed potential at a target particle $\mathbf{x}_i$ has contributions from four types of interactions as determined by the dual tree traversal algorithm described below. The four types are direct particle-particle (PP) interactions of nearby particles, and particle-cluster (PC), cluster-particle (CP), and cluster-cluster (CC) approximations of well-separated clusters.

Algorithm 3.1 is a high-level overview of the BLDTT. Lines 1-4 describe the input consisting of target and source particle data, interpolation degree n, MAC parameter θ, and the maximum number of particles in the leaves of each tree, M_0, N_0. Line 5 describes the output potentials. Line 6 builds the target tree and source tree containing target clusters C_t and source clusters C_s. Line 7 is the upward pass to compute proxy charges $\widehat{q}_{\mathbf{k}}$ at proxy particles $\mathbf{s}_{\mathbf{k}}$ in source clusters C_s. Line 8 is the dual tree traversal to compute PP, PC, CP, and CC interactions. Line 9 is the downward pass to interpolate potentials from proxy particles $\mathbf{t}_{\boldsymbol{\ell}}$ to target particles $\mathbf{x}_i$ in target clusters C_t. The steps will be described in detail below.

Algorithm 3.1 High-level overview of the barycentric Lagrange dual tree traversal (BLDTT) fast summation method.

1: **input** target particles $\mathbf{x}_i, i = 1 : M$
2: **input** source particles and charges $\mathbf{y}_j, q_j, j = 1 : N$
3: **input** interpolation degree n, MAC parameter θ
4: **input** max particles per target leaf M_0, max particles per source leaf N_0
5: **output** potentials $\phi(\mathbf{x}_i), i = 1 : M$
6: build target tree and source tree
7: upward pass to compute proxy charges $\widehat{q}_{\mathbf{k}}$ at proxy particles $\mathbf{s}_{\mathbf{k}}$ in source clusters
8: dual tree traversal to compute PP, PC, CP, CC interactions
9: downward pass to interpolate potentials from proxy particles $\mathbf{t}_{\boldsymbol{\ell}}$ to target particles $\mathbf{x}_i$

3.2.2 Tree building

The target and source trees are constructed by the same routines, described here for the target tree. The maximum number of particles per leaf is a user-specified parameter, M_0 for the target tree and N_0 for the source tree. The root cluster is the minimal bounding box containing all target particles. The root is recursively divided into child clusters, terminating when a cluster contains fewer than M_0 particles. Division occurs at the midpoint of the cluster; in general the cluster is bisected in all three coordinate directions, resulting in eight child clusters, with two exceptions. First, a cluster is divided into only two or four children

in order to maintain a good aspect ratio, that is, a ratio of longest to shortest side lengths no greater than $\sqrt{2}$. Second, a cluster is divided into only two or four children to avoid creating leaf clusters with fewer than $M_0/2$ particles on average; in particular, if a cluster contains between M_0 and $2M_0$ particles, it is divided into two children, and if it contains between $2M_0$ and $4M_0$ particles, it is divided into four children. Upon creation, each cluster is shrunk to the minimal bounding box containing its particles, and a tensor product grid of Chebyshev points adapted to the box is created; these are also referred to as proxy particles. After building the trees, the BLDTT performs the upward pass, dual tree traversal, and downward pass, but before discussing these steps, the next subsection describes the four types of interactions which are eventually combined to compute potentials.

3.2.3 Four types of interactions

Figure 3.4 depicts the four types of interactions between a target cluster C_t (left, blue) and a source cluster C_s (right, red), where dots are target/source particles $\mathbf{x}_i, \mathbf{y}_j$, and crosses are target/source proxy particles $\mathbf{t}_{\boldsymbol{\ell}}, \mathbf{s}_{\mathbf{k}}$. Also shown are the target/source cluster radii r_t, r_s, and the target-source cluster distance R. These diagrams depict 2D versions of the interactions; in practice the particles are distributed in 3D and the clusters are rectangular boxes. In general, clusters can interact via their particles (dots) or their proxy particles (crosses). Figure 3.4 shows the four cases: (a) C_t and C_s use particles (PP), (b) C_t uses particles and C_s uses proxy particles (PC), (c) C_t uses proxy particles and C_s uses particles (CP), (d) C_t and C_s both use proxy particles (CC). The interactions are described in detail below. To simplify notation, we present the interactions in 1D instead of 3D, and thus replace the bold 3-vector $\mathbf{x}_i$ by the non-bold scalar x_i. We note that the extension to 3D is straightforward using tensor products.

Particle-particle interaction. Figure 3.4a depicts a PP interaction. In this case the PP potential at a target particle $x_i \in C_t$ due to direct interaction with the source particles $y_j \in C_s$ is denoted by

$$\phi_{PP}(x_i, C_t, C_s) = \sum_{y_j \in C_s} G(x_i, y_j) q_j, \quad x_i \in C_t. \tag{3.5}$$

Particle-cluster approximation. Figure 3.4b depicts a PC interaction. The kernel is approximated by holding x_i fixed and interpolating with respect to y_j at the proxy particles s_k in C_s,

$$G(x_i, y_j) \approx \sum_{k=0}^{n} G(x_i, s_k) L_k(y_j), \quad x_i \in C_t, \quad y_j \in C_s. \tag{3.6}$$

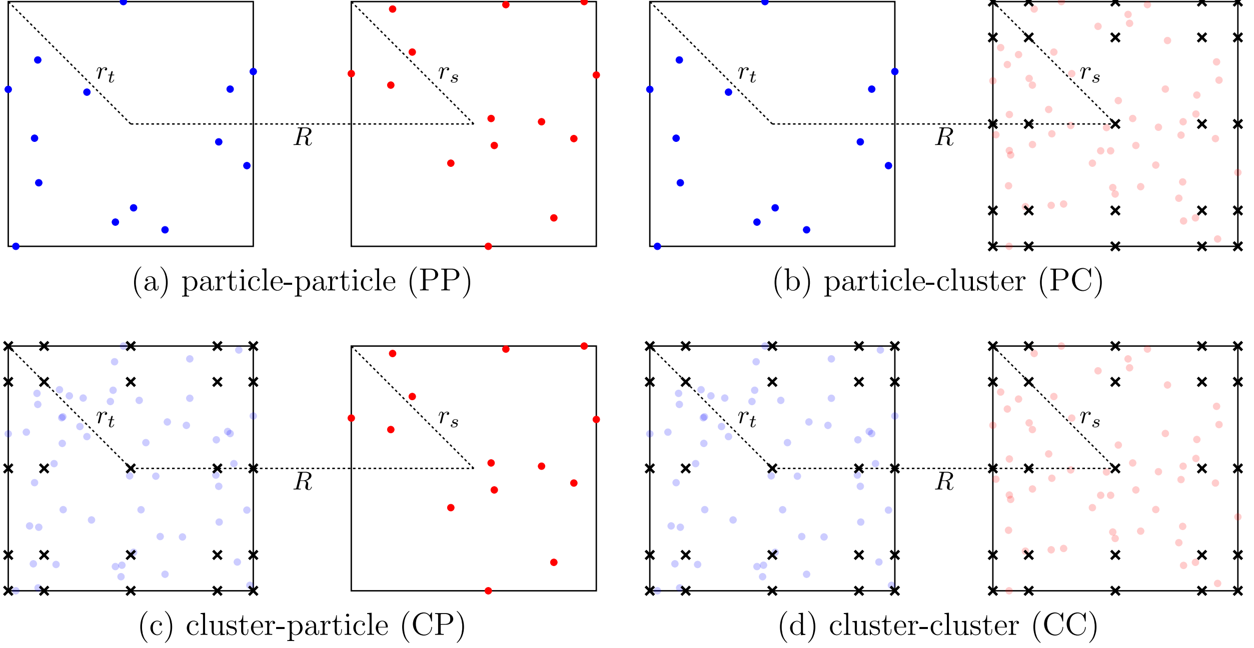


Figure 3.4: Four types of interactions are used, in each case the target cluster C_t on the left interacts with the source cluster C_s on the right, (a) direct particle-particle interaction (PP), (b) particle-cluster approximation (PC), (c) cluster-particle approximation (CP), (d) cluster-cluster approximation (CC), dots are target/source particles $\mathbf{x}_i, \mathbf{y}_j$, crosses are target/source proxy particles $\mathbf{t}_{\boldsymbol{\ell}}, \mathbf{s}_{\mathbf{k}}$, target/source cluster radii r_t, r_s, target-source cluster distance R.

Substituting into the particle-particle interaction and rearranging terms yields the PC potential,

$$\phi_{PC}(x_i, C_t, \widehat{C}_s) = \sum_{k=0}^{n} G(x_i, s_k)\widehat{q}_k, \quad x_i \in C_t, \tag{3.7}$$

where the proxy charges $\widehat{q}_k$ of the proxy particles s_k are

$$\widehat{q}_k = \sum_{y_j \in C_s} L_k(y_j) q_j. \tag{3.8}$$

Equation 3.7 uses $C_t, \widehat{C}_s$ to indicate that the target particles x_i interact with the proxy source particles s_k.

Cluster-particle approximation. Figure 3.4c depicts a CP interaction. The kernel is approximated by interpolating with respect to x_i at the proxy particles t_ℓ in C_t and holding y_j fixed,

$$G(x_i, y_j) \approx \sum_{\ell=0}^{n} L_\ell(x_i) G(t_\ell, y_j), \quad x_i \in C_t, \quad y_j \in C_s. \tag{3.9}$$

Substituting into the particle-particle interaction and rearranging terms yields the CP potential,

$$\phi_{CP}(x_i, \widehat{C}_t, C_s) = \sum_{\ell=0}^{n} \phi(t_\ell, \widehat{C}_t, C_s) L_\ell(x_i), \quad x_i \in C_t, \tag{3.10}$$

where the CP proxy potential $\phi(t_\ell, \widehat{C}_t, C_s)$ is

$$\phi(t_\ell, \widehat{C}_t, C_s) = \sum_{y_j \in C_s} G(t_\ell, y_j) q_j, \quad t_\ell \in \widehat{C}_t. \tag{3.11}$$

Equation 3.10 and Eq. 3.11 use $\widehat{C}_t, C_s$ to indicate that the proxy target particles of C_t interact with the source particles y_j. Equation 3.10 interpolates from the proxy target particles t_ℓ to the target particles x_i.

Cluster-cluster interaction. Figure 3.4d depicts a CC interaction. The kernel is interpolated with respect to x_i at the proxy particles t_ℓ in C_t and with respect to y_j at the proxy particles s_k in C_s,

$$G(x_i, y_j) \approx \sum_{k=0}^{n} \sum_{\ell=0}^{n} L_\ell(x_i) G(t_\ell, s_k) L_k(y_j), \quad x_i \in C_t, \quad y_j \in C_s. \tag{3.12}$$

Substituting into the particle-particle interaction and rearranging terms yields the CC potential,

$$\phi_{CC}(x_i, \widehat{C}_t, \widehat{C}_s) = \sum_{\ell=0}^{b} \phi(t_\ell, \widehat{C}_t, \widehat{C}_s) L_\ell(x_i), \quad x_i \in C_t, \tag{3.13}$$

where the CC proxy potential $\phi(t_\ell, \widehat{C}_t, \widehat{C}_s)$ is

$$\phi(t_\ell, \widehat{C}_t, \widehat{C}_s) = \sum_{k=0}^{n} G(t_\ell, s_k) \widehat{q}_k, \quad t_\ell \in \widehat{C}_t, \tag{3.14}$$

and the proxy charges $\widehat{q}_k$ were defined in Eq. 3.8. Equation 3.13 and Eq. 3.14 use $\widehat{C}_t, \widehat{C}_s$ to indicate that the proxy target particles t_ℓ interact with the proxy source particles s_k. Equation 3.13 interpolates from the proxy target particles t_ℓ to the target particles x_i.

The following three subsections describe the rest of Algorithm 3.1, comprising the upward pass, dual tree traversal, and downward pass.

3.2.4 Upward pass

The upward pass computes the proxy charges $\widehat{q}_k$ defined in Eq. 3.8 for the proxy particles s_k in each source cluster $\widehat{C}_s$ in the source tree, as required in the PC and CC approximations

in Eq. 3.7 and Eq. 3.14. Note that each source particle y_j contributes to the proxy charges $\widehat{q}_k$ of exactly one cluster at a given level of the tree. Hence with N source particles and tree depth $O(\log N)$, computing the proxy charges directly by Eq. 3.8 requires $O(N \log N)$ operations.

The BLDTT algorithm uses an alternative approach as follows. Let C_s denote a parent cluster with Lagrange polynomials $L_k(y)$ and interpolation points s_k, and let $C_s^i, i = 1:8$ denote the eight child clusters with Lagrange polynomials $L^i_{k_i}(y)$ and interpolation points s_{k_i}. As an alternative to Eq. 3.8, the proxy charges of the parent $\widehat{q}_k$ are computed from the proxy charges of the children $\widehat{q}_{k_i}$ by the expression

$$\widehat{q}_k = \sum_{i=1}^{8} \sum_{k_i=0}^{n} L_k(s_{k_i}) \widehat{q}_{k_i}, \tag{3.15}$$

which is derived in §3.2.7. The derivation uses the definitions of the parent and child proxy charges and the relation

$$L_k(y) = \sum_{k_i=0}^{n} L_k(s_{k_i}) L_{k_i}(y), \tag{3.16}$$

which is a special case of Eq. 3.1.

The upward pass starts by computing the proxy charges of the leaves of the source tree using Eq. 3.8 and then ascending to the root by Eq. 3.15. This is analogous to the upward pass in the FMM [43] where the multipole moments of a parent cluster are obtained from the moments of its children. As with the FMM, computing the proxy charges this way requires $O(N)$ operations, which can be seen as follows. Computing the proxy charges for the leaves requires $O(n^3 N)$ operations; each of the N source particles contributes to one leaf, which contains $O(n^3)$ proxy particles (in 3D). Then, evaluating the child-to-parent relation in Eq. 3.15 for each parent proxy charge requires $O(n^6)$ operations and is independent of the number of source particles. Since there are $O(N)$ clusters in the tree, ascending the tree requires an additional $O(n^6 N)$ operations. Hence the operation count for the BLDTT upward pass is $O(n^3 N) + O(n^6 N) = O(N)$, as it is for the FMM.

3.2.5 Dual tree traversal

The dual tree traversal determines which pairs of clusters in the target and source trees interact by one of the four options described above (PP, PC, CP, CC). Before the traversal starts, two sets of potentials are initialized to zero, potentials $\phi(x_i)$ at the target particles and potentials $\phi(t_\ell)$ at the proxy target particles. In the course of the traversal, the potentials $\phi(x_i)$ are incremented due to PP and PC interactions, and the potentials $\phi(t_\ell)$ are incremented

due to CP and CC interactions. Following the dual tree traversal, the $\phi(t_\ell)$ are interpolated to the target particles x_i and combined with the $\phi(x_i)$ in the downward pass.

The dual tree traversal uses the recursive procedure DTT(C_t, C_s) described in Algorithm 3.2, which takes a target cluster C_t and a source cluster C_s as input. Initially the procedure is called for the root clusters of the target and source trees. In what follows, the clusters are considered to be well-separated if $(r_t + r_s)/R < \theta$, where r_t, r_s are the target and source cluster radii and R is the center-center distance between the clusters.

If C_t and C_s are well-separated (line 2), then they interact in one of four ways depending on the number of particles they contain relative to the number of proxy particles in a cluster, which is denoted by $n_p = (n+1)^3$ in 3D. If C_t and C_s are both large (lines 3-4), then the CC proxy potentials are incremented using Eq. 3.14; else if C_t is large and C_s is small (lines 5-6), then the CP proxy potentials are incremented using Eq. 3.11; else if C_t is small and C_s is large (lines 7-8), then the PC potentials are incremented using Eq. 3.7; else C_t and C_s are both small (line 9) and the PP potentials are incremented using Eq. 3.5.

If C_t and C_s are not well-separated, then the traversal continues as follows. If C_t and C_s are leaves (lines 11-12), then the PP potentials are incremented using Eq. 3.5. Otherwise if C_s is a leaf, then it interacts recursively with the children of C_t (line 13), while if C_t is a leaf, then it interacts recursively with the children of C_s (line 14). Finally if C_t and C_s are both not leaves, then the smaller cluster interacts recursively with the children of the larger cluster (lines 15-17).

Algorithm 3.2 The dual tree traversal approach used in the BLDTT.

```
1: procedure DTT(target cluster C_t, source cluster C_s)
2: if (r_t + r_s)/R < θ then
3:    if |C_t| > n_p and |C_s| > n_p then
4:       increment CC proxy potentials φ(t_ℓ) += φ(t_ℓ, Ĉ_t, Ĉ_s) for all t_ℓ ∈ Ĉ_t by Eq. 3.14
5:    else if |C_t| > n_p and |C_s| ≤ n_p then
6:       increment CP proxy potentials φ(t_ℓ) += φ(t_ℓ, Ĉ_t, C_s) for all t_ℓ ∈ Ĉ_t by Eq. 3.11
7:    else if |C_t| ≤ n_p and |C_s| > n_p then
8:       increment PC potentials φ(x_i) += φ(x_i, C_t, Ĉ_s) for all x_i ∈ C_t by Eq. 3.7
9:    else increment PP potentials φ(x_i) += φ(x_i, C_t, C_s) for all x_i ∈ C_t by Eq. 3.5
10: else
11:    if C_t and C_s are leaves then
12:       increment PP potentials φ(x_i) += φ(x_i, C_t, C_s) for all x_i ∈ C_t by Eq. 3.5
13:    else if C_s is a leaf then for each child C'_t of C_t do DTT(C'_t, C_s)
14:    else if C_t is a leaf then for each child C'_s of C_s do DTT(C_t, C'_s)
15:    else
16:       if |C_s| < |C_t| then for each child C'_t of C_t do DTT(C'_t, C_s)
17:       else for each child C'_s of C_s do DTT(C_t, C'_s)
```

The DTT yields potentials $\phi(x_i)$ due to PP and PC interactions and proxy potentials $\phi(t_\ell)$ due to CP and CC interactions. In the case of N homogeneously distributed source and target particles, the operation count of the dual tree traversal has been shown to be $O(N)$ [50, 49].

3.2.6 Downward pass

The downward pass interpolates the proxy potentials $\phi(t_\ell)$ to the target particles x_i and increments the potentials $\phi(x_i)$. This can be done in two ways as described below.

First note that each target particle x_i is contained in a chain of target clusters,

$$x_i \in C_t^1 \subset C_t^2 \subset \cdots \subset C_t^L, \tag{3.17}$$

where the superscript denotes the level in the target tree; level L is the root, and level 1 are the leaves. The target cluster C_t^m at level m in the chain has Lagrange polynomials $L_{k_m}^m(x_i)$ and proxy potentials $\phi(t_{k_m}^m)$ that contribute to $\phi(x_i)$,

$$\phi(x_i) \mathrel{+}= \sum_{m=1}^{L} \sum_{k_m=0}^{n} L_{k_m}^m(x_i)\phi(t_{k_m}^m), \tag{3.18}$$

where $t_{k_m}^m$ are the proxy particles of C_t^m, and the += indicates that the results are aggregated with the potentials $\phi(x_i)$ due to PP and PC interactions previously computed in the DTT. In Eq. 3.18 the inner sum interpolates potential values from the proxy particles $t_{k_m}^m$ to the target particle x_i, and the outer sum accumulates the results from each level in the tree. Computing $\phi(x_i)$ as indicated in Eq. 3.18 requires $O(M \log M)$ operations; the factor M is the number of target particles x_i, the factor $\log M$ is the number of levels in the target tree, and the inner sum requires $O(n^3)$ operations independent of M.

Rather than interpolating from the proxy particles t_m^k directly to the target particle x_i as in Eq. 3.18, we utilize a recursive alternative. In what follows, C_t^m is a parent cluster at level m and C_t^{m-1} is a child cluster at level $m-1$. The procedure interpolates the parent proxy potentials $\phi(t_{k_m}^m)$ to child proxy potentials $\phi(t_{k_{m-1}}^{m-1})$,

$$\phi(t_{k_{m-1}}^{m-1}) \mathrel{+}= \sum_{k_m=0}^{n} L_{k_m}^m(t_{k_{m-1}}^{m-1})\phi(t_{k_m}^m), \tag{3.19}$$

where the += indicates that the results of the parent-to-child interpolation on the right are aggregated with the child proxy potentials $\phi(t_{k_{m-1}}^{m-1})$ due to CP and CC interactions previously computed in the DTT. This procedure starts with the root cluster of the target tree (level

$m = L$) and descends to the leaves (level $m = 1$). Upon reaching the leaves, the proxy potentials $\phi(t^1_{k_1})$ are interpolated to the target particles x_i and aggregated with the PP and PC potentials previously computed in the DTT,

$$\phi(x_i) \mathrel{+}= \sum_{k_1=0}^{n} L^1_{k_1}(x_i)\phi(t^1_{k_1}). \tag{3.20}$$

It should be noted that the expressions for $\phi(x_i)$ in Eq. 3.18 and Eq. 3.20 are equivalent, as shown in §3.2.7.

The recursive form of the downward pass described here is similar to the local-to-local step in the FMM, where the local coefficients are shifted and accumulated, and as in that case the operation count is reduced to $O(M)$. In particular, the parent-to-child interpolation in Eq. 3.19 requires $O(n^6)$ operations (in 3D), independent of the number of target particles M, and the tree contains $O(M)$ clusters, so interpolation from the root down to the leaves by Eq. 3.19 requires $O(n^6M)$ operations. Then the final interpolation from the leaf proxy particles to the target particles by Eq. 3.20 requires $O(n^3)$ operations for each target, yielding complexity $O(n^6M) + O(n^3M) = O(M)$ for the downward pass.

3.2.7 Derivation of upward and downward passes

3.2.7.1 Preliminaries

The derivation of the upward pass and downward pass rely on a property of interpolating polynomials. For a general function $f(x)$, the degree n interpolation polynomial $p_n(x)$ is given by

$$p_n(x) = \sum_{k=0}^{n} f(s_k)L_k(x) \tag{3.21}$$

where the s_k denote a set of interpolation points. Now consider the special case where $f(x) = L_\ell(x)$, a degree n polynomial. Then

$$p_n(x) = \sum_{k=0}^{n} L_\ell(s_k)L_k(x). \tag{3.22}$$

Notice that $p_n(x)$ and $L_\ell(x)$ are both degree n polynomials, and by construction, they take on the same values at the set of $n + 1$ interpolation s_k. Therefore, $p_n(x) = L_\ell(x)$, and we have the following relation

$$L_\ell(x) = \sum_{k=0}^{n} L_\ell(s_k)L_k(x), \tag{3.23}$$

that is, the degree n interpolating polynomial of a degree n polynomial is itself. Equation 3.23 will be used in the sections to follow, where $L_\ell(x)$ and $L_k(x)$ refer to interpolating polynomials in parent and child clusters.

3.2.7.2 Details of upward pass

Recall Eq. 3.8 for the definition of the proxy charges $\widehat{q}_k$ of a parent source cluster C_s in the 1D case,

$$\widehat{q}_k = \sum_{y_j \in C_s} L_k(y_j) q_j, \tag{3.24}$$

where y_j, q_j are the source cluster particles and charges, and $L_k(y)$ are the Lagrange polynomials associated with the cluster. Also consider the parent's eight child clusters C_s^i, $i = 1:8$ with Lagrange polynomials $L_{k_i}^i(y)$, interpolation points s_{k_i}, and proxy charges $\widehat{q}_{k_i}$. The parent proxy charge $\widehat{q}_k$ in Eq. 3.24 can be expressed in terms of the child proxy charges $\widehat{q}_{k_i}$ as follows,

$$\begin{aligned}
\widehat{q}_k &= \sum_{y_j \in C_s} L_k(y_j) q_j && \text{(3.25a)}\\
&= \sum_{i=1}^{8} \sum_{y_j \in C_s^i} L_k(y_j) q_j && \text{(3.25b)}\\
&= \sum_{i=1}^{8} \sum_{y_j \in C_s^i} \left(\sum_{k_i=0}^{n} L_k(s_{k_i}) L_{k_i}(y_j) \right) q_j && \text{(3.25c)}\\
&= \sum_{i=1}^{8} \sum_{k_i=0}^{n} L_k(s_{k_i}) \sum_{y_j \in C_s^i} L_{k_i}(y_j) q_j && \text{(3.25d)}\\
&= \sum_{i=1}^{8} \sum_{k_i=0}^{n} L_k(s_{k_i}) \widehat{q}_{k_i}. && \text{(3.25e)}
\end{aligned}$$

Equation 3.25a is the definition of the parent proxy charges, Eq. 3.25b splits this into the sum over the eight child clusters, Eq. 3.25c uses the relation in Eq. 3.23, Eq. 3.25d rearranges the sums, and Eq. 3.25e applies the definition of the child proxy charges. This result extends in a straightforward way to 3D. In summary, the upward pass ascends the source tree from the leaves to the root, computing the parent proxy charges of each source cluster from the child proxy charges as described here.

3.2.7.3 Details of downward pass

The downward pass adds the CP and CC interactions to the potentials $\phi(x_i)$. For simplicity of notation, the formulas are written in 1d with straightforward extension to 3D, and we consider the case in which the tree has two levels ($L = 2$). Recall Eq. 3.18, which interpolates the proxy potentials $\phi(t^m_{k_m})$ at each level in the tree directly to the target particles x_i,

$$\phi(x_i) \mathrel{+}= \sum_{m=1}^{2} \sum_{k_m=0}^{n} L^m_{k_m}(x_i)\phi(t^m_{k_m}), \tag{3.26}$$

where $L^m_{k_m}(x_i)$ is a Lagrange polynomial associated with the cluster C^m_t at level m containing the target x_i, and $\mathrel{+}=$ indicates that the right side is aggregated with the PP and PC interactions already computed in the DTT. Then the right side of Eq. 3.26 can be rewritten as follows,

$$\begin{aligned}
\sum_{m=1}^{2} \sum_{k_m=0}^{n} L^m_{k_m}(x_i)\phi(t^m_{k_m}) &= \sum_{k_1=0}^{n} L^1_{k_1}(x_i)\phi(t^1_{k_1}) + \sum_{k_2=0}^{n} L^2_{k_2}(x_i)\phi(t^2_{k_2}) && \text{(3.27a)}\\
&= \sum_{k_1=0}^{n} L^1_{k_1}(x_i)\phi(t^1_{k_1}) + \sum_{k_2=0}^{n} \left(\sum_{k_1=0}^{n} L^2_{k_2}(t^1_{k_1}) L^1_{k_1}(x_i) \right) \phi(t^2_{k_2}) && \text{(3.27b)}\\
&= \sum_{k_1=0}^{n} L^1_{k_1}(x_i)\phi(t^1_{k_1}) + \sum_{k_1=0}^{n} L^1_{k_1}(x_i) \left(\sum_{k_2=0}^{n} L^2_{k_2}(t^1_{k_1})\phi(t^2_{k_2}) \right) && \text{(3.27c)}\\
&= \sum_{k_1=0}^{n} L^1_{k_1}(x_i) \left(\phi(t^1_{k_1}) + \sum_{k_2=0}^{n} L^2_{k_2}(t^1_{k_1})\phi(t^2_{k_2}) \right). && \text{(3.27d)}
\end{aligned}$$

Equations 3.27a, 3.27c and 3.27d are straightforward definitions and algebra, while Eq. 3.27b relies on the interpolation relation in Eq. 3.23. Then the alternative version of Eq. 3.26 is

$$\phi(x_i) \mathrel{+}= \sum_{k_1=0}^{n} L^1_{k_1}(x_i) \left(\phi(t^1_{k_1}) + \sum_{k_2=0}^{n} L^2_{k_2}(t^1_{k_1})\phi(t^2_{k_2}) \right), \tag{3.28}$$

which corresponds to Eq. 3.20, where the terms in parentheses correspond to Eq. 3.19; the second term is the parent-to-child interpolation and the first term is aggregation with proxy potentials in the leaves previously computed in the DTT. In summary, instead of interpolating from $t^1_{k_1}$ to x_i *and* from $t^2_{k_2}$ to x_i (Eq. 3.27a), one interpolates from $t^2_{k_2}$ to $t^1_{k_1}$, aggregates with previously computed results at $t^1_{k_1}$, and finally interpolates from $t^1_{k_1}$ to x_i (Eq. 3.27d). This procedure generalizes to accommodate trees of any depth.

3.2.8 Description of BLTC

We briefly describe our previous barycentric Lagrange treecode (BLTC) [66, 6] which has an algorithmic structure resembling the Barnes-Hut treecode [1]. Unlike the BLDTT which builds a tree on both the source and target particles, the BLTC builds a tree of clusters on the source particles and a set of batches on the target particles, where the batches correspond to the leaves of a target tree. Once the source tree is built, the BLTC computes the proxy charges for each source cluster directly from the source particles by Eq. 3.8. The source tree is then traversed for each target batch, starting at the root of the tree and checking whether a given source cluster and target batch are well-separated. If they are well-separated and the source cluster contains more particles than interpolation points, then the cluster and batch interact by the PC approximation in Eq. 3.7. If they are not well-separated, then the batch interacts with the children of the source cluster. Leaves in the source tree that are not well-separated from a given target batch, and source clusters that are well-separated but contain more interpolation points than particles interact directly with the target batch by the PP interaction in Eq. 3.5. For M target particles and N source particles, the BLTC operation count is $O(N \log N) + O(M \log N)$, where the first term arises from the computation of the proxy charges and the second term arises from the source tree traversal. There is no downward pass in the BLTC.

3.3 BLDTT implementation

The implementation of the BLDTT for multiple GPUs is largely similar to that of our previous BLTC implementation [6], and is available as part of the BaryTree library for fast summation of particle interactions available on GitHub at `github.com/Treecodes/BaryTree`. The code uses OpenACC directives for GPU acceleration and MPI remote memory access for distributed memory parallelization. Tree building, tree traversal, and MPI communication of particles and clusters occur on the CPU, while the upward pass, particle and cluster interaction computations, and downward pass occur on the GPU. We review here several important details of this implementation.

3.3.1 Computing interaction lists

We decouple dual tree traversal from the computation of the particle interactions. The dual tree traversal is performed on the CPU, creating four interaction lists for each cluster of the target tree, one for each type of interaction. Each list contains the indices of the source clusters that interact with the target cluster by the given interaction type. The interactions are then computed by directly iterating over the interaction lists; this improves the efficiency

of GPU calculations because these lists can be iterated over rapidly and GPU compute kernels (described below) can be queued asynchronously.

3.3.2 MPI distributed memory parallelization

To implement distributed memory parallelization, we use locally essential trees (LET) [86]. Particles are partitioned by recursive coordinate bisection to create compact sub-domains on each MPI rank using the Zoltan library of Trilinos [87, 88], a software package developed at Sandia National Laboratory for load balancing and domain partitioning. Each MPI rank constructs the local source tree and local target tree for its particles. The LET of a rank is the union of the rank's local source tree and all source clusters from remote ranks interacting with its local target tree. Even though constructing the LETs requires an all-to-all communication, the amount of data acquired by each rank grows only logarithmically with the problem size [86]. The construction and communication required by the LETs are performed using MPI passive target synchronization remote memory access (RMA). RMA is a one-sided communication model within MPI in which an origin process can *put* data onto a target process or *get* data from a target process through specially declared memory *windows*, with no active involvement from the target process. This approach enables each rank to construct its LET asynchronously from other ranks.

3.3.3 GPU details

The GPU implementation uses eight compute kernels, two for the upward pass, four for computing interactions determined by the dual tree traversal, and two for the downward pass. The compute kernels are generated with OpenACC directives, compiled with the PGI C compiler. We utilize asynchronous launch of kernels in multiple GPU streams to hide as much latency as possible. The approach described here generalizes to multiple GPUs in a straightforward manner, in which each GPU corresponds to one MPI rank.

The first upward pass kernel computes the proxy charges for a given leaf in the source tree. For each leaf, this kernel is launched asynchronously, and any further computation is blocked until all leaf proxy charges are computed. The second upward pass kernel computes the proxy charges of a parent cluster using the proxy charges of its children. For a given level of the tree, this kernel is launched asynchronously for each cluster at that level, and any further computation is blocked until all proxy charges at that level are computed. The proxy charges at a given level are computed after the computation of the proxy charges at the previous level is complete.

The four DTT kernels compute the interaction of a target cluster with a source cluster. Each PP, PC, CP, and CC interaction launches one compute kernel. All such kernels are launched asynchronously and further computation is blocked until these kernels complete. The four interaction kernels have a similar structure in which the outer loop is over the target particles or proxy particles in the target cluster, and the inner loop is over the source particles or proxy particles in the source cluster. Importantly, due to the Lagrange form of barycentric interpolation, the inner loop iterations are independent, unlike alternative approximation methods which are sequential. The outer loop is mapped to the `gang` construct in OpenACC and the inner loop is mapped to the `vector` construct. Conceptually, a member of a `gang` corresponds to a thread block, while a member of a `vector` corresponds to an individual thread.

The two downward pass compute kernels are similar in structure to the upward pass kernels. At each level of the target tree above the leaves, beginning with the root, the first downward pass compute kernel is launched asynchronously for each target cluster at that level to interpolate the proxy potentials of the cluster to its children. Further computation is blocked until all compute kernels at a given level complete. Finally, the second downward pass compute kernel is launched asynchronously for each target leaf to interpolate the proxy potentials to the target particles.

3.4 Methodology

We demonstrate the BLDTT on a series of test cases and compare its performance to that of the BLTC. First, we compare the scaling of the BLTC and BLDTT on problem sizes from 1E5 to 1E8 particles. Second, we briefly demonstrate the GPU acceleration of the BLDTT over a CPU implementation. Third, we display the performance of the BLDTT on various particle distributions: random uniform, Gaussian, and Plummer [89, 90]. We also demonstrate the benefit of including CP and PC interactions in the BLDTT algorithm. Fourth, we display the performance of the BLDTT on various geometries: a $1 \times 10 \times 10$ slab, a $1 \times 1 \times 10$ slab, and a spherical shell with all particles at radius 1. Fifth, we investigate BLDTT performance in cases where the number of sources and targets are unequal. Sixth, we show performance results for various interaction kernels, including an oscillatory kernel, the Yukawa kernel, and the regularized Coulomb kernel. Last, we show MPI strong scaling performance on 1 to 32 GPUs.

Except for the Plummer distribution runs, the source particle charges are random uniformly distributed on $[-1, 1]$. Except for the runs involving unequal numbers of sources and targets, the source and target particle sets are identical. All computations use the Coulomb interaction

kernel except for the examples in §3.5.6. All runs used a maximum leaf or batch size of 2000. To facilitate comparison of the BLDTT and BLTC across this wide variety of problems, Figs. 3.7, 3.8, 3.11 and 3.13 all use the same axes to display time versus error.

The calculations are done in double precision arithmetic and the reported errors are the relative ℓ_2 error,

$$E = \left(\sum_{i=1}^{M} (\phi_i^{ds} - \phi_i^{fs})^2 \Big/ \sum_{i=1}^{M} (\phi_i^{ds})^2 \right)^{1/2}, \tag{3.29}$$

where ϕ_i^{ds} are the target potentials computed by direct summation and ϕ_i^{fs} are computed by the BLDTT fast summation method. The error was sampled at a random subset of 0.1% of the target particles in all cases.

The computations were performed on the NVIDIA P100 GPU nodes at the San Diego Supercomputer Center Comet machine, where each node contains four GPUs, and each GPU has 16GB of memory. These resources were provided through the through the Extreme Science and Engineering Discovery Environment (XSEDE) [91]. Except for the MPI strong scaling results in §3.5.7, each computation was run on a single GPU. The code was compiled with the PGI C compiler using the `-O3` optimization flag. For parallel scaling results, the Zoltan library of Trilinos [87, 88] was used to perform recursive coordinate bisection to load balance the particles. As described in the previous section, tree building and interaction list building are performed on the CPU, while the upward pass, particle interactions, and downward pass are performed on the GPU.

3.5 Results

3.5.1 Problem size scaling

Figure 3.5 shows the compute time (s) for direct summation (green), BLTC (red), and BLDTT (blue) with N=1E5, 1E6, 1E7, 1E8 random uniformly distributed source and target particles in the $[-1, 1]^3$ cube interacting by the Coulomb kernel. The BLTC and BLDTT use MAC parameter $\theta = 0.7$ and interpolation degree $n = 8$, yielding 7-8 digit accuracy. Figure 3.5(a) is a linear plot, showing that the BLDTT is about twice as fast as the BLTC, and both are much faster than direct summation. Figure 3.5(b) is a logarithmic plot with reference lines scaling as $O(N)$ (dashed), $O(N \log N)$ (dotted), and $O(N^2)$ (dash-dotted), showing that as the problem size increases, the BLTC has asymptotic $O(N \log N)$ scaling, while the BLDTT has asymptotic $O(N)$ scaling, as expected. Table 3.1 records the compute time and error; the asymptotic scaling of the compute time can be quantitatively confirmed,

and while there is a slight increase in the error with problem size, the BLDTT error is consistently smaller than the BLTC error.

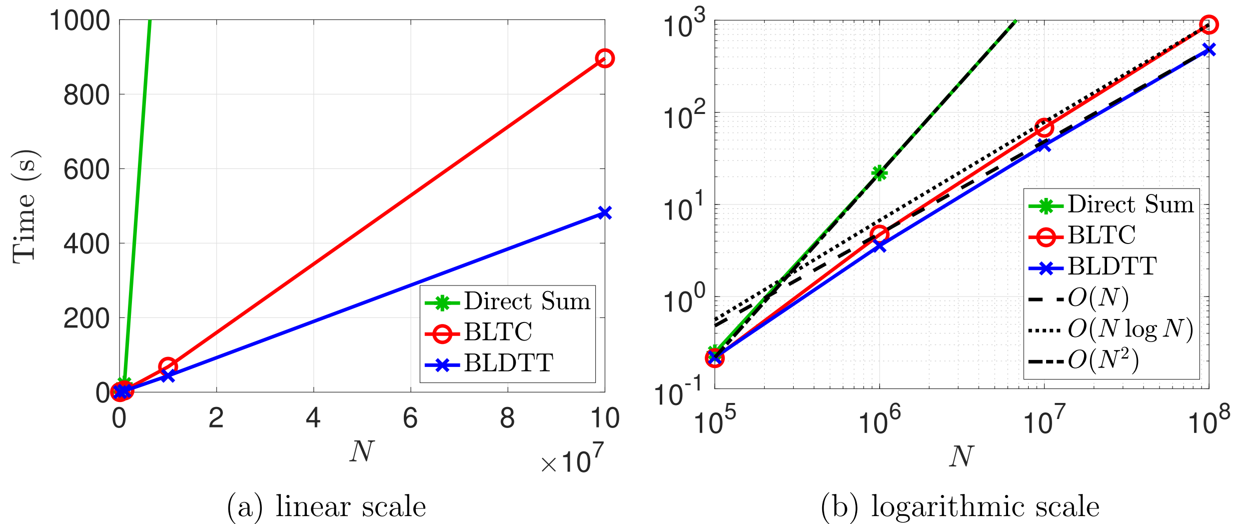


Figure 3.5: Comparison of BLTC and BLDTT, compute time (s) versus number of particles N=1E5, 1E6, 1E7, 1E8, random uniformly distributed particles in $[-1,1]^3$ interacting by the Coulomb kernel, MAC parameter $\theta = 0.7$, degree $n = 8$ yielding 7-8 digit accuracy, direct sum (green), BLTC (red), BLDTT (blue), (a) linear scale, (b) logarithmic scale, scaling lines $O(N^2)$ (dash-dotted), $O(N \log N)$ (dotted), $O(N)$ (dashed), simulations ran on one NVIDIA P100 GPU.

N	BLTC time (s)	BLTC error	BLDTT time (s)	BLDTT error
1E5	2.15E−1	1.75E−8	2.19E−1	1.58E−8
1E6	4.71E+0	1.42E−7	3.56E+0	3.67E−8
1E7	6.81E+1	4.68E−7	4.40E+1	4.12E−8
1E8	8.96E+2	9.23E−7	4.82E+2	4.17E−8

Table 3.1: Comparison of BLTC and BLDTT, number of particles N = 1E5, 1E6, 1E7, 1E8, random uniformly distributed particles in $[-1,1]^3$ interacting by the Coulomb kernel, MAC parameter $\theta = 0.7$, degree $n = 8$, compute time (s) from Fig. 3.5, ℓ_2 error, simulations ran on one NVIDIA P100 GPU.

3.5.2 GPU acceleration of BLDTT

In this subsection we compare to the BLDTT running on a single NVIDIA P100 GPU to running on 8 CPU cores of an Intel Xeon E5-2680v3 processor at 2.50 GHz with MPI parallelization. We perform the same four calculations above, so the errors are the same as those in Table 3.1. Table 3.2 gives the compute times, showing that the BLDTT achieves

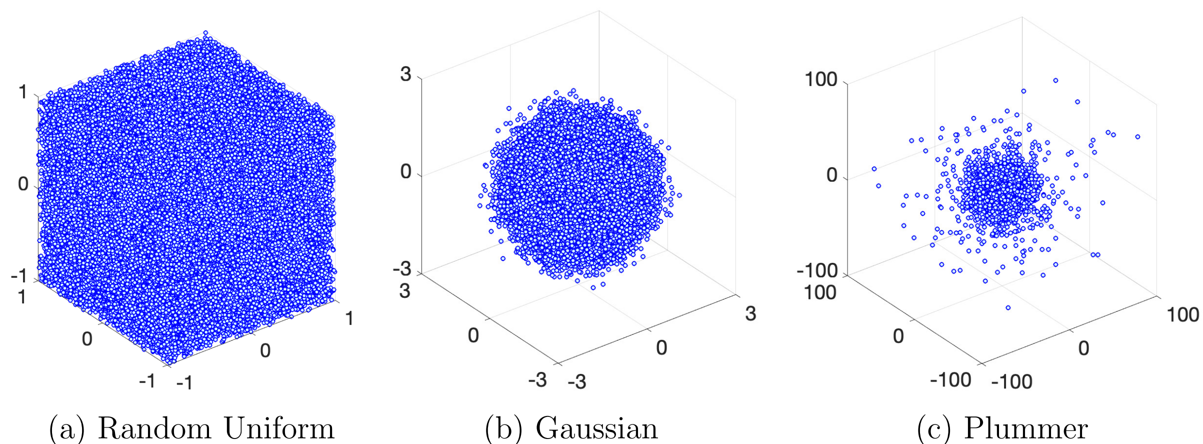
30-40× speedup on the GPU compared the 8 CPU cores. The efficiency of the BLDTT running on the GPU is due to the independent nature of the terms in the barycentric Lagrange approximation, which allows them to be computed concurrently.

N	CPU time (s)	GPU time (s)	speedup
1E5	7.84E+0	2.19E−1	35.8
1E6	1.45E+2	3.56E+0	40.7
1E7	1.40E+3	4.40E+1	31.8
1E8	1.70E+4	4.82E+2	35.3

Table 3.2: Comparison of BLDTT running on 8 CPU cores and on one NVIDIA P100 GPU, number of particles N = 1E5, 1E6, 1E7, 1E8, random uniformly distributed particles in $[-1,1]^3$ interacting by the Coulomb kernel, MAC parameter $\theta = 0.7$, degree $n = 8$, compute time (s), speedup, same errors as in Table 3.1.

3.5.3 Non-uniform particle distributions

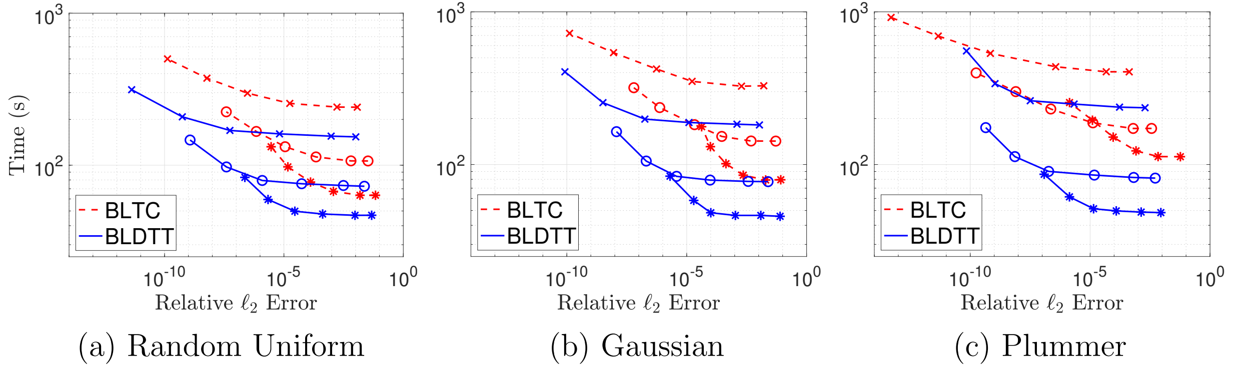
We investigate the performance of the BLDTT for three different random particle distributions: (a) uniform particles in $[-1,1]^3$, (b) Gaussian particles with radial pdf $\frac{1}{\sqrt{6\pi}} \exp(-r^2/6)$, (c) Plummer particles [89, 90] with radial pdf $\frac{3}{4\pi}(1+r^2)^{-5/2}$ and cutoff at ±100 in all three Cartesian coordinates. The charges of the uniform and Gaussian particles are uniformly distributed in $[-1,1]$, while the Plummer particle charges are set to $1/N$, where N is the total number of particles. To give a sense of the structure of the distributions, Fig. 3.6 depicts the three distributions with N=4E5 particles. Compared to the uniform case (a), the Gaussian and Plummer distributions (b,c) are more highly concentrated near the origin, with the Gaussian decaying more rapidly away from the origin and the Plummer decaying less rapidly.



(a) Random Uniform (b) Gaussian (c) Plummer

Figure 3.6: Sample random distributions with N=4E5 particles, (a) uniform, (b) Gaussian, (c) Plummer.

Figure 3.7 shows the compute time (s) versus relative ℓ_2 error for the BLDTT (blue, solid) and BLTC (red, dashed) on these three distributions with N=2E7 particles. Each connected curve represents constant MAC with $\theta = 0.5$ ($\times$), $\theta = 0.7$ ($\circ$), $\theta = 0.9$ ($*$), and the interpolation degree $n = 1, 2, 4, 6, 8, 10$ increases from right to left on each curve. For these parameter choices the errors span the range from 1 digit to 10 digit accuracy. Large θ is more efficient for low accuracy and small θ is more efficient for high accuracy. The results show that the BLDTT has consistently better performance than the BLTC and is less sensitive to the particle distribution.



(a) Random Uniform (b) Gaussian (c) Plummer

Figure 3.7: Different particle distributions, compute time (s) versus relative ℓ_2 error, N=2E7 random particles, (a) uniform, (b) Gaussian, (c) Plummer, BLTC (red, dashed), BLDTT (blue, solid), connected curves represent constant MAC θ (0.5 $\times$; 0.7 $\circ$; 0.9 $*$), interpolation degree $n = 1, 2, 4, 6, 8, 10$ increases from right to left on each curve, simulations ran on one NVIDIA P100 GPU.

To demonstrate the effect of including PC and CP interactions in the BLDTT, Fig. 3.8 shows the compute time versus relative ℓ_2 error for the BLDTT (blue, solid) as presented in this paper using PP, PC, CP and CC interactions, and a version of the BLDTT (red, dashed) using only CC and PP interactions. When only CC and PP interactions are used, the interaction between a target cluster and a source cluster is handled by PP interaction if either cluster contains more interpolation points than particles, whereas the flexibility to choose PC or CP interactions in those cases improves performance at higher interpolation degree, especially for the non-uniform particle distributions.

To further understand the effect of including PC and CP interactions, next we compare the number of pointwise interactions used by the two versions of the BLDTT, where by pointwise interaction we mean one evaluation of the kernel $G(\mathbf{x}, \mathbf{y})$. Results are shown for MAC $\theta = 0.9$ and interpolation degree $n = 1, 2, 4, 6, 8, 10$, for the same three random distributions with N=2E7 particles as above. Figure 3.9 displays results for the four types of interactions in stacked bars, CC (blue), PP (orange), PC (yellow), CP (purple), from bottom

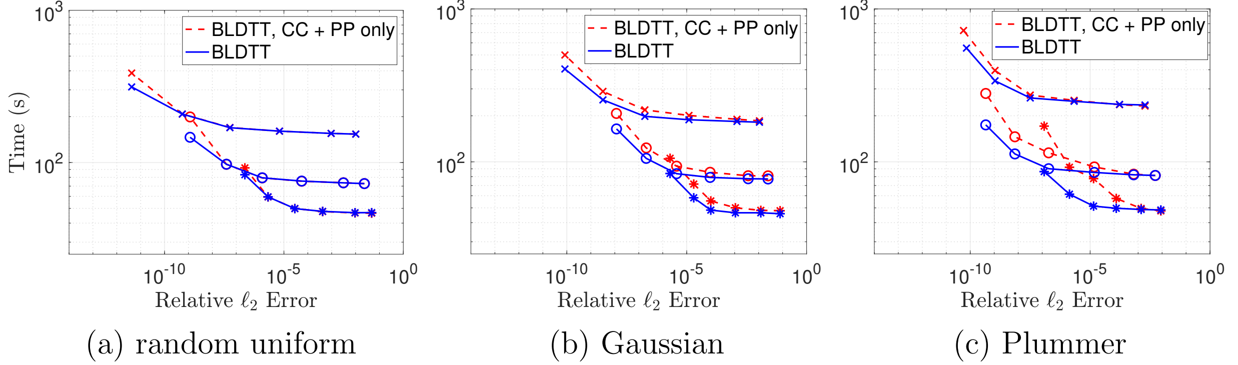


Figure 3.8: Different particle distributions, compute time (s) versus relative ℓ_2 N=2E7 random particles, (a) uniform, (b) Gaussian, (c) Plummer, BLDTT with only CC and PP interactions (red, dashed), BLDTT with PP, PC, CP, and CC interactions (blue, solid), connected curves represent constant MAC θ (0.5 $\times$; 0.7 $\circ$; 0.9 $*$), interpolation degree $n = 1, 2, 4, 6, 8, 10$ increases from right to left on each curve, simulations ran on one NVIDIA P100 GPU.

to top, where the left bar in each pair is the BLDTT with CC and PP interactions only, and the right bar is the BLDTT with PP, PC, CP, and CC interactions. In this case a direct sum calculation would use 4E14 PP interactions, while the BLDTT calculations use less than 6E12 interactions. The results show that for high degree, introducing PC and CP interactions into the BLDTT significantly reduces the number of PP interactions, replacing them with a much smaller number of PC and CP interactions, and this effect is more prominent for the nonuniform particle distributions.

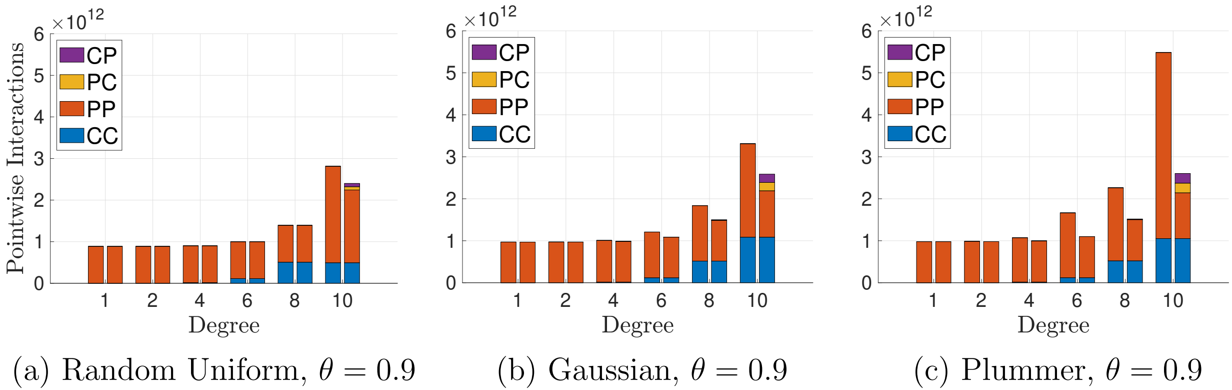
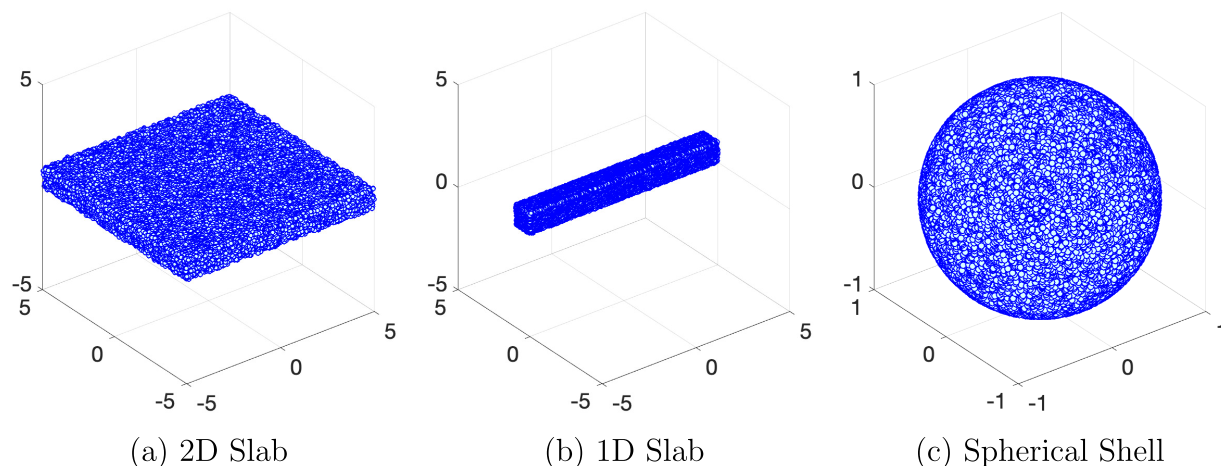


Figure 3.9: Different particle distributions, number of pointwise interactions (kernel evaluations $G(\mathbf{x}, \mathbf{y})$), N=2E7 random particles, (a) uniform, (b) Gaussian, (c) Plummer, MAC $\theta = 0.9$, interpolation degree $n = 1, 2, 4, 6, 8, 10$, each pair of bars shows BLDTT with CC and PP only (left) and BLDTT with PP, PC, CP, CC (right), direct sum calculation would use 4E14 PP interactions, simulations ran on one NVIDIA P100 GPU.

3.5.4 Non-cubic particle domains

We demonstrate here the performance of the BLDTT on three examples with non-cubic particle domains depicted in Fig. 3.10: (a) thin slab of dimensions $1 \times 10 \times 10$, (b) square rod of dimensions $1 \times 1 \times 10$, and (c) spherical surface of radius 1. In all cases the particles are random uniformly distributed.



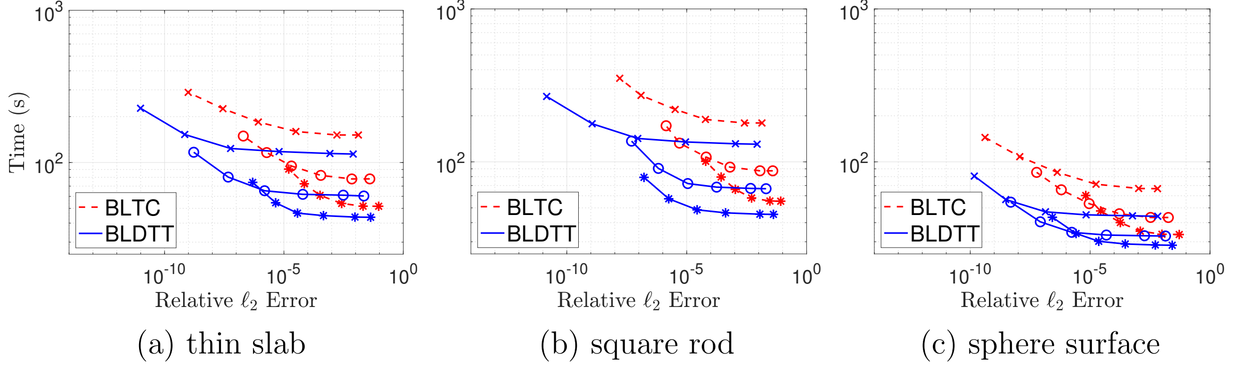
(a) 2D Slab (b) 1D Slab (c) Spherical Shell

Figure 3.10: Non-cubic domains, N=4E5 random uniformly distributed particles, (a) thin slab of dimensions $1 \times 10 \times 10$, (b) square rod of dimensions $1 \times 1 \times 10$, (c) spherical surface of radius 1.

Figure 3.11 shows the compute time (s) versus the relative ℓ_2 error for the BLDTT (blue, solid) and BLTC (red, dashed) on these three examples with N=2E7 particles, using MAC $\theta = 0.5, 0.7, 0.9$ and interpolation degree $n = 1, 2, 4, 6, 8, 10$. The results show that the BLDTT has consistently better performance than the BLTC. Compared to the cubic domain results in Fig. 3.7(a), the BLDTT achieves similar levels of error and runs somewhat faster for the non-cubic domains. Heuristically, the BLDTT run time depends on the complexity of the tree; in particular, the tree is an oct-tree for the cubic domain, close to a quad-tree for the thin slab and sphere surface, and close to a binary tree for the square rod. The results indicate that BLDTT automatically adapts to the complexity of the tree without requiring explicit reprogramming.

3.5.5 Unequal targets and sources

To demonstrate the performance of the BLDTT with unequal target and source particles, we consider a cluster-particle variant of the BLTC for comparison [92, 37]. The CP-BLTC builds a tree on the M targets and a set of batches on the N sources, and rather than using PP and PC interactions, it uses PP and CP interactions. Instead of an $O(N \log N)$



(a) thin slab (b) square rod (c) sphere surface

Figure 3.11: Non-cubic domains, N=2E7 random uniformly distributed particles, (a) thin slab of dimensions $1 \times 10 \times 10$, (b) square rod of dimensions $1 \times 1 \times 10$, (c) sphere surface of radius 1, compute time (s) versus relative ℓ_2 error, BLTC (red, dashed), BLDTT (blue, solid), connected curves represent constant MAC θ (0.5 $\times$; 0.7 $\circ$; 0.9 $*$), interpolation degree $n = 1, 2, 4, 6, 8, 10$ increases from right to left on each curve, simulations ran on one NVIDIA P100 GPU.

upward pass to compute proxy charges, the CP-BLTC has an $O(M \log M)$ downward pass to interpolate proxy potentials to targets. While the compute phase of the BLTC is $O(M \log N)$, the compute phase of the CP-BLTC is $O(N \log M)$. Since the compute phase is in general the most expensive part of the algorithm, we expect the BLTC to perform better than the CP-BLTC when $N > M$, and vice versa.

Figure 3.12 shows the compute time (s) versus relative ℓ_2 error for the BLTC (red, dashed), CP-BLTC (green, dash-dotted), and BLDTT (blue, solid) with (a) M=2E7 targets, N=2E6 sources, (b) M=2E6 targets, N=2E7 sources, for MAC θ and interpolation degree n as above. For (a) $M > N$, the CP-BLTC outperforms the BLTC, for (b) $N > M$, the BLTC outperforms the CP-BLTC for errors below 1E–5, while the BLDTT outperforms the two treecodes in both cases. Note however in (b) that for MAC $\theta = 0.9$ and error larger than 1E–3, the CP-BLTC runs slightly faster than the BLDTT. This is due to the cost of the upward pass in the BLDTT, which with low degree n, makes up a substantial portion of the compute time. We note that the upward pass is more expensive than the downward pass because in the current implementation, the upward pass parallelizes less well than the downward pass on the GPU. The results demonstrate the ability of the BLDTT to efficiently adapt to the case of unequal targets and sources.

3.5.6 Other interaction kernels

In previous sections we considered particles interacting through the Coulomb potential. Here we demonstrate the performance of the BLDTT on three other interaction kernels: (a)

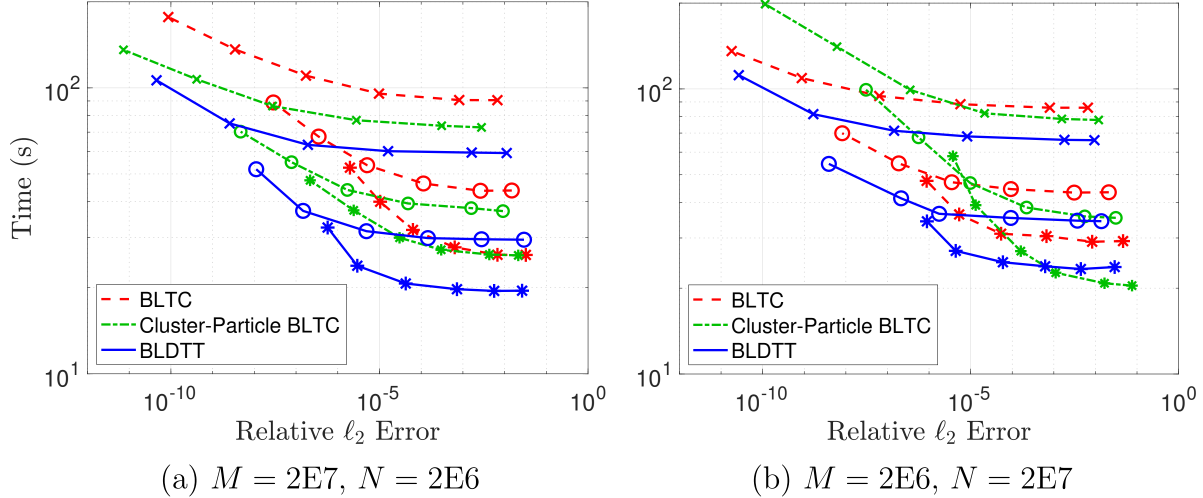


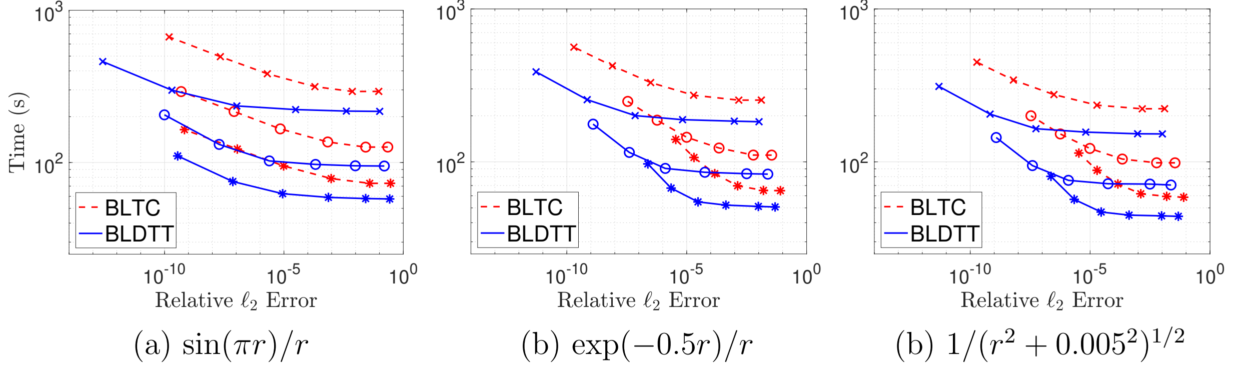
Figure 3.12: Unequal targets and sources, (a) M=2E7 targets, N=2E6 sources, (b) M=2E6 targets, N=2E7 sources, random uniformly distributed particles, compute time (s) versus relative ℓ_2 error, BLTC (red, dashed), CP-BLTC (green, dash-dotted), BLDTT (blue, solid), connected curves represent constant MAC θ (0.5 $\times$; 0.7 $\circ$; 0.9 $*$), interpolation degree $n = 1, 2, 4, 6, 8, 10$, simulations ran on one NVIDIA P100 GPU.

an oscillatory kernel, $\sin(\pi r)/r$, (b) a Yukawa kernel, $\exp(-0.5r)/r$, and (c) a regularized Coulomb kernel, $1/(r^2+\epsilon^2)^{1/2}$, with $\epsilon = 0.005$. Figure 3.13 shows the compute time (s) versus relative ℓ_2 error for the BLDTT (blue, solid) and BLTC (red, dashed) for these three kernels on N=2E7 random uniformly distributed particles in the cube $[-1, 1]^3$ for various values of the MAC θ and interpolation degree n. The results show that the BLDTT has consistently better performance than the BLTC. Comparing with the Coulomb potential resuts in Fig. 3.7(a), we see that the BLDTT has similar performance for the various interaction kernels, reflecting the kernel-independent nature of the method.

3.5.7 MPI strong scaling

Finally, we demonstrate the MPI strong scaling of the BLDTT up to 32 NVIDIA P100 GPUs with one MPI rank per GPU. The particles are partitioned into geometrically localized domains by Trilinos Zoltan [87, 88]. Figure 3.14 depicts a sample domain decomposition for N=1.6E6 random uniformly distributed particles in the cube $[-1, 1]^3$, (a) across 8 ranks with 2E5 particles per rank, and (b) across 16 ranks with 1E5 particles per rank. Colors represent particles residing on different ranks.

We consider a problem with N=64E6 particles using MAC $\theta = 0.7$ and interpolation degree $n = 8$ yielding error $\approx$ 1E–8. Figure 3.15 shows the compute time versus the number of GPUs for (a) random uniform, (b) Gaussian, and (c) Plummer distributions for the



(a) $\sin(\pi r)/r$ (b) $\exp(-0.5r)/r$ (b) $1/(r^2+0.005^2)^{1/2}$

Figure 3.13: Different interaction kernels, N=2E7 random uniformly distributed particles in the cube $[-1,1]^3$, (a) oscillatory, $\sin(\pi r)/r$, (b) Yukawa, $\exp(-0.5r)/r$, (c) regularized Coulomb, $1/(r^2+0.005^2)^{1/2}$, compute time (s) versus relative ℓ_2 error, BLTC (red, dashed), BLDTT (blue, solid), connected curves represent constant MAC θ (0.5 $\times$; 0.7 $\circ$; 0.9 $*$), interpolation degree $n = 1, 2, 4, 6, 8, 10$ increases from right to left on each curve, simulations ran on one NVIDIA P100 GPU.

BLDTT (blue) and BLTC (red), where dashed lines indicate ideal scaling and the boxed numbers show the parallel efficiency. As was shown earlier for one GPU in Fig. 3.7, the BLDTT is consistently faster than the BLTC up to 32 GPUs, and the speedup improves for the nonuniform particle distributions. The BLDTT and BLTC have generally similar parallel efficiency for all three distributions; for example on 32 GPUs, the BLDTT has parallel efficiency 77%, 65%, and 66% for the uniform, Gaussian, and Plummer distributions, compared to 83%, 81%, and 64% for the BLTC.

The slightly lower parallel efficiency for the BLDTT compared to the BLTC can be explained by the greater efficiency of the BLDTT algorithm itself. Figure 3.16 shows the component breakdown as a percentage of run time (total wall clock time) of the (a) BLTC and (b) BLDTT for the uniform distribution results in Fig. 3.15(a). The components shown are the upward pass (blue), compute due to local sources and source clusters (orange), compute due to remote sources and source clusters (yellow), downward pass (purple), LET construction and communication (green), and other (light blue), which includes tree building and interaction list building. The breakdown is based on timing results for the most expensive MPI rank in each case. Note that the LET construction accounts for an increasing percentage of the run time as the number of ranks increases. This is to be expected as more particles reside on remote ranks and must be communicated, and this is the primary factor that impedes ideal parallel scaling. The LET construction time is nearly identical for the BLTC and BLDTT, however since the BLDTT computations are more efficient, the LET construction accounts for a larger percentage of the run time and this results in the lower parallel efficiency seen in Fig. 3.15 as the number of ranks increases.

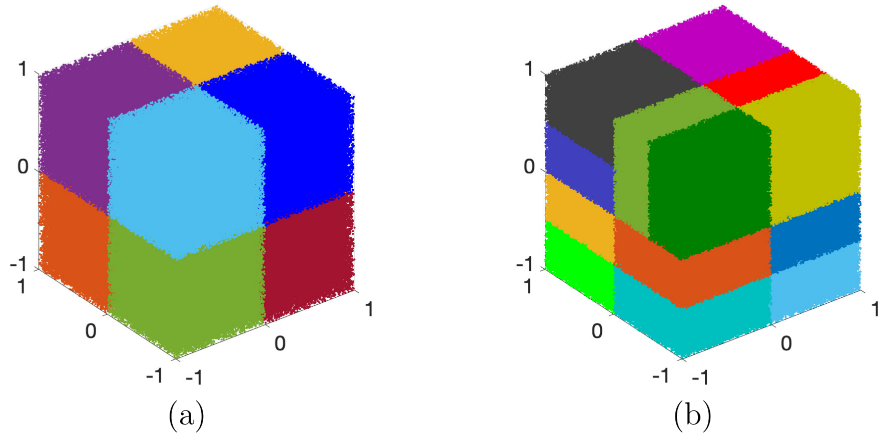


Figure 3.14: Examples of domain decomposition, N=1.6E6 random uniformly distributed particles in the cube $[-1,1]^3$, (a) 8 ranks with 2E5 particles per rank, (b) 16 ranks with 1E5 particles per rank, colors represent particles residing on different ranks, partitioning by Trilinos Zoltan.

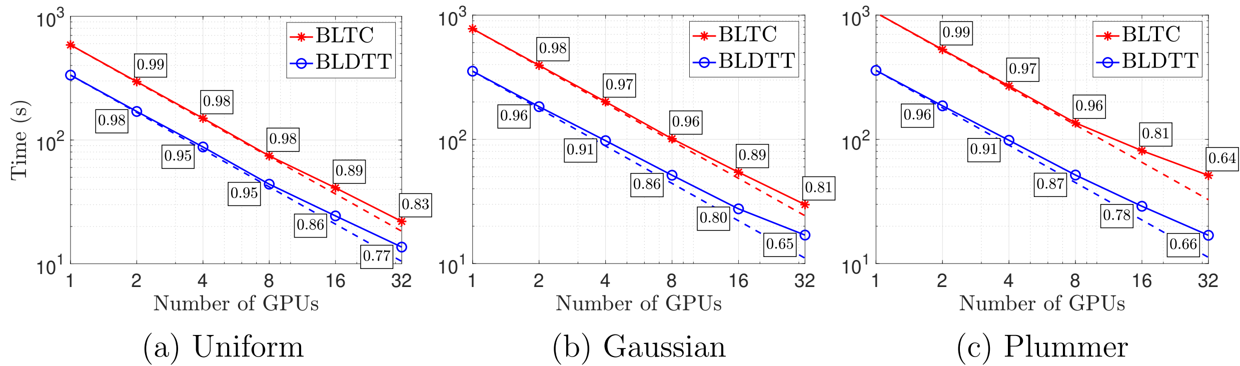
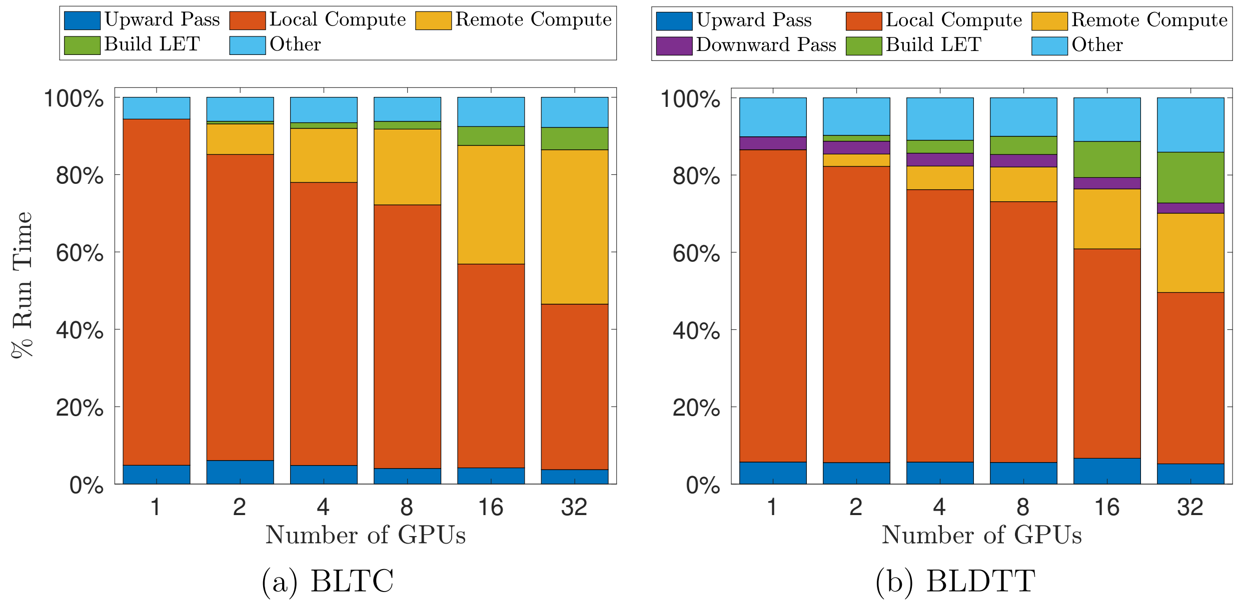


Figure 3.15: MPI strong scaling, N=64E6 particles, (a) random uniform, (b) Gaussian, (c) Plummer distributions, MAC $\theta = 0.7$, interpolation degree $n = 8$ yielding 7-8 digit accuracy, compute time (s) versus number of GPUs, BLTC (red), BLDTT (blue), ideal scaling (dashed lines), parallel efficiency (boxed numbers).



(a) BLTC

(b) BLDTT

Figure 3.16: Component breakdown of run time across 1 to 32 NVIDIA P100 GPUs, 64E6 random uniformly distributed particles in the cube $[-1,1]^3$, MAC $\theta = 0.7$, interpolation degree $n = 8$, error $\approx$ 1E–8, (a) BLTC, (b) BLDTT, upward pass (blue), compute due to local sources and source clusters (orange), compute due to remote sources and source clusters (yellow), downward pass (purple), LET construction and communication (green), and other (light blue), which includes tree building and interaction list building, breakdown is based on timing results for most expensive MPI rank in each case.

CHAPTER 4

A Poisson–Boltzmann Equation Solver

This chapter addresses work related to the treecode-accelerated boundary integral Poisson–Boltzmann solver (TABI-PB), a software package for modeling biomolecular solvation. §4.1 gives an overview of the Poisson–Boltzmann model and previous work on the TABI-PB solver. §4.2 investigates the impact of surface triangulation codes on the performance of TABI-PB. §4.3 implements a node patch method for discretizing the boundary integral problem, as opposed to the previous centroid collocation scheme. §4.4 describes the development of a new BLDTT-based GPU-accelerated TABI-PB solver, applying the work detailed in Chapter 3 to TABI-PB. §4.5 describes the application of TABI-PB to the calculation of electrostatic free energies of binding between proteins and ligands, comparing with a popular finite-difference based Poisson–Boltzmann software. In addition to these projects, work performed over the course of this thesis integrated TABI-PB into the popular Adaptive Poisson–Boltzmann Solver (APBS) developed at Pacific Northwest National Laboratory. The current version of TABI-PB is available at `github.com/Treecodes/TABI-PB` and as a submodule of the APBS package developed at Pacific Northwest National Laboratory [29] at `github.com/Electrostatics/APBS`. The content of this chapter largely follows the work of [93], which is currently in revision and submitted to J. Comput. Chem., and several papers in preparation, including [94].

4.1 Background

4.1.1 The Poisson–Boltzmann equation

Implicit solvent models play an important role in computational modeling of electrostatic interactions between biomolecules and their solvent environment [8, 9, 10]. Of particular importance is the Poisson–Boltzmann (PB) model [11, 12]. Consider an interior domain $\Omega_1 \subset \mathbb{R}^3$ containing the solute biomolecule, and an exterior domain $\Omega_2 = \mathbb{R}^3 \setminus \overline{\Omega}_1$ containing the ionic solvent. In a 1:1 electrolyte at low ionic concentrations, one can utilize the linearized

PB equation for the electrostatic potential ϕ,

$$-\nabla \cdot (\varepsilon(\mathbf{x})\nabla\phi(\mathbf{x})) + \overline{\kappa}^2(\mathbf{x})\phi(\mathbf{x}) = \sum_{k=1}^{N_c} q_k \delta\left(\mathbf{x} - \mathbf{y}_k\right), \tag{4.1}$$

where ε is the dielectric constant, $\overline{\kappa}$ is the modified Debye-Hückel parameter in units of Å^{-2}, N_c is the number of atoms in the solute biomolecule, $\mathbf{y}_k$ is the position of the kth atom of the solute, and q_k is the associated partial charge in units of fundamental charge e_c. The interface conditions are

$$\phi_1(\mathbf{x}) = \phi_2(\mathbf{x}), \quad \varepsilon_1 \frac{\partial\phi_1(\mathbf{x})}{\partial n} = \varepsilon_2 \frac{\partial\phi_2(\mathbf{x})}{\partial n}, \quad \mathbf{x} \in \Gamma, \tag{4.2}$$

where $\phi_1(\mathbf{x})$ and $\phi_2(\mathbf{x})$ are the limiting values approaching the interface $\Gamma = \overline{\Omega}_1 \cap \overline{\Omega}_2$ from inside and outside the biomolecule, respectively, and n is the outward normal on the interface. The first condition expresses continuity of the electrostatic potential across the interface, and the second condition expresses continuity of the electric flux. The far-field boundary condition is

$$\lim_{|\mathbf{x}|\to\infty} \phi(\mathbf{x}) = 0. \tag{4.3}$$

The present work assumes that ε and $\overline{\kappa}$ are piecewise constant,

$$\varepsilon(\mathbf{x}) = \begin{cases} \varepsilon_1, & \mathbf{x} \in \Omega_1, \\ \varepsilon_2, & \mathbf{x} \in \Omega_2, \end{cases}, \quad \overline{\kappa}^2(\mathbf{x}) = \begin{cases} 0, & \mathbf{x} \in \Omega_1, \\ \left(\frac{8\pi N_A e_c^2}{1000 k_B T}\right) I_s, & \mathbf{x} \in \Omega_2, \end{cases} \tag{4.4}$$

where N_A is Avogadro's number, k_B is the Boltzmann constant, T is the temperature, and I_s is the molar concentration of the ionic solvent.

A key quantity of interest is the electrostatic solvation energy,

$$\Delta G_{\text{solv}} = \frac{1}{2} \sum_{k=1}^{N_c} q_k \phi_{\text{reac}}(\mathbf{y}_k), \tag{4.5}$$

where the *reaction field potential* at an atomic position,

$$\phi_{\text{reac}}(\mathbf{y}_k) = \lim_{\mathbf{x}\to\mathbf{y}_k} \left(\phi(\mathbf{x}) - \sum_{j=1}^{N_c} \frac{q_j}{4\pi|\mathbf{x} - \mathbf{y}_j|} \right), \tag{4.6}$$

is the difference between the total potential and the Coulomb potential due to the polarization of the medium in which the solute is embedded.

4.1.2 Treecode-accelerated boundary integral PB solver (TABI-PB)

A variety of numerical approaches have been applied to the Poisson–Boltzmann model, including finite-difference [13, 14, 15, 16, 17, 18, 19, 20], finite-element [12, 21, 22], and boundary integral [23, 24, 25, 26, 27] schemes. In particular, a treecode-accelerated boundary integral scheme for the linearized Poisson–Boltzmann equation (TABI-PB) was recently developed [26, 28]. Boundary integral schemes for the PB equation solve for the surface potential on a triangulated discretization of the interface. These schemes generally benefit from rigorous enforcement of the interface conditions and the boundary condition at infinity. On the other hand these schemes face the expense of solving a dense linear system, and hence TABI-PB has traditionally utilized a treecode algorithm to reduce the computational cost from $O(N^2)$ to $O(N \log N)$, where N is the number of triangles representing the interface.

Juffer et al. developed a coupled set of well-conditioned boundary integral equations for the surface potential ϕ_1 and its normal derivative on the solute/solvent interface [24],

$$\begin{aligned}
\frac{1}{2}\left(1+\varepsilon\right)\phi_1\left(\mathbf{x}\right) &= \int_\Gamma \left[K_1\left(\mathbf{x},\mathbf{y}\right)\frac{\partial \phi_1\left(\mathbf{y}\right)}{\partial n} + K_2\left(\mathbf{x},\mathbf{y}\right)\phi_1\left(\mathbf{y}\right)\right] dS_{\mathbf{y}} + S_1(\mathbf{x}),\ \mathbf{x}\in\Gamma,\\
\frac{1}{2}\left(1+\frac{1}{\varepsilon}\right)\frac{\partial \phi_1\left(\mathbf{x}\right)}{\partial n} &= \int_\Gamma \left[K_3\left(\mathbf{x},\mathbf{y}\right)\frac{\partial \phi_1\left(\mathbf{y}\right)}{\partial n} + K_4\left(\mathbf{x},\mathbf{y}\right)\phi_1\left(\mathbf{y}\right)\right] dS_{\mathbf{y}} + S_2(\mathbf{x}),\ \mathbf{x}\in\Gamma,
\end{aligned} \tag{4.7}$$

where $\varepsilon = \varepsilon_1/\varepsilon_2$ is the solute/solvent ratio of dielectric constants, and K_1, K_2, K_3, K_4 are kernels depending on the Coulomb and screened Coulomb potentials,

$$G_0\left(\mathbf{x},\mathbf{y}\right) = \frac{1}{4\pi\left|\mathbf{x}-\mathbf{y}\right|},\quad G_\kappa\left(\mathbf{x},\mathbf{y}\right) = \frac{e^{-\kappa\left|\mathbf{x}-\mathbf{y}\right|}}{4\pi\left|\mathbf{x}-\mathbf{y}\right|}. \tag{4.8}$$

$$\begin{aligned}
K_1\left(\mathbf{x},\mathbf{y}\right) = G_0\left(\mathbf{x},\mathbf{y}\right) - G_\kappa\left(\mathbf{x},\mathbf{y}\right),\quad & K_2\left(\mathbf{x},\mathbf{y}\right) = \varepsilon\frac{\partial G_\kappa\left(\mathbf{x},\mathbf{y}\right)}{\partial n_{\mathbf{y}}} - \frac{\partial G_0\left(\mathbf{x},\mathbf{y}\right)}{\partial n_{\mathbf{y}}}\\
K_3\left(\mathbf{x},\mathbf{y}\right) = \varepsilon\frac{\partial G_0\left(\mathbf{x},\mathbf{y}\right)}{\partial n_{\mathbf{x}}} - \frac{1}{\varepsilon}\frac{\partial G_\kappa\left(\mathbf{x},\mathbf{y}\right)}{\partial n_{\mathbf{x}}},\quad & K_4\left(\mathbf{x},\mathbf{y}\right) = \varepsilon\frac{\partial^2 G_\kappa\left(\mathbf{x},\mathbf{y}\right)}{\partial n_{\mathbf{x}}\partial n_{\mathbf{y}}} - \frac{1}{\varepsilon}\frac{\partial^2 G_0\left(\mathbf{x},\mathbf{y}\right)}{\partial n_{\mathbf{x}}\partial n_{\mathbf{x}}}
\end{aligned} \tag{4.9}$$

The normal derivatives of the potentials G are given by:

$$\begin{aligned}
\frac{\partial G\left(\mathbf{x},\mathbf{y}\right)}{\partial n_{\mathbf{y}}} &= \sum_{i=1}^{3} n_i\left(\mathbf{y}\right)\partial_{y_i} G\left(\mathbf{x},\mathbf{y}\right),\quad \frac{\partial G\left(\mathbf{x},\mathbf{y}\right)}{\partial n_{\mathbf{x}}} = -\sum_{i=1}^{3} n_i\left(\mathbf{x}\right)\partial_{x_i} G\left(\mathbf{x},\mathbf{y}\right),\\
\frac{\partial^2 G\left(\mathbf{x},\mathbf{y}\right)}{\partial n_{\mathbf{x}}\partial n_{\mathbf{y}}} &= -\sum_{j=1}^{3}\sum_{i=1}^{3} n_j\left(\mathbf{x}\right) n_i\left(\mathbf{y}\right)\partial_{x_j}\partial_{y_i} G\left(\mathbf{x},\mathbf{y}\right)
\end{aligned} \tag{4.10}$$

for the three spatial components n of the normal direction.

The source terms in Eq. 4.7 are

$$S_1(\mathbf{x}) = \frac{1}{\varepsilon_1}\sum_{k=1}^{N_c} q_k G_0\left(\mathbf{x},\mathbf{y}_k\right), \quad S_2(\mathbf{x}) = \frac{1}{\varepsilon_1}\sum_{k=1}^{N_c} q_k \frac{\partial G_0\left(\mathbf{x},\mathbf{y}_k\right)}{\partial n_{\mathbf{x}}}, \tag{4.11}$$

where N_c is the number of atoms in the biomolecule, and q_k is the charge of the kth atom. In the context of the integral equation formulation, the electrostatic solvation energy is given by

$$\begin{aligned} \Delta G_{\text{solv}} &= \frac{1}{2}\sum_{k=1}^{N_c} q_k \phi_{\text{reac}}(\mathbf{y}_k) \\ &= \frac{1}{2}\sum_{k=1}^{N_c} q_k \int_\Gamma \left[K_1(\mathbf{y}_k,\mathbf{y})\frac{\partial \phi_1(\mathbf{y})}{\partial n} + K_2(\mathbf{y}_k,\mathbf{y})\phi_1(\mathbf{y})\right] dS_{\mathbf{y}}. \end{aligned} \tag{4.12}$$

As mentioned above, the TABI-PB solver calculates the surface integrals using a boundary element method (BEM) on the triangulated SES. The discretized integrals are given by

$$\begin{aligned} \frac{1}{2}\left(1+\varepsilon\right)\phi_1\left(\mathbf{x}_i\right) &= \sum_{\substack{j=1\\ j\neq i}}^{N}\left[K_1\left(\mathbf{x}_i,\mathbf{x}_j\right)\frac{\partial \phi_1\left(\mathbf{x}_j\right)}{\partial n} + K_2\left(\mathbf{x}_i,\mathbf{x}_j\right)\phi_1\left(\mathbf{x}_j\right)\right] A_j + S_1(\mathbf{x}_i), \\ \frac{1}{2}\left(1+\frac{1}{\varepsilon}\right)\frac{\partial \phi_1\left(\mathbf{x}_i\right)}{\partial n} &= \sum_{\substack{j=1\\ j\neq i}}^{N}\left[K_3\left(\mathbf{x}_i,\mathbf{x}_j\right)\frac{\partial \phi_1\left(\mathbf{x}_j\right)}{\partial n} + K_4\left(\mathbf{x}_i,\mathbf{x}_j\right)\phi_1\left(\mathbf{x}_j\right)\right] A_j + S_2(\mathbf{x}_i), \end{aligned} \tag{4.13}$$

where $\mathbf{x}$ is the position and A_j is the area of the jth element, and the discretization contains N elements. In TABI-PB, this resulting linear system for the surface potentials and their normal derivatives is solved by GMRES iteration, which requires a matrix-vector product in each step of the iteration. A treecode algorithm can be employed to accelerate the matrix-vector product, reducing its cost from $O(N^2)$ to $O(N \log N)$ [26].

4.2 Project 1: Comparison of molecular surface triangulation codes

4.2.1 Project description

Since boundary integral schemes utilize a triangularization of the molecular surface, the computed potential and solvation energy in TABI-PB are highly dependent on the surface definition and the quality of the triangulation. The aim of this project is to investigate how the different triangulations affect the accuracy and efficiency of the subsequent computations.

There are several surface definitions commonly used, including the solvent excluded surface (SES), the Skin surface, and the Gaussian or "blobby" surface, as well as multiple

software implementations for triangulating these surfaces. The SES, or Connolly surface [95, 96], is formed by the inward facing surface of the probe rolled along the van der Waals (vdW) surface. The SES is comprised of contact surface portions, where the probe can touch the vdW surface, and reentrant surface portions, formed by the inward facing surface of the probe when it cannot touch the vdW surface, i.e., when it is in contact with more than one atom in the solute. The surface is composed of spherical and toroidal patches. Previous work investigated the performance of SES surfaces and Skin surfaces in Delphi, a finite-difference PB solver [97], and the performance of Gaussian surfaces relative to SES surfaces in AFMPB, a fast multipole PB solver [98].

The most widely used software for generating the SES surface is MSMS [99], while NanoShaper is a recently introduced alternative [97]. MSMS and NanoShaper are two publicly available codes for triangulating the SES. Each code provides a means for the user to to control the resolution of the triangulation, the codes use different algorithms and in general will produce different triangulations of a given SES.

MSMS is an SES triangulation software developed by Sanner [99]. After generating an analytical representation of the surface, the algorithm generates a triangulation of specified density by fitting predefined triangulated patches to the surface. MSMS has gained widespread popularity for generating surface meshes. The density of the mesh is controlled by a user-specified parameter d that sets the number of triangles in units of vertices/angstrom^2.

NanoShaper is a package for triangulating multiple surface definitions, developed by Decherchi and Rocchia [100]. For constructing the SES, NanoShaper first builds a description of the surface with a set of patches, analytically if possible or else with an approximation. The program then employs a ray-casting algorithm in which rays parallel to the coordinate axes are cast and intersections with the surface are calculated. The vertex positions of intersection are then used by the marching cubes algorithm to obtain the triangulation. The density of the mesh is controlled by a scaling parameter s that specifies the inverse side length of a cubic grid cell in units of angstroms.

4.2.2 Methodology

To assess the SES surface triangulations produced by MSMS and NanoShaper, we compute the surface area S_{a} and electrostatic solvation energy ΔG_{solv} for a test set of 38 biomolecules comprising peptides, proteins, and nucleic acid fragments, where S_{a} is computed by summing the areas of the triangles and ΔG_{solv} is computed using TABI-PB. In addition to examining the accuracy of these results, we report the total CPU time for the TABI-PB computation; the CPU time for generating the triangulation and other pre-processing steps is negligible in

comparison to the boundary element computation time. Surface visualizations were generated with VTK ParaView.

The biomolecules in the test set, listed in Table 4.1, are those with widely available PDB entries from the list used in a previous molecular surface comparison study[98]. We generate PQR files for each test biomolecule using PDB2PQR [101] with the CHARMM force field and water molecules removed.

Table 4.1: PDB ID and number of atoms for test set of 38 biomolecules comprising proteins, peptides, and nucleic acid fragments.

Index	1	2	3	4	5	6	7	8
PDB ID	2LWC	1GNA	1S4J	1CB3	1V4Z	1BTQ	1I2X	1AIE
# atoms	75	163	182	183	266	304	513	522
Index	9	10	11	12	13	14	15	16
PDB ID	1ZWF	375D	440D	4HLI	3ES0	3IM3	2IJI	1COA
# atoms	586	593	629	697	781	851	890	1057
Index	17	18	19	20	21	22	23	24
PDB ID	2AVP	1SM5	2ONT	4GSG	3ICB	1DCW	3LDE	1AYI
# atoms	1085	1137	1161	1195	1202	1257	1294	1365
Index	25	26	27	28	29	30	31	32
PDB ID	2YX5	3DFG	3LOD	1TR4	1RMP	1IF4	4DUT	3SQE
# atoms	1385	2198	2246	3423	3478	4071	4217	4647
Index	33	34	35	36	37	38		
PDB ID	1HG8	4DPF	3FR0	2H8H	2CEK	1IL5		
# atoms	4960	5824	6952	7084	8346	8349		

The MSMS triangulations were generated using density values $d = 1, 2, 4, 8, 16$ and the NanoShaper triangulations were generated using scaling parameter values $s = 1, 2, 3, 4, 5$. For all surfaces a probe radius of 1.4 Å was used.

The physical parameter values were ionic concentration $I_s = 0.15$ M, temperature $T = 300$ K, and solute and solvent dielectric constants $\varepsilon_1 = 1$, $\varepsilon_2 = 80$. The treecode parameters were multipole acceptance criterion $\theta = 0.8$, Taylor series order $p = 3$, and maximum number of particles in a leaf $N_0 = 500$. The GMRES tolerance was 1E–4, with 10 iterations between restarts and maximum number of iterations 110.

All computations were performed in serial on the University of Michigan FLUX cluster, using 2.8 GHz Intel Xeon E5-2680v2 processors. The code was compiled with the GCC Fortran compiler using the `-O2` optimization flag. The version of the TABI-PB solver used in this work is available on SourceForce at `sourceforge.net/projects/tabipb/`.

4.2.3 Results

We first study geometric features of the surface triangulations by considering the triangle size, shape, and aspect ratio, and qualitatively comparing the generated surface meshes. We then extrapolate with respect to the number of triangles to calculate highly accurate converged values of the surface area and solvation energy; the converged result is the y-intercept of a simple extrapolation of the computed surface area or solvation energy versus N^{-1} for the two highest resolution meshes produced by MSMS and NanoShaper. For some values of the density parameter d, MSMS produced spurious results for the larger molecules or failed to even produce a triangulation at all, as observed by previous investigators [102]; in these cases the extrapolation used the highest resolution meshes for which MSMS did not fail.

4.2.3.1 Triangulation filter

Both MSMS and NanoShaper produce a number of small or thin triangles which reduce the computational accuracy and efficiency, and it is common practice to delete them from the simulations. Hence in the present work, triangles are deleted if their area is less than 1E–5 $Å^2$ or if the distance between the centroids of two neighboring triangles is less than 1E–5 Å. Table 4.2 gives the percent of deleted triangles averaged over all triangulations using MSMS and NanoShaper. Among the deleted triangles, some had area less than machine precision and these are designated as zero-area triangles. With MSMS the deleted triangles are 0.064 % of the total, while with NanoShaper the total is more than 100 times smaller. Table 4.2 also shows that most of the deleted MSMS triangles were zero-area, while none of this type were produced by NanoShaper.

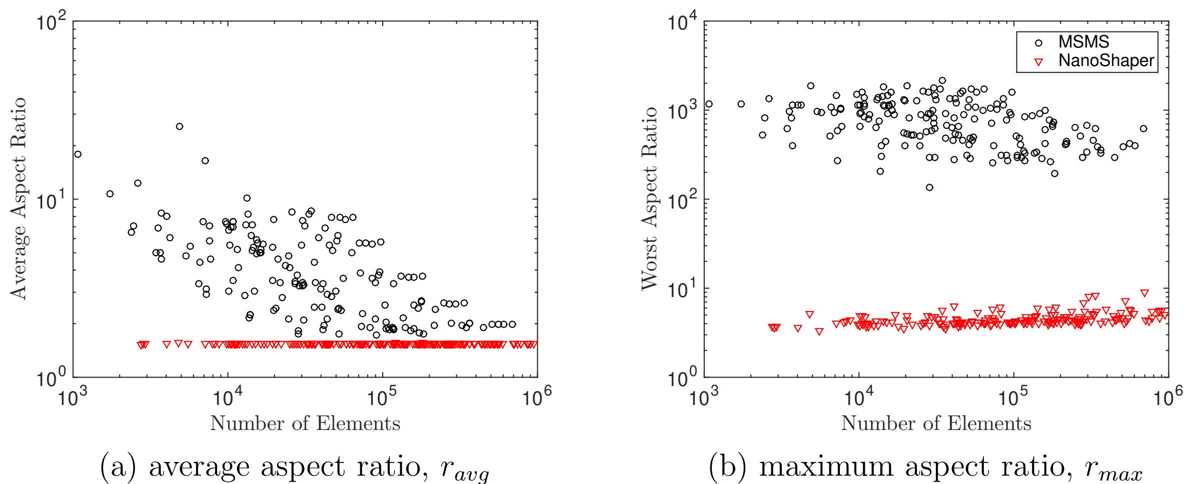
Table 4.2: Triangulation filter results showing percent of deleted triangles and zero-area triangles, values shown are averaged over all triangulations using MSMS and NanoShaper, zero-area triangles are a subset of deleted triangles.

code	deleted triangles	zero-area triangles
MSMS	6.4E–2 %	5.4E–2 %
NanoShaper	5.2E–4 %	0.0E–0 %

4.2.3.2 Triangle aspect ratios

Even after filtering the triangulation as above, the aspect ratio of the remaining triangles can affect the computational performance. The aspect ratio of a triangular surface element is defined as the ratio of the longest to shortest sides. Figure 4.1 displays the (a) average aspect

ratio, r_{avg}, and (b) maximum aspect ratio, r_{max}, versus the number of triangles, N, for each triangulation, where N varies from approximately 1E3 to 1E6, for the chosen density and scaling parameters. Figure 4.1(a) shows that the average aspect ratio of MSMS triangles is as large as $r_{avg} \approx 30$ for small N and decreases to approximately $r_{avg} \approx 2$ for large N, while the average aspect ratio of NanoShaper triangles is closer to $r_{avg} \approx 1$ for all N. Fig. 4.1(b) shows that the maximum aspect ratio of MSMS triangles varies between approximately 1E2 and 2E3, while the maximum aspect ratio of NanoShaper triangles is below 1E1 across all triangulations.



(a) average aspect ratio, r_{avg}

(b) maximum aspect ratio, r_{max}

Figure 4.1: Triangle aspect ratio versus number of elements N for each generated surface, (a) average aspect ratio, r_{avg}, (b) maximum aspect ratio, r_{max}, MSMS (black, $\circ$), NanoShaper (red, $\triangledown$).

4.2.3.3 Surface mesh features

Figure 4.2 displays the triangulation and surface potential for a representative protein (1AIE) using (a) MSMS and (b) NanoShaper with similar resolution, $N \approx$ 3E4 triangles in each case. The surfaces are similar at first glance, although the NanoShaper surface appears slightly smoother than the MSMS surface. Figure 4.3 displays a zoom of the triangulations, where several irregular features are highlighted; in the MSMS mesh, green boxes enclose *stitches* formed by high aspect ratio triangles, and a white box encloses a *cusp* formed by neighboring triangles that meet at a acute angle, while in the NanoShaper mesh, a yellow box encloses a possible irregular feature, which could in fact simply be an artifact of the surface lighting. It should be noted that irregular features are present in the MSMS mesh even after filtering; by contrast, the NanoShaper mesh is relatively free of such irregular features. The irregular features diminish the efficiency of the calculations; as shown below, calculations

using MSMS meshes require more iterations to converge in comparison with calculations using NanoShaper meshes.

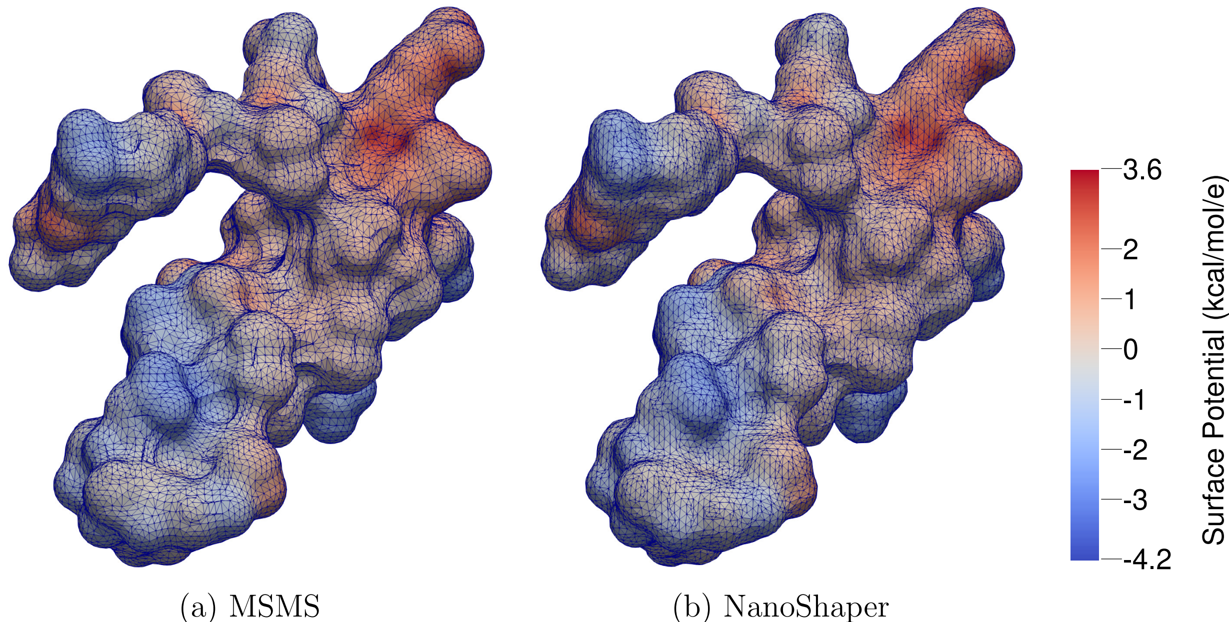


Figure 4.2: Protein 1AIE, SES triangulation and electrostatic potential, (a) MSMS, density $d = 6$, $N = 31480$ triangles, (b) NanoShaper, scaling parameter $s = 2$, $N = 32208$ triangles.

4.2.3.4 Dependence of S_a and $\Delta G_{\mathbf{solv}}$ on mesh resolution

In this section we examine the dependence of the surface area S_a and solvation energy ΔG_{solv} on the mesh resolution for four representative proteins (1AIE, 1HG8, 3FR0, 1IL5). Figure 4.4 plots S_a and Fig. 4.5 plots ΔG_{solv} versus N^{-1}, where N is the number of surface elements. As expected the MSMS and NanoShaper results for S_a and ΔG_{solv} converge to similar limits since both codes approximate the solvent excluded surface, but several differences can be seen in their dependence on N.

First, concerning the surface area in Fig. 4.4, the MSMS and NanoShaper results converge to almost the same values in Fig. 4.4(a,d) (1AIE, 1IL5), but in Fig. 4.4(b,c) (1HG8, 3FR0), the NanoShaper surface area is 2-3% larger than the MSMS surface area. In all cases the convergence with N^{-1} is smooth. The MSMS results approach their limit somewhat faster, although MSMS was unable to generate reliable meshes with larger values of N; either it fails to produce a mesh, or the generated mesh was poorly formed. The largest value we obtained using MSMS was $N \approx 2\text{E}6$, whereas NanoShaper had no such limitation. Hence if

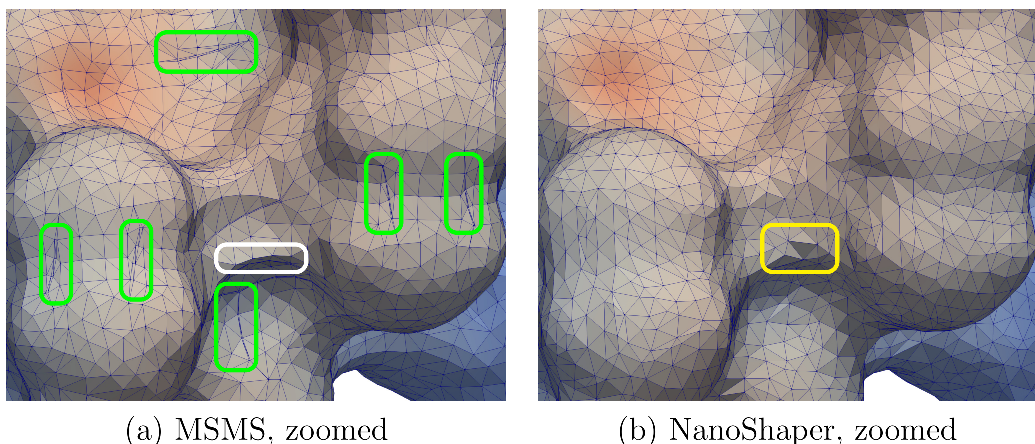


Figure 4.3: Protein 1AIE, zoom of SES triangulation, (a) MSMS, density $d = 6$, $N = 31480$ triangles, green boxes enclose *stitches* formed by high aspect ratio triangles, white box encloses a *cusp* formed by neighboring triangles that meet at an acute angle, (b) NanoShaper, scaling parameter $s = 2$, $N = 32208$ triangles, yellow box encloses a possible irregular feature.

it is necessary to generate a very dense mesh, or even a less dense mesh for a biomolecule with a large surface area, then NanoShaper has an advantage.

Second, concerning the solvation energy in Fig. 4.5, the MSMS and NanoShaper results converge to almost the same value. The MSMS results again approach their limiting value somewhat faster than the NanoShaper results, although the NanoShaper dependence on N is smoother than the MSMS dependence.

Figure 4.6 displays the (a) surface area S_a and (b) solvation energy ΔG_{solv} for the entire set of 38 biomolecules, where the NanoShaper results are plotted versus MSMS results. In this case to reduce the dependence of the computed values on the mesh resolution N, we extrapolated the computed S_a and ΔG_{solv} to the limit $N \to \infty$ using the two highest resolution meshes, density $d = 8, 16$ for MSMS and scaling factor $s = 4, 5$ for NanoShaper. The correspondence between MSMS and NanoShaper results in Fig. 4.6 is very good, except for two molecules, 1I2X and 375D, which consist of multiple domains, for which MSMS did not generate an accurate mesh. These two anomalous cases are indicated by the two markers furthest away from the diagonal line in Fig. 4.6(a,b). In addition, MSMS failed to produce surfaces for 13 other runs, and produced highly distorted surfaces with spurious solvation energy for 3 more runs. These 16 spurious runs were removed from the calculations in this section. By contrast, NanoShaper failed in only one case, a low resolution mesh with scaling factor $s = 1$ for the smallest molecule in the test set (2LWC, 75 atoms).

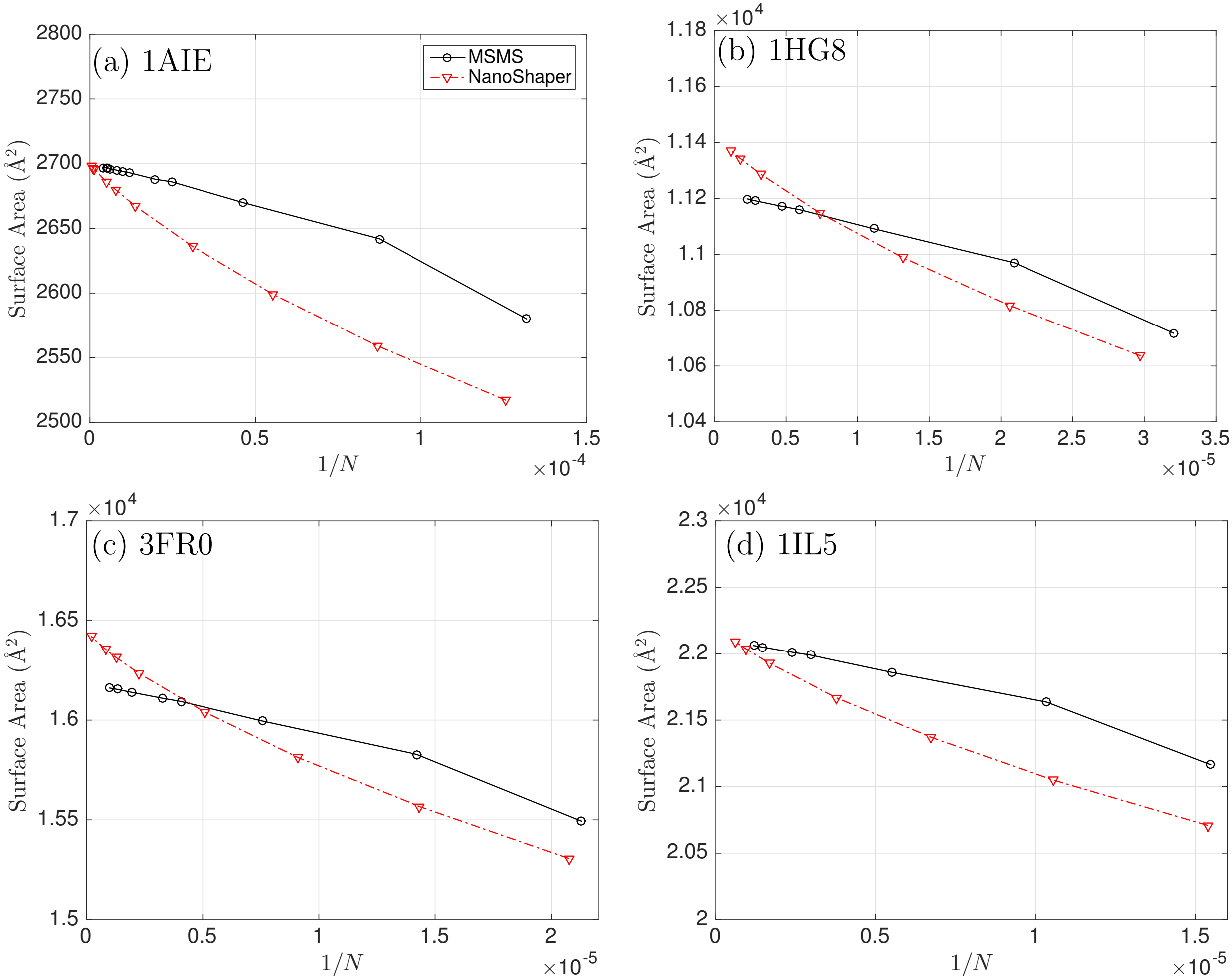


Figure 4.4: Surface area S_a versus N^{-1} for four representative proteins, where N is the number of triangles, MSMS (black, solid, $\circ$), NanoShaper (red, dashed, $\triangledown$).

4.2.3.5 Computational efficiency

Figure 4.7(a) displays the total TABI-PB run time versus the number of triangles N for computing the solvation energy ΔG_{solv} using MSMS and NanoShaper meshes, where the solid lines are least squares fits to the data. The run time for creating and filtering the meshes is less than eight seconds in all cases, and thus constitutes a negligible fraction of the total run time. The results show that in general, NanoShaper meshes require less run time than MSMS meshes. This is supported by Fig. 4.7(b) showing the number of GMRES iterations in each case, where the maximum number of iterations was set to 110. The results show that in general, NanoShaper meshes require fewer GMRES iterations than MSMS meshes. Moreover, in the case of MSMS, the iteration limit was reached in 23 out 177 meshes, while in the case of NanoShaper, the iteration limit was never reached.

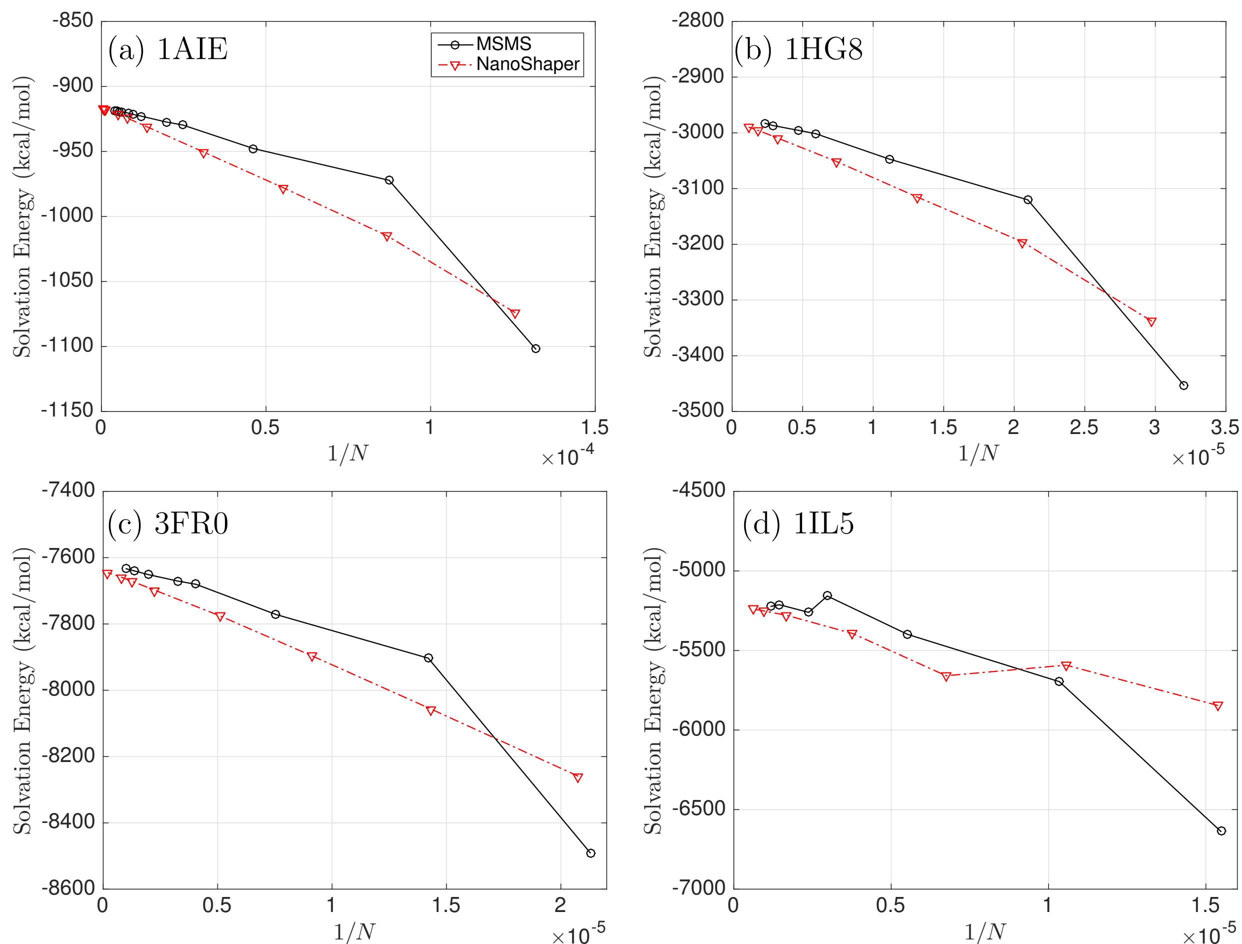


Figure 4.5: Solvation energy E_{sol} versus N^{-1} for four representative proteins, where N is the number of triangles, MSMS (black, solid, ∘), NanoShaper (red, dashed, ▽).

Table 4.3 displays the average run time and average number of GMRES iterations per triangle for each mesh type over the entire set of 38 biomolecules. The results show that NanoShaper meshes require about 2/3 of the run time and 1/4 of the number of iterations required by MSMS meshes. The larger number of GMRES iterations required for MSMS meshes is attributed to the presence of triangles with large aspect ratio, as seen in Fig. 4.1, and irregular features in the generated surfaces, as seen in Fig. 4.3.

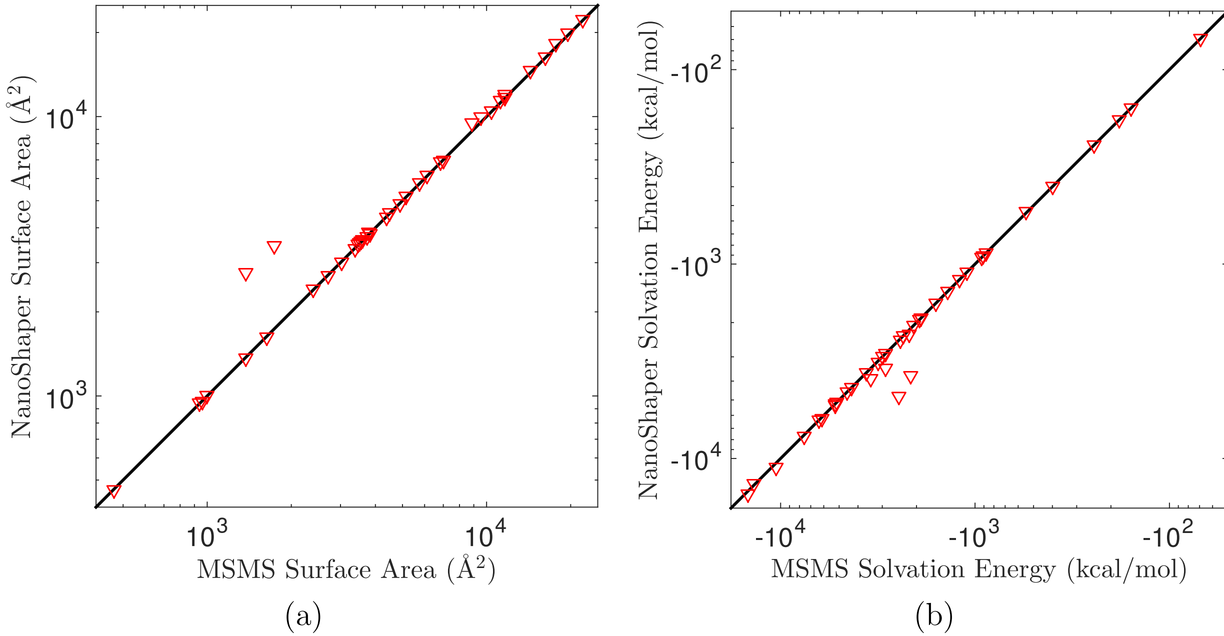


Figure 4.6: NanoShaper versus MSMS results for entire set of 38 biomolecules using values extrapolated to the limit $N \to \infty$, (a) surface area S_a, (b) solvation energy ΔG_{solv}, black lines indicate perfect correspondence.

Table 4.3: Average run time (s) and average number of GMRES iterations per triangle for MSMS and NanoShaper meshes over entire set of 38 biomolecules. Simulations ran in serial on Intel Xeon CPU.

	average run time (s)/triangle	average iterations/triangle
MSMS	6.67E–3	1.17E–3
NanoShaper	4.19E–3	2.92E–4

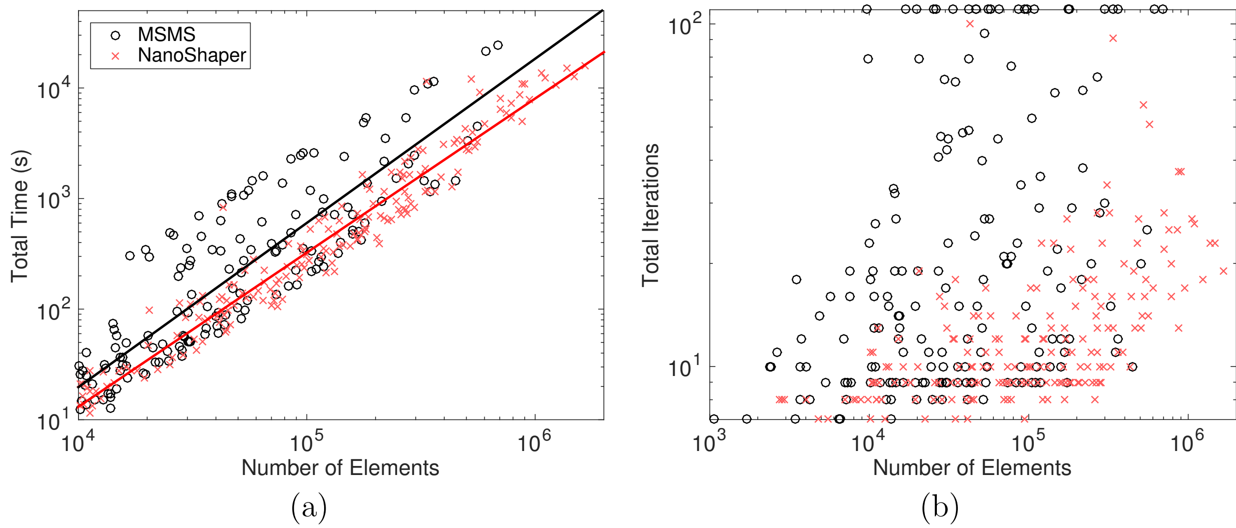


Figure 4.7: Computational efficiency, total run time of TABI-PB for computing solvation energy E_{sol} using MSMS (black, $\circ$) and NanoShaper (red, $\times$) versus number of triangles N, (a) run time (s), solid lines are least squares fits, (b) number of GMRES iterations (maximum 110). Simulations ran in serial on Intel Xeon CPU.

4.3 Project 2: Implementation of node patch method

4.3.1 Project description

Previously, TABI-PB has used a constant element centroid collocation scheme for discretizing the boundary integral problem. Another strategy is the so-called node patch scheme [12], in which the boundary integral equations are discretized so that the problem is computed on the vertices of the surface mesh elements instead of the faces. Areas are assigned to these vertex nodes by summing one-third the area of all faces that share the vertex. This process is depicted in Fig. 4.8. Note that, because there are significantly less vertices than faces, this procedure results in a large reduction of problem unknowns without decreasing the resolution of the mesh. This project implements and compares a node patch scheme within TABI-PB to the previous centroid collocation scheme.

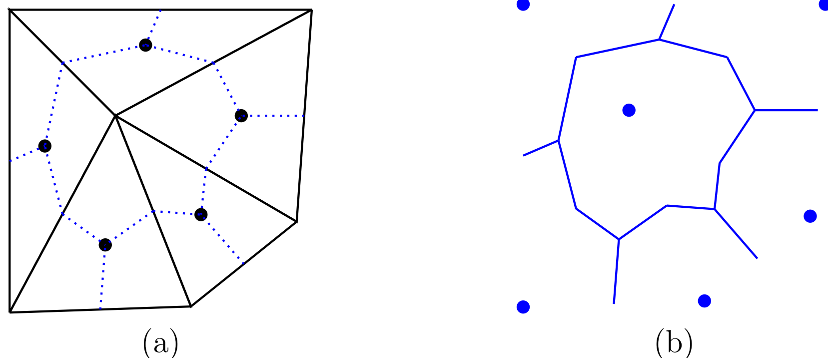


Figure 4.8: Construction of a node patch from faces of a surface mesh. (a) Five faces meet at a vertex. A patch shown in (b) is formed by taking one third of the area of each triangle surrounding the vertex.

4.3.2 Methodology

To test our implementation of this scheme, we use the same test set as well as the same physical and computational parameters described in §4.2.2. All surfaces were generated with NanoShaper. All computations were performed in serial on the University of Michigan Flux cluster, using 2.67 GHz Intel Xeon X5650 processors. This investigation was performed using the C rewrite of TABI-PB first developed for integrating the package into APBS. The code was compiled with GCC C compiler using the `-O3` optimization flag.

4.3.3 Results

In Fig. 4.9, we compare the convergence of solvation energy values with respect to refining the surface mesh for the newly implemented node patch and the original collocation, for

four representative proteins (the same four as in Section §4.2.3.4). The four plots depict $1/N$ versus solvation energy, where N is the number of computational elements, i.e., the number of mesh faces for collocation and the number of mesh vertices for node patch. Note that, for all four proteins, the node patch result appears to converge much faster, suggesting that, for a certain required level of error, the node patch method may use significantly fewer computational elements and take less CPU time.

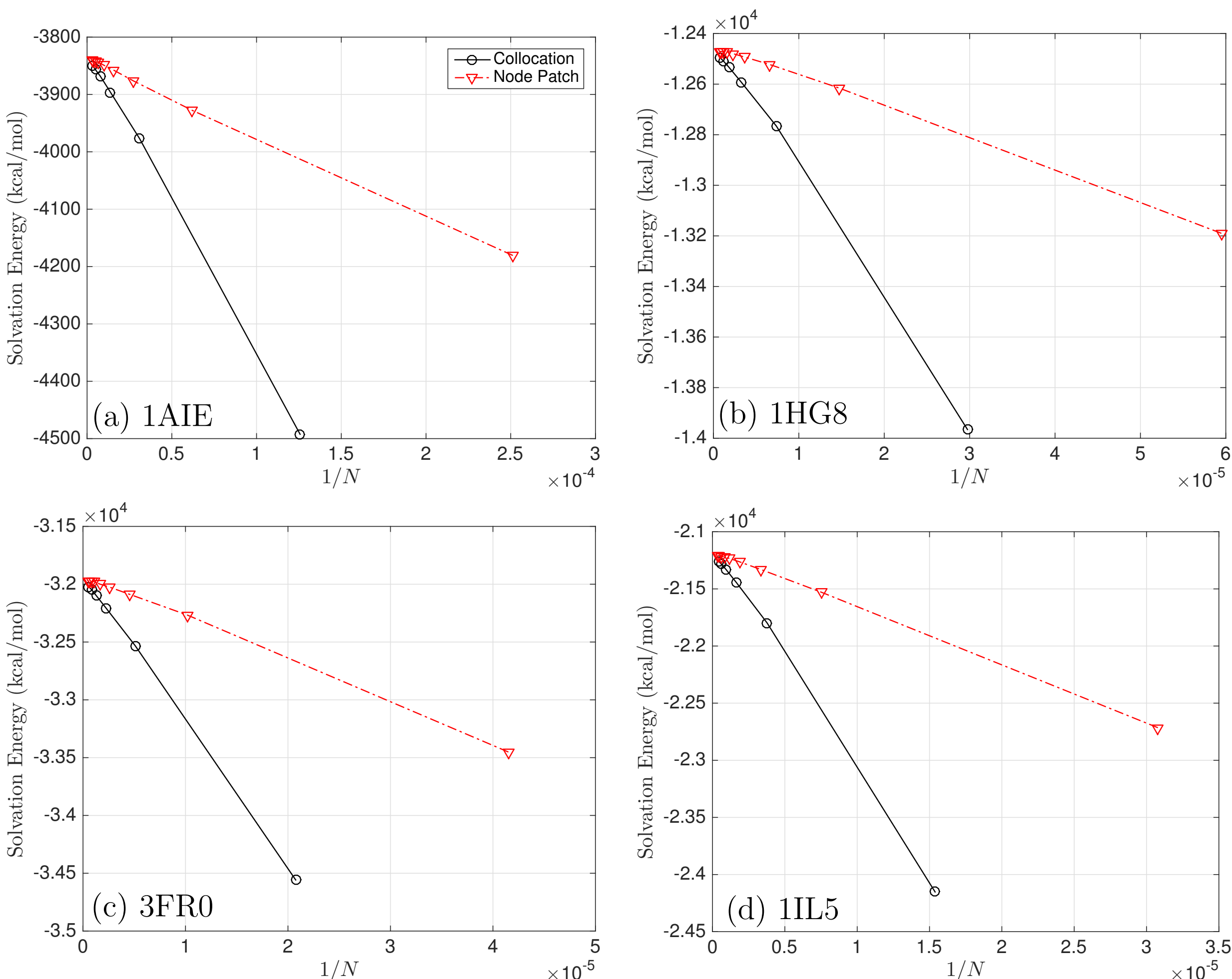


Figure 4.9: $1/N$ versus solvation energy for four representative proteins, where N is number of computational elements (faces for collocation, vertices for node patch), collocation (black, solid, $\circ$), node patch (red, dashed, $\triangledown$).

In Fig. 4.10, we show the relative error in the solvation energy versus total CPU time for the same four proteins, comparing node patch, the version of collocation in the original APBS-distributed TABI-PB (old collocation), and the version of collocation in a refactoring of the code performed before implementing node patch. We note here that there is no difference between new and old collocation in Fig. 4.9 because the refactoring only impacted

the computational performance, not the computational result itself. The converged result against which the relative errors are calculated was taken from a linear extrapolation of the solvation energies of the three highest resolution results. The error for collocation was computed against the collocation converged result, and the error for node patch was computed against the node patch converged result. The code refactoring produced a small speedup, around 1.25×, between the old and new collocation. The node patch method produces a much larger additional speedup, around 5 to 10×. All of these computations were performed with a GMRES tolerance of 1E–4; thus, the odd behavior of the node patch error convergence in Fig. 4.10(c) is simply noise and all relative error smaller than 1E–4 should be treated as spurious. Using the same converged result for both node patch and collocation for computing relative errors, for example, the collocation extrapolated result, produces very similar results.

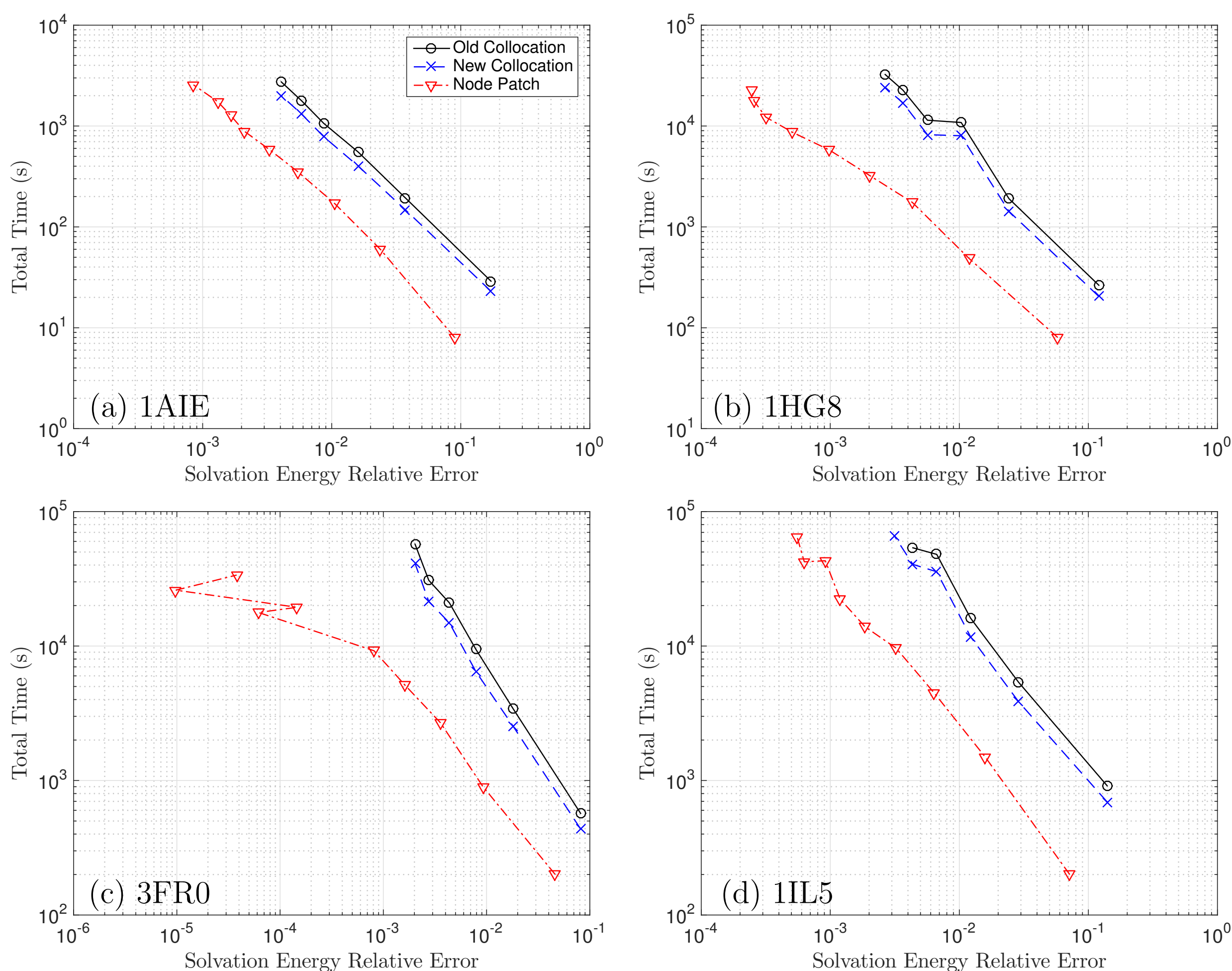


Figure 4.10: Solvation energy relative error versus CPU time for four representative proteins, collocation in original C rewrite of TABI-PB (black, solid, ○), collocation in code refactored TABI-PB (blue, dashed, ×), node patch (red, dashed, ▽). Note that the GMRES tolerance is 1E–4. Simulations ran in serial on Intel Xeon CPU.

Figure 4.11 shows the computational performance of TABI-PB with collocation and with node patch across the entire data set, where relative error is computed as described above. It is clear from both graphs that, for problems of all sizes and geometries, node patch in general requires less computational elements and less CPU time than collocation to reach the same level of accuracy.

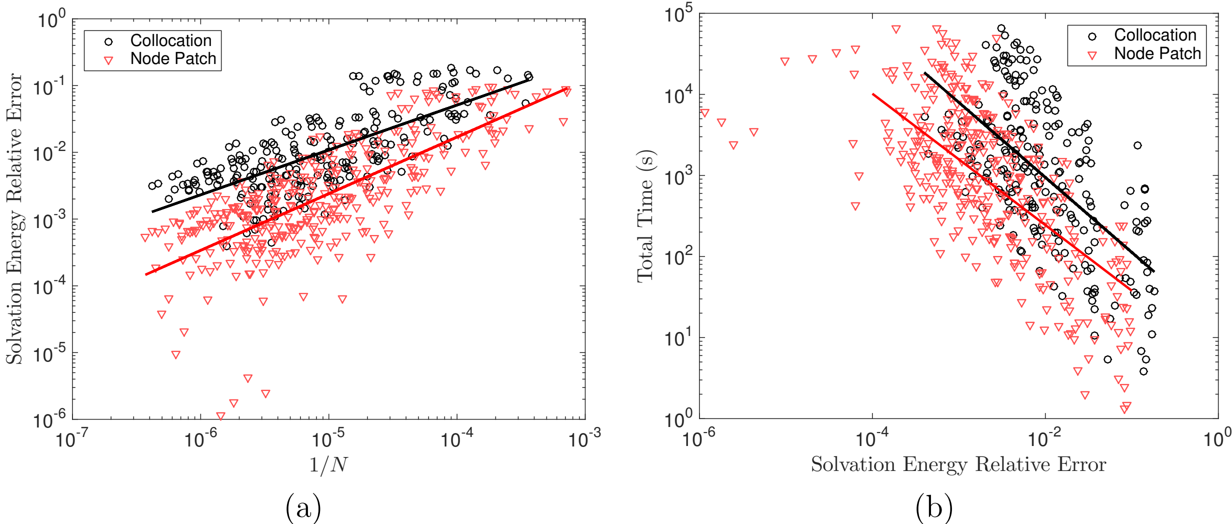


Figure 4.11: Computational performance of TABI-PB using collocation (black, ∘) and node patch (red, ▽), over all runs from test set, (a) $1/N$ versus solvation energy error, (b) solvation energy error versus run time. Simulations ran in serial on Intel Xeon CPU.

4.4 Project 3: GPU-accelerated BLDTT TABI-PB

4.4.1 Project description

Previous versions of TABI-PB used $O(N \log N)$ Taylor treecodes to compute the matrix-vector product in GMRES. Here we apply the recently introduced $O(N)$ barycentric Lagrange dual tree traversal fast summation method (BLDTT) [7] detailed in Chapter 3. In addition, we develop a GPU-accelerated version of TABI-PB based on the BLDTT.

Implementing the BLDTT in TABI-PB necessitates a full rewrite of TABI-PB. Note that the required matrix-vector product consists of evaluating sums on the right hand side of Eq. 4.13. This simply amounts to evaluating linear combinations of G_0, G_κ and their normal derivatives. Instead of the 16 potentials present in the original TABI-PB implementation as presented in [26], we reorganize these linear combinations into four potentials with four target charges and four source charges, shown in Table 4.4.

Following from the implementation of the BLDTT in BaryTree, the GPU-accelerated version of TABI-PB uses OpenACC pragmas. For the computations in the matrix-vector

product, their implementation is virtually identical to that described in Chapter 3. OpenACC pragmas are additionally used to accelerate computation of the source terms S_1 and S_2 by Eq. 4.11 and computation of the solvation energy. This rewrite utilized the node patch method for generating the linear system. We also note that this rewrite was done in object-oriented C++.

ℓ	Target Charge $p_{i\ell}$	Source Charge $q_{j\ell}$
0	1	$\dfrac{\partial \phi_1(\mathbf{x}_j)}{\partial n}$
1	$n_1(\mathbf{x}_i)$	$n_1(\mathbf{x}_j)\,\phi_1(\mathbf{x}_j)\,A_j$
2	$n_2(\mathbf{x}_i)$	$n_2(\mathbf{x}_j)\,\phi_1(\mathbf{x}_j)\,A_j$
3	$n_3(\mathbf{x}_i)$	$n_3(\mathbf{x}_j)\,\phi_1(\mathbf{x}_j)\,A_j$

(a) Charges

Term	Value
V_0	$q_{j0}(G_0 - G_\kappa) + \sum_{\ell=1}^{3} q_{j\ell} n_\ell(\mathbf{x}_j)\,\partial_{x_{j\ell}}(\varepsilon G_\kappa - G_0)$
V_1	$q_{j0} n_1(\mathbf{x}_i)\,\partial_{x_{i1}}(G_\kappa/\varepsilon - G_0) + \sum_{\ell=1}^{3} q_{j\ell} n_1(\mathbf{x}_i)\, n_\ell(\mathbf{x}_j)\,\partial_{x_{i1}}\partial_{x_{j\ell}}(G_0 - G_\kappa)$
V_2	$q_{j0} n_2(\mathbf{x}_i)\,\partial_{x_{i2}}(G_\kappa/\varepsilon - G_0) + \sum_{\ell=1}^{3} q_{j\ell} n_2(\mathbf{x}_i)\, n_\ell(\mathbf{x}_j)\,\partial_{x_{i2}}\partial_{x_{j\ell}}(G_0 - G_\kappa)$
V_3	$q_{j0} n_3(\mathbf{x}_i)\,\partial_{x_{i3}}(G_\kappa/\varepsilon - G_0) + \sum_{\ell=1}^{3} q_{j\ell} n_3(\mathbf{x}_i)\, n_\ell(\mathbf{x}_j)\,\partial_{x_{i3}}\partial_{x_{j\ell}}(G_0 - G_\kappa)$

(b) Potentials

Table 4.4: (a) Target and source charges $p_{i\ell}$ and $q_{j\ell}$ for computing the matrix-vector product in GMRES, (b) terms for computing the matrix-vector product in GMRES. For a given source particle $\mathbf{x}_j$ and target particle $\mathbf{x}_i$, the input vector values for $\mathbf{x}_j$ are $\phi_1(\mathbf{x}_j)$ and $\frac{\partial \phi_1(\mathbf{x}_j)}{\partial n}$. The contribution to the output value $\phi_1(\mathbf{x}_i)$ is $p_{i0}V_0$, and the contribution to $\frac{\partial \phi_2(\mathbf{x}_i)}{\partial n}$ is $p_{i1}V_1 + p_{i2}V_2 + p_{i3}V_3$.

4.4.2 Methodology

For the CPU results, we investigate number of surface elements versus total TABI-PB run time and solvation energy relative error versus run time for a set of four test biomolecules with PDB IDs 451C (1215 atoms, a cytochrome protein of *Pseudomonas aeruginosa*, involved in redox catalysis and the electron transport chain), 1A63 (2065 atoms, RNA binding domain of *E. coli* rho factor, involved in transcription termination), 3SQE (4647 atoms, a domain of human prothrombin), and 1IL5 (8350 atoms, the A chain of ricin), comparing the Taylor treecode TABI-PB and the BLDTT TABI-PB.

We generate PQR files for each test biomolecule using PDB2PQR [101] with the CHARMM force field and water molecules removed. The NanoShaper triangulations were generated using scaling parameter values $s = 1.0, 1.5, 2.0, 2.5, 3.0, 3.5, 4.0$. The solvation energy relative errors were calculated with respect to converged solvation energy values extrapolated from the three highest density runs from a direct summation boundary integral PB run (TABI-PB with $\theta = 0.0$). For all surfaces a probe radius of 1.4 Å was used.

The physical parameter values were ionic concentration $I_s = 0.15$ M, temperature $T = 300$ K, and solute and solvent dielectric constants $\varepsilon_1 = 1$, $\varepsilon_2 = 80$. For the Taylor treecode TABI-PB, the fast summation parameters were multipole acceptance criterion $\theta = 0.8$, Taylor series order $p = 3$, and maximum number of particles in a leaf $N_0 = 500$. For the BLDTT TABI-PB, the fast summation parameters were multipole acceptance criterion $\theta = 0.8$, interpolation degree $n = 3$, and maximum number of particles in a leaf $N_0 = 50$. The GMRES tolerance was 1E–4, with 10 iterations between restarts and maximum number of iterations 110. These leaf sizes are respectively the optimal choices for the two implementations. The other parameters were chosen to maintain consistency between the implementations while running comparisons.

The computations were performed in serial on the standard compute nodes of the San Diego Supercomputer Center (SDSC) Comet machine, with Intel Xeon E5-2680v3 CPUs running at 2.5 GHz. Each computation was run on a single core. The code was compiled with the GCC C++ compiler using the `-O3` optimization flag.

For the GPU results, we compare the run time for the BLDTT TABI-PB on a single CPU core to the run time for BLDTT TABI-PB on a single GPU for biomolecule with PDB ID 1A63. For the GPU runs, the parameters listed above are the same, except for N_0, which is 2000. The GPU computations were performed on the NVIDIA P100 GPU nodes of the SDSC machine, where each node contains four GPUs, and each GPU has 16GB of memory. Each computation was run on a single GPU. The code was compiled with the PGI C++ compiler using the `-O3` optimization flag.

The current BLDTT TABI-PB solver used in this work is available on GitHub at `github.com/Treecodes/TABI-PB`.

4.4.3 Results

To demonstrate scaling, Fig. 4.12 depicts the number of surface elements versus total run time (s) for the four representative biomolecules 451C, 1A63, 3SQE, and 1IL5. For all four cases, the BLDTT TABI-PB (blue, $\circ$) shows clearly better scaling than the Taylor treecode TABI-PB (red, $\triangledown$).

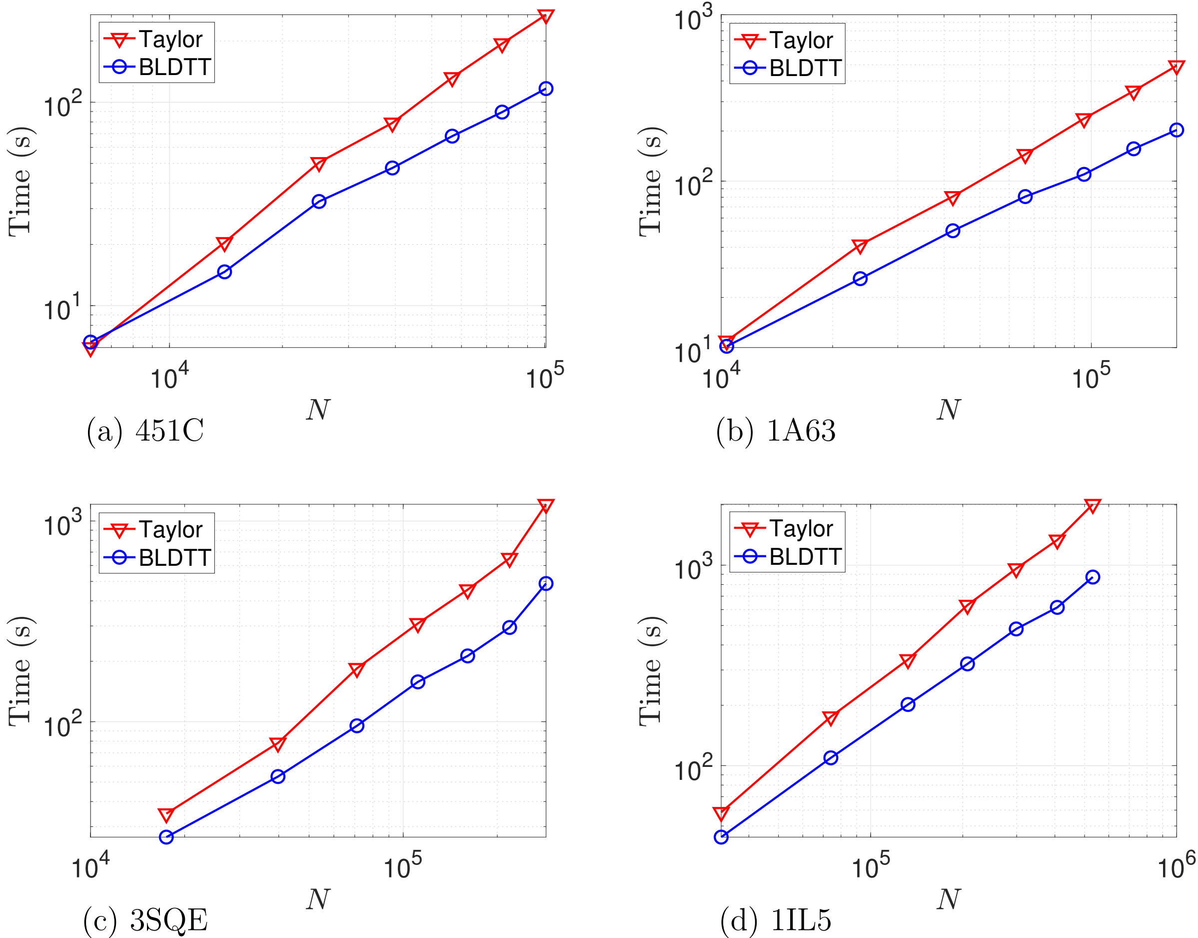


Figure 4.12: Number of surface elements versus run time (s) for four representative proteins, Taylor treecode TABI-PB (red, ▽), BLDTT TABI-PB (blue, ∘), NanoShaper scaling parameters 1.0, 1.5, 2.0, 2.5, 3.0, 3.5, 4.0, DS run as BLDTT TABI-PB with $\theta = 0.0$, TC run with $\theta = 0.8$, $p = 3$, $N_0 = 500$, DTT run with $\theta = 0.8$, $n = 3$, $N_0 = 50$. Simulations ran in serial on Intel Xeon CPU.

Figure 4.13 depicts solvation energy relative error versus total run time (s) for the four representative biomolecules, where the relative error is calculated with respect to converged solvation energy values extrapolated from the three highest density meshes from a direct summation run of TABI-PB (TABI-PB with $\theta = 0.0$). The BLDTT TABI-PB clearly achieves consistently lower error than the Taylor treecode TABI-PB.

Table 4.5 breaks down the results for 1A63, showing for all NanoShaper scaling parameters run the number of elements, solvation energy (kcal/mol), solvation energy relative errors, and run time (s) for direct sum boundary integral PB (TABI-PB run with $\theta = 0.0$), labeled as DS, Taylor treecode TABI-PB, labeled as TC, and BLDTT TABI-PB, labeled as DTT. The DS error is computed with respect to the extrapolated solvation energy result, given

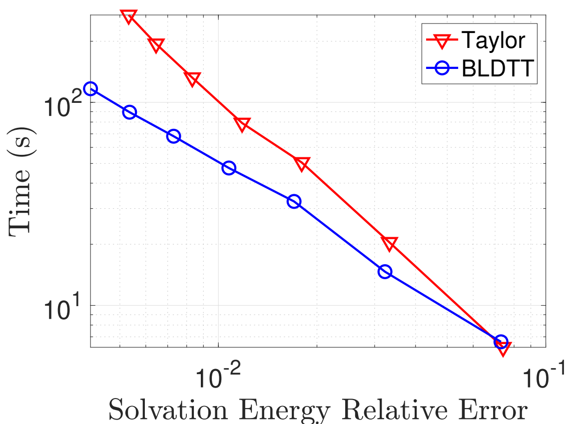
in the last row. The TC and DTT errors are computed with respect to the corresponding direct sum solvation energy result. Note that, for each mesh, DTT is not only faster than TC, with DTT being 2.5× faster at mesh scale 4.0, but the fast summation error is more than 5× smaller for DTT than TC. This suggests that the interpolation degree for DTT could be made even smaller while maintaining the same solvation energy error with respect to the converged direct sum value.

Scale	N	ΔG_{solv} (kcal/mol)			ΔG_{solv} Error (%)			Time (s)		
		DS	TC	DTT	DS	TC	DTT	DS	TC	DTT
1.0	10,350	-2587.68	-2589.32	-2587.95	8.13	0.0632	0.0104	53	11	10
1.5	23,754	-2479.64	-2480.65	-2479.76	3.62	0.0411	0.0051	330	41	26
2.0	42,260	-2443.14	-2444.24	-2443.14	2.09	0.0450	0.0114	1129	81	50
2.5	66,312	-2424.39	-2425.27	-2424.55	1.31	0.0362	0.0068	2941	145	81
3.0	95,568	-2415.12	-2416.21	-2415.36	0.92	0.0451	0.0097	6129	238	110
3.5	130,146	-2409.25	-2410.37	-2409.40	0.67	0.0465	0.0061	12,150	348	156
4.0	170,064	-2405.49	-2407.29	-2405.73	0.52	0.0746	0.0096	20,683	494	203
	∞	-2393.12								

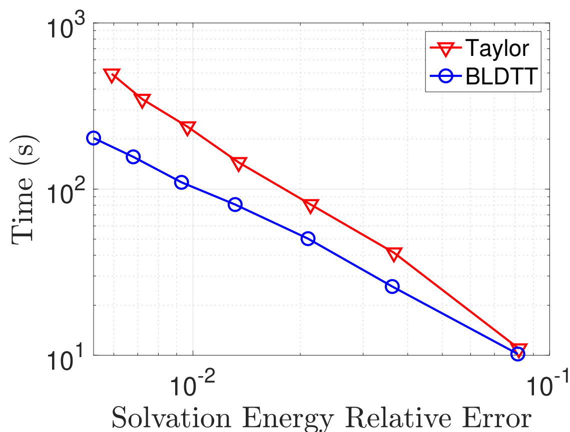
Table 4.5: Solvation energy (kcal/mol), solvation energy relative error, and run time (s) for direct sum boundary integral PB (DS), Taylor treecode TABI-PB (TC), and BLDTT TABI-PB (DTT), for PDB ID 1A63, NanoShaper scaling parameters 1.0, 1.5, 2.0, 2.5, 3.0, 3.5, 4.0, DS run as BLDTT TABI-PB with $\theta = 0.0$, TC run with $\theta = 0.8$, $p = 3$, $N_0 = 500$, DTT run with $\theta = 0.8$, $n = 3$, $N_0 = 50$. Simulations ran in serial on Intel Xeon CPU.

We demonstrate the speedups provided by the GPU-accelerated version of the BLDTT TABI-PB. Figure 4.14 shows the number of surface elements versus run time (s) for all NanoShaper scaling parameters run for a single CPU core (red, ∘) and an NVIDIA P100 GPU (blue, ∘). While (a) includes NanoShaper surface meshing in the total run time, (b) excludes it. Note that the NanoShaper software is CPU only, and so is the only part of the code run that is not GPU-accelerated. Figure 4.14(a) shows rather poor speedups of only 10×, but excluding NanoShaper and including only TABI-PB code in (b) shows a more respectable speedup of about 20×.

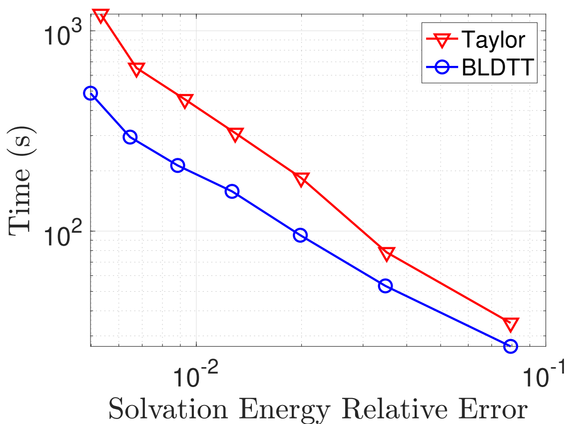
Figure 4.15 breaks down the components of total run time for the (a) CPU and (b) GPU runs in Fig. 4.14. In all cases, over 40% of the total GPU-accelerated BLDTTT TABI-PB run time in (b) is NanoShaper mesh building. As the number of surface elements increases, the % run time spent on actual TABI-PB computation phases (upward pass, downward pass, computing interactions) increases, from about 10% at scale 1.0, to about 50% at scale 4.0. For the CPU runs in (a), the % run time spent on TABI-PB computation phases is consistently around 70%. Notably, computing the source terms and solvation energy ΔG_{solv} are non-negligible contributors to the total time for the CPU runs; for the GPU runs, their contributions are completely negligible.



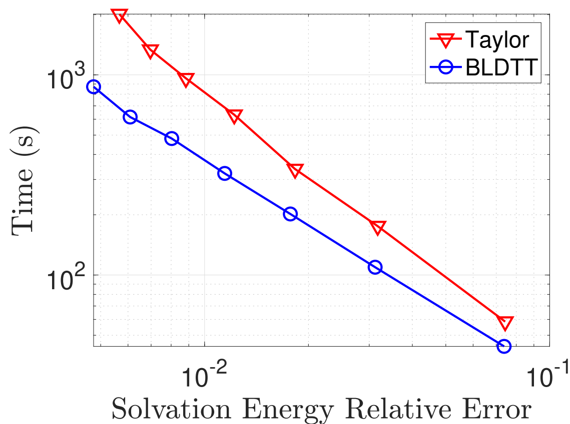
(a) 451C



(b) 1A63



(c) 3SQE



(d) 1IL5

Figure 4.13: Solvation energy relative error versus run time (s) for four representative proteins, Taylor treecode TABI-PB (red, $\triangledown$), BLDTT TABI-PB (blue, $\circ$), NanoShaper scaling parameters 1.0, 1.5, 2.0, 2.5, 3.0, 3.5, 4.0, DS run as BLDTT TABI-PB with $\theta = 0.0$, TC run with $\theta = 0.8$, $p = 3$, $N_0 = 500$, DTT run with $\theta = 0.8$, $n = 3$, $N_0 = 50$. Simulations ran in serial on Intel Xeon CPU.

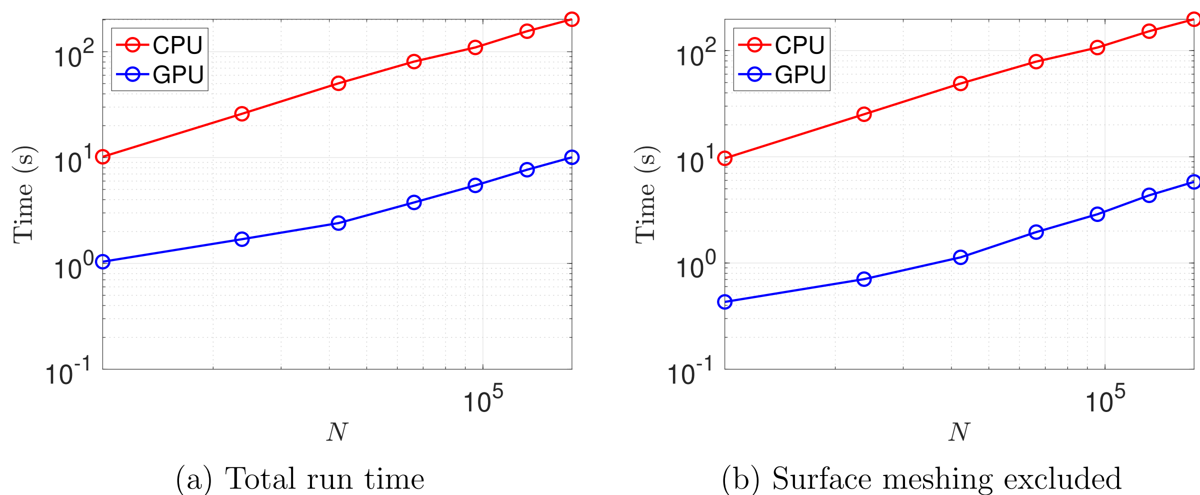


Figure 4.14: Number of surface elements versus run time (s) for a single Intel Xeon CPU core (red, ∘) versus NVIDIA P100 GPU (blue, ∘), across NanoShaper scaling parameters 1.0, 1.5, 2.0, 2.5, 3.0, 3.5, 4.0, PDB ID 1A63, (a) all components of run time including surface meshing with NanoShaper, (b) surface meshing time excluded. Note that the NanoShaper surface meshing software is CPU only, and is the only part of the code that has not been GPU-accelerated.

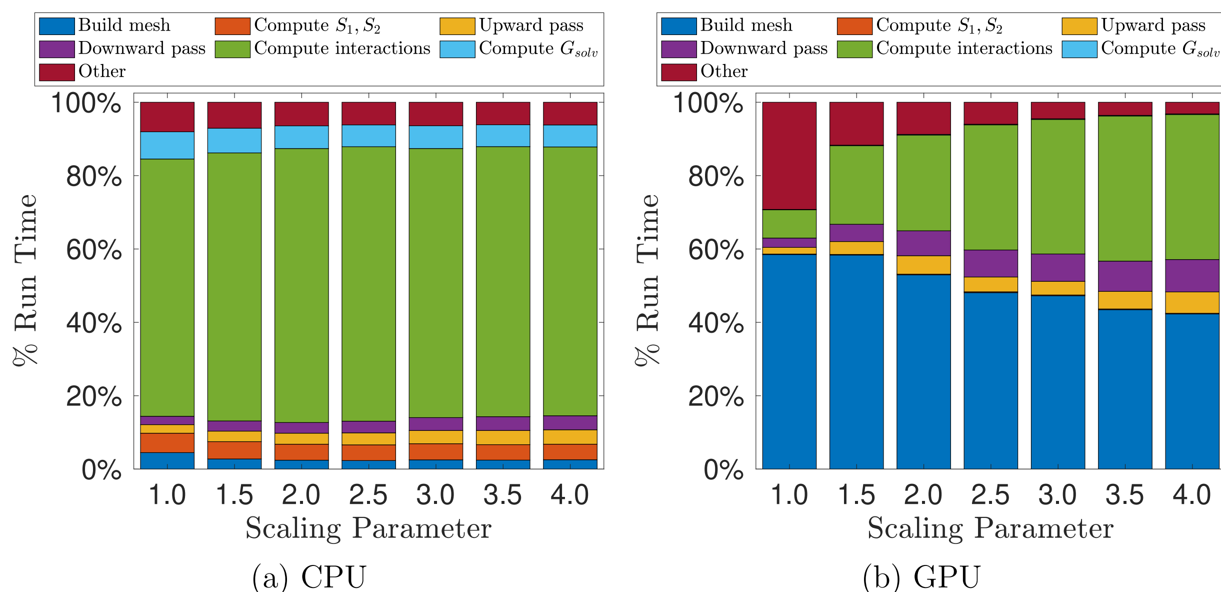


Figure 4.15: Component breakdown of run time across NanoShaper scaling parameters 1.0, 1.5, 2.0, 2.5, 3.0, 3.5, 4.0, PDB ID 1A63, (a) single Intel Xeon CPU core, (b) NVIDIA P100 GPU, build NanoShaper surface mesh (blue), compute source terms (orange), upward pass (yellow), downward pass (purple), compute interactions (green), compute ΔG_{solv}(light blue), other (burgundy) which includes tree building, interaction list building, and all parts of GMRES other than the matrix-vector product.

4.5 Application: Electrostatic binding free energy calculation

4.5.1 Project description

The end goal of developing TABI-PB is to produce a package that is useful in analyzing practical problems in protein solvation. We detail here the actual application of the TABI-PB software to a data set of protein complexes, in which we investigate the ability of TABI-PB to calculate static binding solvation energies. In the future, we hope to expand the application of our software to a greater range of problems, including comparisons with experimental data, using the improvements detailed above.

A common use of the Poisson–Boltzmann model is the calculation of electrostatic free energies of binding between proteins and ligands or other biomolecular structures. This project investigates the use of TABI-PB on calculating these energies for a standard test set of monomers, comparing the performance of our software to MIBPB, a common finite-difference Poisson–Boltzmann solver.

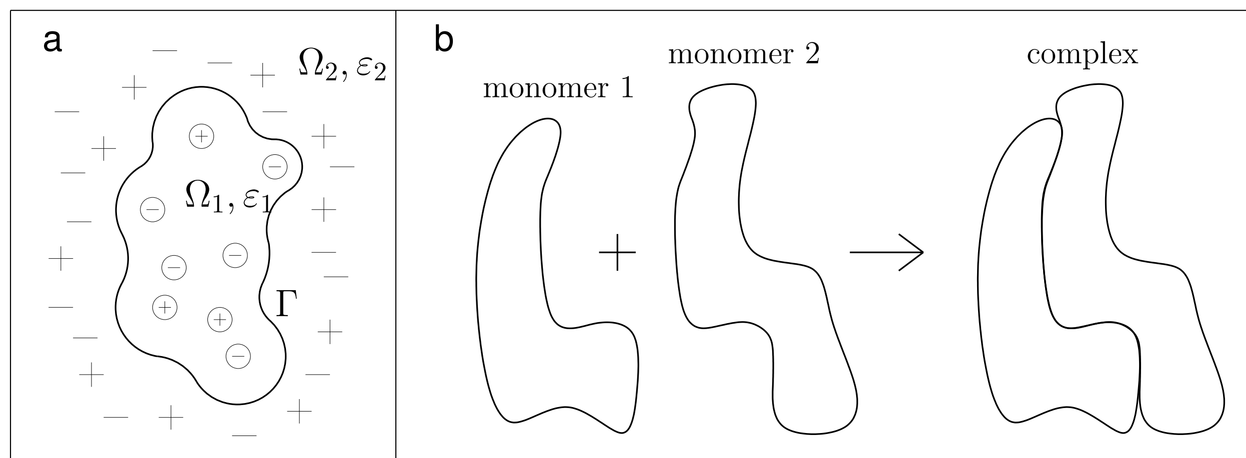


Figure 4.16: (a) The Poisson–Boltzmann model of biomolecular solvation, where Ω_1 is the solute domain with dielectric constant ε_1, Ω_2 is the solvent domain with dielectric constant ε_2, and Γ is the molecular surface. (b) The binding model for two biomolecular monomers. If the electrostatic solvation energy for monomer A is $G^{\mathrm{A}}_{\mathrm{solv}}$, for monomer B is $G^{\mathrm{B}}_{\mathrm{solv}}$, and for the bound complex is $G^{\mathrm{AB}}_{\mathrm{solv}}$, then the binding solvation energy ΔG_{solv} is the difference between $G^{\mathrm{AB}}_{\mathrm{solv}}$ and $G^{\mathrm{B}}_{\mathrm{solv}} + G^{\mathrm{B}}_{\mathrm{solv}}$

The molecular binding model in Fig. 4.16(b) shows monomer 1 and monomer 2 binding together to form a complex. We note here that all solvation energies discussed in this section only concern the electrostatic component. From the thermodynamic loop in Fig. 4.17, the

electrostatic binding free energy of the complex is

$$\Delta\Delta G_{\text{bind}} = \Delta E_{\text{coul}} + \Delta\Delta G_{\text{solv}} \tag{4.14a}$$

$$= (E^{\text{AB}}_{\text{coul}} - E^{\text{A}}_{\text{coul}} - E^{\text{B}}_{\text{coul}}) + (\Delta G^{\text{AB}}_{\text{solv}} - \Delta G^{\text{A}}_{\text{solv}} - \Delta G^{\text{B}}_{\text{solv}}). \tag{4.14b}$$

Rearranging the terms, this can also be expressed as

$$\Delta\Delta G_{\text{bind}} = (E^{\text{AB}}_{\text{coul}} + \Delta G^{\text{AB}}_{\text{solv}}) - (E^{\text{A}}_{\text{coul}} + \Delta G^{\text{A}}_{\text{solv}}) - (E^{\text{B}}_{\text{coul}} + \Delta G^{\text{B}}_{\text{solv}}), \tag{4.15}$$

where we see that the electrostatic binding free energy of the complex is the difference between the electrostatic free energy of the complex and the two independent monomers. The more negative the binding free energy is, the more likely binding will occur spontaneously.

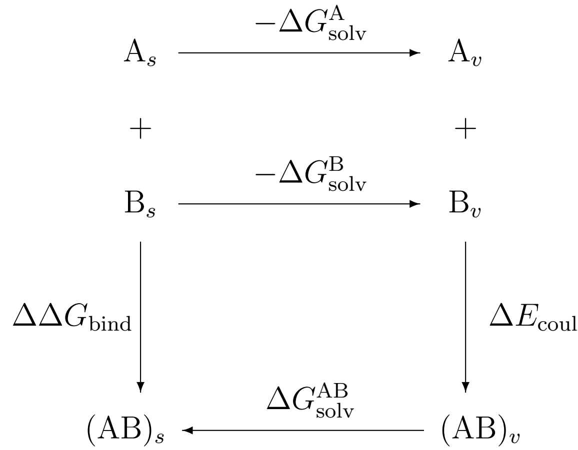


Figure 4.17: Thermodynamic loop illustrating the binding of two monomers A and B, only electrostatic interactions considered. Subscripts s, v denote solvent and vacuum, respectively. Electrostatic binding free energy $\Delta\Delta G_{\text{bind}}$ is determined by the change in vacuum electrostatic energy ΔE_{coul} and electrostatic binding solvation energy $\Delta\Delta G_{\text{solv}} = \Delta G^{\text{AB}}_{\text{solv}} - \Delta G^{\text{A}}_{\text{solv}} - \Delta G^{\text{B}}_{\text{solv}}$.

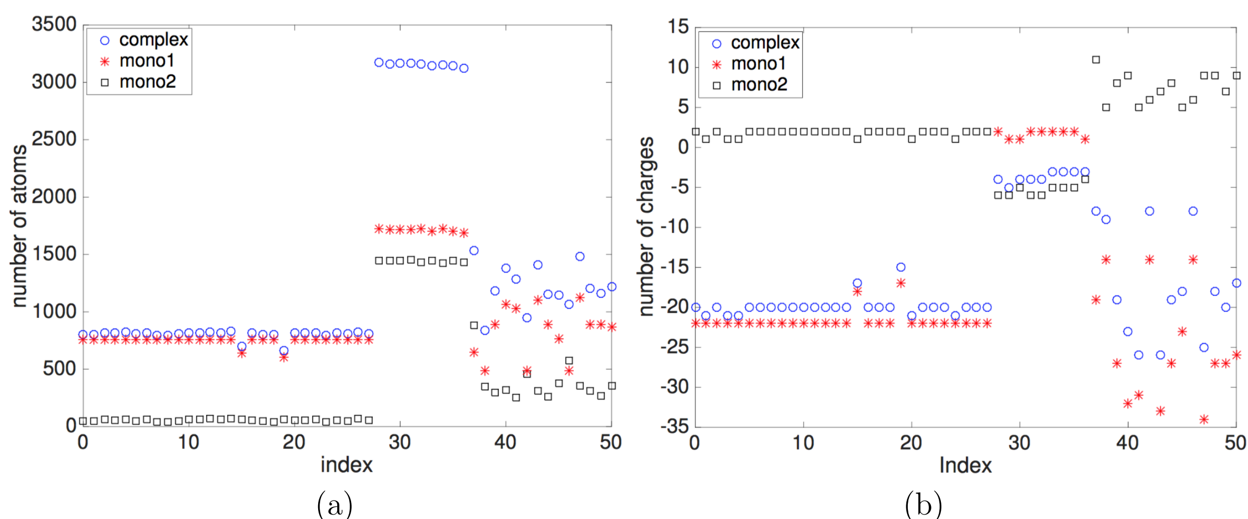
Note that the vacuum electrostatic energy ΔE_{coul} is computed by direct summation of Coulomb potentials and is thus free of numerical errors beyond roundoff error, while the solvation energy $\Delta\Delta G_{\text{solv}}$ is computed by the numerical PB solver, so the latter determines the accuracy of the computed electrostatic binding free energy $\Delta\Delta G_{\text{bind}}$.

Practically, within TABI-PB, the process of computing $\Delta\Delta G_{\text{solv}}$ is straightforward. Surface mesh generation and TABI-PB computations are run independently on the two monomers, generating electrostatic solvation energy values for both. The same process is then performed on the complex, including generating a new mesh for the complex. The difference between the electrostatic solvation energy of the complex and the sum of the electrostatic solvation energies energies of the independent monomers is the electrostatic binding solvation energy.

4.5.2 Methodology

We investigate three sets of biomolecular complexes for calculating binding energies. Dataset 1 is a collection of DNA-minor groove drug complexes with PDB IDs 102D, 109D, 121D, 127D, 129D, 166D, 195D, 1D30, 1D63, 1D64, 1D86, 1DNE, 1EEL, 1FMG, 1FMS, 1JTL, 1LEX, 1PRP, 227D, 261D, 164D, 289D, 298D, 2DBE, 302D, 311D, 328D, 360D. Dataset 2 includes various wild-type and mutant barnase-barstar complexes with PDB IDs 1B27, 1B2S, 1B2U, 1B3S, 2AZ4, 1X1W, 1X1Y, 1X1U, 1X1X. Dataset 3 includes RNA-peptide complexes of various sizes with PDB IDs 1A1T, 1A4T, 1BIV, 1EXY, 1G70, 1HJI, 1I9F, 1MNB, 1NYB, 1QFQ, 1ULL, 1ZBN, 2A9X, 484D. These standard data sets originate from Dr. Marcia Fenley's group at Florida State University [103, 104]. They are available for download Dr. Guowei Wei's group at Michigan State University,

`http://users.math.msu.edu/users/wei/Data/bindingdata.tar.gz`



(a) (b)

Figure 4.18: Features of the target biomolecules across all three datasets, (a) number of atoms, (b) total net charge.

The number of atoms and net charges of the 51 target biomolecular systems across the three datasets are shown in Fig. 4.18. Dataset 1 is a collection of DNA-minor groove drug complexes, composed of a relatively large DNA segment (mono1) and a much smaller drug molecule (mono2). The DNA segment is very negatively charged ($\approx -20e_c$) and the drug molecule is neutral. Dataset 2 includes various wild-type and mutant barnase-barstar complexes. The barnase (mono1) and barstar (mono2) are about the same size, but barnase is slightly positively charged and barstar is more negatively charged Dataset 3 includes RNA-peptide complexes. For most of the set, RNA (mono1) is bigger than the peptide

(mono2), but relatively the same size in magnitude RNA is very negatively charged while peptide is relatively positively charged. These statistics give us valuable information for interpreting binding energy results as reported later.

We investigate binding energies using both MSMS and NanoShaper to generate surface meshes in TABI-PB. To produce a comparable result, we choose NanoShaper scaling values at 1.76, 2.47, 3.49, 4.94, 7.13, which generate surfaces of roughly the same number of triangles as MSMS with density values 5, 10, 20, 40, 80, respectively. We compute limiting values for NanoShaper with linear extrapolation versus triangles^{-1}, both for densities 1.76, 2.47 (corresponding to MSMS extrapolation from $d = 5, 10$), and for densities 2.47, 3.49 (MSMS $d = 10, 20$). An approximate relation between the tested density values are given in Table 4.6, and the rest of this document will refer to the densities by the corresponding d_i for both MSMS and NanoShaper.

Table 4.6: Corresponding densities for MSMS and NanoShaper.

	MSMS	NanoShaper
d_1	5	1.76
d_2	10	2.47
d_3	20	3.49
d_4	40	4.94
d_5	80	7.13

In our calculations, the temperature is $T = 298$K, the dielectric constant is $\varepsilon_1 = 1$ in the solute and $\varepsilon_2 = 80$ in the solvent, and the ionic strength is 0.1M NaCl. The treecode used Taylor expansion order $p = 3$, MAC parameter $\theta = 0.5$, and maximum number of particles in a leaf $N_0 = 500$; these values ensure that the treecode approximation error is smaller than the discretization error [26]. The units for both solvation energy and binding energy are kcal/mol for all the tabular data and figures.

All computations were performed in serial on the University of Michigan FLUX cluster, using 2.8 GHz Intel Xeon E5-2680v2 processors. This investigation was performed using the original Fortran version of TABI-PB. The code was compiled with the GCC Fortran compiler using the `-O2` optimization flag.

The electrostatic binding solvation energy using Eq. 4.14b for the 51 complexes and monomers was computed using TABI-PB and the computational setup described above. Additionally, benchmarking data was generated using the MIBPB Poisson–Boltzmann solver [105] with high resolution mesh $h = 0.2$. Extrapolations of binding solvation energy were calculated before adding in the vacuum electrostatic energy component E_{elec}. The binding

energy values compared below between TABI-PB and MIBPB are all total electrostatic binding free energy values.

4.5.3 Results

4.5.3.1 Comparison between TABI-PB and MIBPB

In this section, we compare binding energy computed with the two Poisson–Boltzmann solvers TABI-PB [26, 28] and MIBPB [105]. The MIBPB solver [106] uses a finite-difference method with rigorous treatment of interface jump conditions [107], geometric singularities [108] and charge singularities [109]. The calculation of binding energies with MIBPB using the same set of biomolecules [105] serves as the benchmark for evaluating the accuracy of the TABI-PB solver. We provide results on various densities using both MSMS and NanoShaper. These results not only validate TABI-PB and the extrapolation scheme, but also provide important evidence in supporting our choice of NanoShaper over MSMS in molecular surface triangulation.

Table 4.7: Top section: average % deviation of TABI-PB calculated binding energy for all three test sets from MIBPB-calculated binding energy. Bottom section: total CPU time (in kiloseconds); TABI-PB comparison is given for NanoShaper (NS) in densities $d_1 - d_5$ and extrapolated in $d_{1/2}$ and $d_{2/3}$, and for MSMS in densities $d_1 - d_3$ and extrapolated in $d_{1/2}$ and $d_{2/3}$. Simulations ran in serial on Intel Xeon CPU.

	NanoShaper							MSMS				
Set	$d_{1/2}$	$d_{2/3}$	d_1	d_2	d_3	d_4	d_5	$d_{1/2}$	$d_{2/3}$	d_1	d_2	d_3
average % deviation from MIBPB binding energy												
Set 1	8.1	6.5	84.1	44.7	25.0	15.0	10.2	12.8	11.1	36.8	19.6	14.6
Set 2	5.6	6.6	34.6	18.6	12.3	8.4	7.9	34.6	38.0	52.5	44.0	40.9
Set 3	8.7	6.7	71.1	39.1	22.3	13.4	10.0	13.5	17.0	56.6	34.8	25.4
Total	7.8	6.6	71.8	38.6	22.0	13.4	9.8	16.9	17.4	45.0	28.1	22.2
total CPU time (ks)												
Set 1	10.8	25.0	3.3	7.5	17.4	37.6	87.1	13.4	32.2	4.4	9.0	23.1
Set 2	15.9	40.5	4.7	11.2	29.3	69.2	177.0	31.0	75.1	7.8	23.3	51.8
Set 3	11.6	26.9	3.5	8.1	18.8	40.8	96.3	19.3	46.1	4.9	14.3	31.8
Total	38.4	92.4	11.6	26.8	65.6	147.7	360.3	63.7	153.4	17.0	46.6	106.8

Table 4.7 shows the average % deviation, computed as average of all % differences, of TABI-PB binding energy results from MIBPB binding energy results as well as the total CPU time for each type of TABI-PB result across all test sets. The TABI-PB binding energy results include both MSMS and NanoShaper at different densities and extrapolated. The

total time is the sum of TABI-PB CPU time for the complex, monomer 1, and monomer 2 for all test cases. For extrapolated results, the total time includes both density levels used for the extrapolation. From the table, we can see that, for both NanoShaper and MSMS, the extrapolated results using TABI-PB are closer to the benchmark MIBPB results than results just using TABI-PB at different densities. TABI-PB results with NanoShaper are, on average, closer across all three sets to the MIBPB result than the TABI-PB results with MSMS are. Additionally, at each density level, TABI-PB with NanoShaper is faster than TABI-PB with MSMS. The CPU time for extrapolated results using two lower densities calculation is less than one higher density calculation. For example, using NanoShaper, the total CPU time for the $d_{1/2}$ result is 38.4ks, with 7.8% deviation, while the total CPU time for the d_3, d_4, d_5 results are 65.6, 147.7, 360.3ks with 22%, 13.4%, 9.8% deviation, respectively. Using extrapolation improves both accuracy and efficiency.

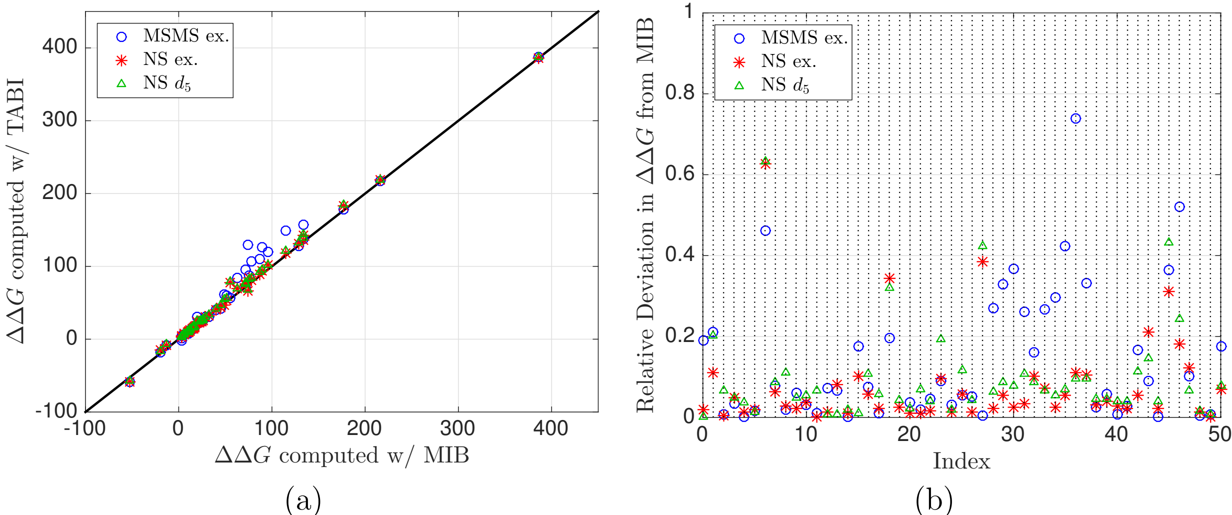


Figure 4.19: (a) MIBPB-calculated binding energy versus TABI-PB-calculated binding energy, (b) relative deviation in TABI-PB-calculated binding energy from MIBPB-calculated binding energy for entire test set, for MSMS and NanoShaper $d_{1/2}$ extrapolations and NanoShaper d_5 results.

Figure 4.19(a) compares the $d_{1/2}$ extrapolated TABI-PB binding energy results using both NanoShaper and MSMS, as well as the NanoShaper d_5 (highest density) results, to MIBPB results for all individual test cases. Extrapolated results and NanoShaper d_5 results using TABI-PB are in line with the MIBPB results. The TABI-PB with NanoShaper results better match the MIBPB results than the TABI-PB with MSMS results.

Figure 4.19(b) displays the relative deviation in binding energy of TABI-PB results from MIBPB results. TABI-PB results use the low density ($d_{1/2}$) NanoShaper and MSMS extrapolations and the high density NanoShaper d_5. We observe that the MSMS extrapolations

are more likely to show a high amount of deviation from the MIBPB results than either the high density NanoShaper or extrapolated NanoShaper results, particularly for test cases in the second test set.

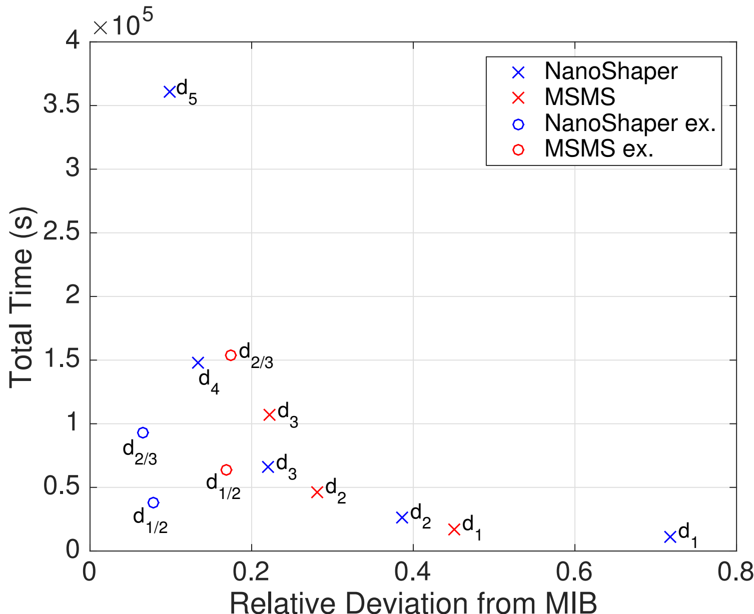


Figure 4.20: Average relative deviation of TABI-PB results from MIBPB results versus total CPU time (s) for computing entire test set, numbers next to data points are corresponding densities. Simulations ran in serial on Intel Xeon CPU.

Additionally, Fig. 4.20 depicts the average deviation of TABI-PB from MIBPB results versus the total time to compute binding energy across all test cases for both NanoShaper and MSMS extrapolated results, as well as NanoShaper d_1, d_2, d_3, d_4, d_5 and MSMS d_1, d_2, d_3 results. As noted above, the total time is the sum of TABI-PB CPU time for the complex, monomer 1, and monomer 2 for all test cases. For extrapolated results, the total time includes both density levels used for the extrapolation. The NanoShaper extrapolations clearly provide the best performance. The NanoShaper $d_{1/2}$ extrapolation, which is our choice, is only slightly less accurate than the best-performed NanoShaper $d_{2/3}$ extrapolation, but uses significantly less CPU time than all but three data points: the two data points from which the extrapolation was performed, and the lowest density MSMS result.

4.5.3.2 Accuracy sensitivity of binding energy and solvation energy

We next investigate the sensitivity of computing solvation energy and binding energy using TABI-PB. The extrapolated values $d_{1/2}$ using NanoShaper are used as reference values to calculate absolute and relative errors which are presented in Fig. 4.21. In particular,

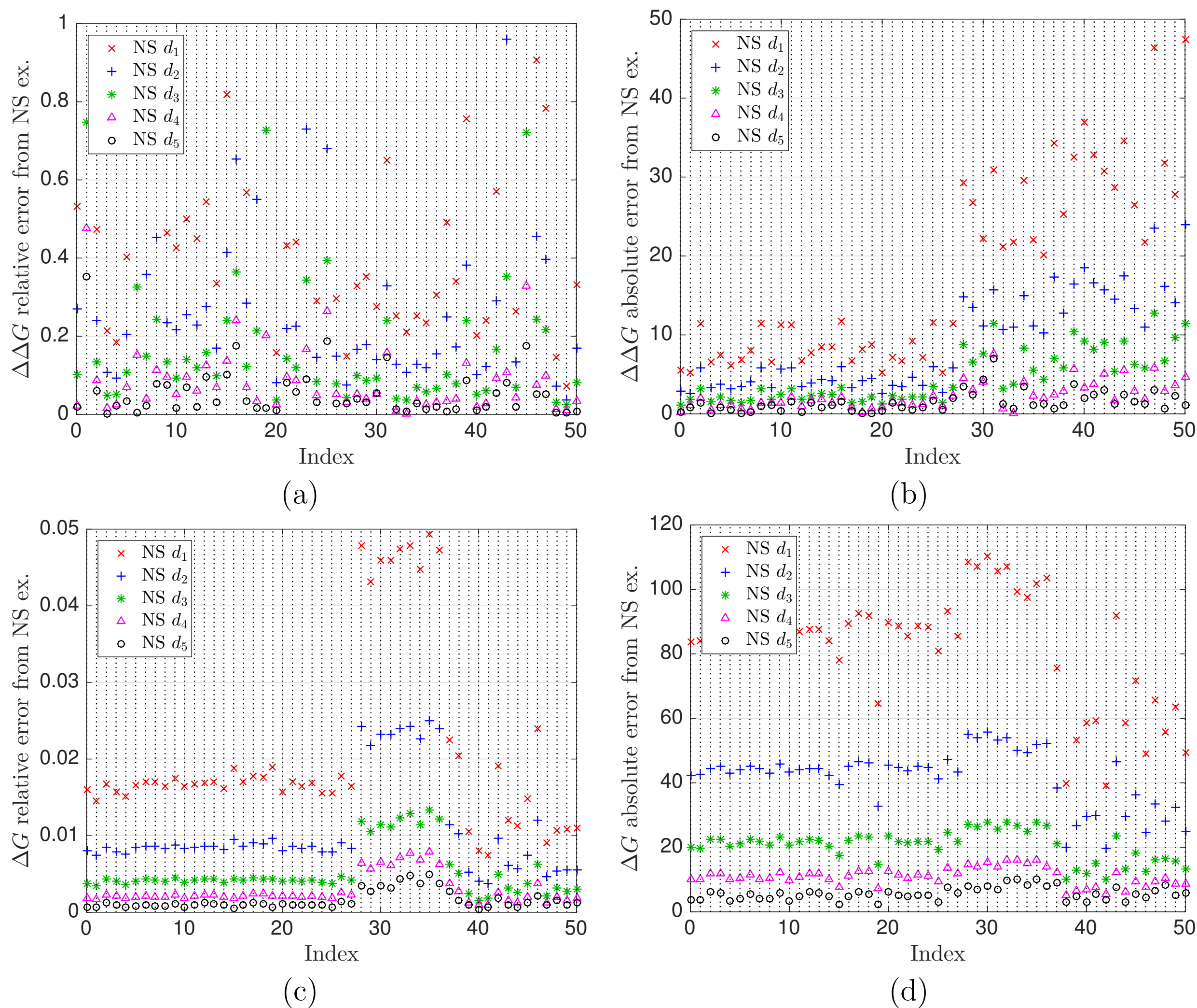


Figure 4.21: (a) Relative error and (b) absolute error in kcal/mol of NanoShaper binding energy in comparison to NanoShaper $d_{1/2}$ extrapolation, (c) relative error and (d) absolute error in kcal/mol of NanoShaper complex solvation energy in comparison to NanoShaper $d_{1/2}$ extrapolation.

Fig. 4.21(a,b) presents the relative error and absolute error in kcal/mol in binding energy and Fig. 4.21(c,d) presents solvation energy errors of the complex. We see that, for a given value of the triangulation density d, the relative error in binding solvation energy $\Delta\Delta G_{\text{bind}}$ is larger than the relative error in solvation energy ΔG_{solv} as seen in Fig. 4.21(a,c). For example, at density $d = 5$, the relative error in binding energy can be as high as nearly 40% as shown in Fig. 4.21(a) while the error in solvation energy is less 1% as shown in Fig. 4.21(c). We observe that a smaller value of binding energy corresponds to a larger relative error. These results show that the binding energy $\Delta\Delta G_{\text{bind}}$ is more sensitive to the accuracy of the PB solver than the solvation energy ΔG_{solv}, and hence, higher numerical resolution is necessary for

accurate binding energy calculations. This difficulty, however, is resolved by our extrapolation schemes. We also observe that the relative errors converge to zero as the density increases, indicating that the extrapolation method is an effective approach for computing accurate binding energy values.

CHAPTER 5

Treecode Acceleration of the 3D Reference Interaction Site Model

This chapter addresses the use of fast summation methods to accelerate the 3D-RISM biomolecular solvation model, particularly long-range asymptotic (LRA) functions which take the form of Coulomb-like potentials. §5.1 summarizes the background of liquid integral equation theory and the 3D-RISM model. The results concerning 3D-RISM in this thesis consist of two primary projects. The primary project related to this topic is detailed in §5.2, which develops cluster-particle treecodes with Taylor series recurrence relations for evaluating the Coulomb potential and the Coulomb-like LRA functions within 3D-RISM. The previous approach used direct sum calculations that scaled as $O\left(N_{\mathrm{grid}}N_{\mathrm{atom}}\right)$ and were a major impediment to studying large proteins and protein complexes. By implementing the numerical methods demonstrated here, we have reduced the computational complexity to at most $O((N_{\mathrm{grid}} + N_{\mathrm{atom}})\log N_{\mathrm{grid}})$ for almost all parts of the calculation. The second project in §5.3 describes the development of GPU-accelerated cluster-particle barycentric Lagrange treecodes (CP-BLTC) based on BaryTree to evaluate these functions, contributing to the future goal of a fully GPU-accelerated 3D-RISM. Further implementation details for this project are presented in Appendix B. The current version of 3D-RISM, which implements the work described in the first project, is available in AmberTools20 and Amber20 at `ambermd.org`. This work was initiated as a collaboration with Tyler Luchko, California State University, Northridge, during a 2016 meeting at the Ohio State University Molecular Biosciences Institute. Except for §5.3, the content of this chapter largely follows the work of [38], which is in preparation.

5.1 Background

5.1.1 Overview

Solvation thermodynamics and the structure of the surrounding liquid play an important role in determining the properties and interactions of molecular systems in solution. While explicit solvent approaches are commonly used, they can be computationally expensive

and require elaborate protocols to calculate different physical properties where solvation is involved, such as solvation free energies [110], preferential interaction parameters [111], and binding free energy [112]. Various implicit solvent methods have been developed to simplify and accelerate the treatment of solvent, including the Poisson–Boltzmann model detailed in Chapter 4. Another approach are integral equation theories, based on the Ornstein-Zernike equation [30], and closely related classical density functional theories [113, 114, 115] as they are complete theories, calculating approximate equilibrium distributions of explicit models, from which all solvation thermodynamics can be computed. The 3D-reference interaction site model of molecular solvation (3D-RISM) [31, 32] is one such integral equation, which has been coupled with classical and quantum mechanics modeling software [116, 117, 118, 119, 120] and shown to provide solvation thermodynamics in good agreement with experiment and explicit solvent calculations [33, 34, 35, 36].

However, 3D-RISM can be computationally expensive, especially for large molecules. 3D-RISM calculations consist of three sequential steps: initialization (calculating potential energy and long-range electrostatic interactions on a 3D grid), iteration to convergence, and integration of the solvent distribution to calculate thermodynamics. For small molecules, iteration time dominates the calculation, which scales with the number of grid points, N_{grid}, as $O\left(N_{\mathrm{grid}} \log N_{\mathrm{grid}}\right)$. Initialization time dominates for typical proteins, scaling with both the number of solute atoms, N_{atom}, and grid points as $O\left(N_{\mathrm{atom}} N_{\mathrm{grid}}\right)$. Integrating solvent thermodynamics is typically 1% or less of the total computation time. Depending on the precision of the calculation, initialization becomes the most expensive part of the calculation for solutes of 1000 atoms or more and is a major barrier to the practical application of 3D-RISM to large molecules.

Limited work has been done to address the computational cost of initialization for open boundary conditions. Because there is no periodic structure, the entire potential energy is calculated for a real-space grid. In addition, to capture contributions beyond the size of the solvent box, analytic long-range asymptotic (LRA) expressions of the solvent correlation functions must also be computed in real- and reciprocal-space. So far, little has been done to address the cost of computing these expressions.

5.1.2 Liquid integral equation theory

The Reference Interaction Site Model (RISM) is a statistical mechanics-based framework for modeling the structure of ionic liquids beyond the simplest mean field approach of the Poisson–Boltzmann equation [31, 121]. We provide here a general overview of the features of liquid integral equation theory necessary to understand RISM. Of central importance to the RISM formalism is the pair distribution function $g(\mathbf{r}_1, \mathbf{\Omega}_1, \mathbf{r}_2, \mathbf{\Omega}_2)$, which is a measure of the

probability density of finding a particle at $\mathbf{r}_2$ with orientation $\mathbf{\Omega}_1$ with respect to a particle at $\mathbf{r}_1$ with orientation $\mathbf{\Omega}_2$. Note that $g(\mathbf{r}_1, \mathbf{\Omega}_1, \mathbf{r}_2, \mathbf{\Omega}_2) > 1$ and $g(\mathbf{r}_1, \mathbf{\Omega}_1, \mathbf{r}_2, \mathbf{\Omega}_2) < 1$ represent areas of relative density enhancement or depletion, respectively, relative to the average density of particle 2 in bulk, with respect to particle 1. Note that, when orientationally averaged so that g is no longer dependent on $\mathbf{\Omega}_1$ or $\mathbf{\Omega}_2$, the pair distribution function is called the radial distribution function. Figure 5.1(a) depicts example radial distribution functions for water, showing RDFs for all three site-site interactions, O–O, O–H, and H–H.

For a homogeneous fluid, the 2-particle number density $\rho^{(2)}$, i.e., the number density of a particle 2 with respect to a particle 1, is given by

$$\rho^{(2)}(\mathbf{r}_1, \mathbf{\Omega}_1, \mathbf{r}_2, \mathbf{\Omega}_2) = \rho_1 \rho_2 g(\mathbf{r}_1, \mathbf{\Omega}_1, \mathbf{r}_2, \mathbf{\Omega}_2), \tag{5.1}$$

where ρ_1 and ρ_2 are the bulk number densities of particles 1 and 2. We define the total correlation function, or TCF, $h(\mathbf{r}_1, \mathbf{\Omega}_1, \mathbf{r}_2, \mathbf{\Omega}_2)$ by the relation

$$1 + h(\mathbf{r}_1, \mathbf{\Omega}_1, \mathbf{r}_2, \mathbf{\Omega}_2) = g(\mathbf{r}_1, \mathbf{\Omega}_1, \mathbf{r}_2, \mathbf{\Omega}_2). \tag{5.2}$$

Thus, the TCF represents the normalized density deviations from bulk for particle 2 with respect to particle 1. The TCF can be partitioned into the direct interaction between particles 1 and 2 and all interactions mediated by particles in the surrounding environment. This partitioning for a homogenous multi-component liquid is given by the Ornstein–Zernike (OZ) relation [122]

$$\begin{aligned} h_{ij}(\mathbf{r}_1, \mathbf{\Omega}_1, \mathbf{r}_2, \mathbf{\Omega}_2) &= c_{ij}(\mathbf{r}_1, \mathbf{\Omega}_1, \mathbf{r}_2, \mathbf{\Omega}_2) \\ &\quad + \sum_k \rho_k \int c_{ik}(\mathbf{r}_1, \mathbf{\Omega}_1, \mathbf{r}_3, \mathbf{\Omega}_3)\ h_{ij}(\mathbf{r}_3, \mathbf{\Omega}_3, \mathbf{r}_2, \mathbf{\Omega}_2)\ d\mathbf{r}_3\ d\mathbf{\Omega}_3 \end{aligned} \tag{5.3}$$

where i, j, k denote molecular species, the integration is performed over all space and orientations, and $c_{ij}(\mathbf{r}_1, \mathbf{\Omega}_1, \mathbf{r}_2, \mathbf{\Omega}_2)$ is the direct correlation function (DCF). From this relation, we can interpret the relative density of particle 2 of species j with respect to particle 1 of species i as the sum of a direct interaction between particles 1 and 2, given by the DCF, and indirect interactions mediated by surrounding solvent of all molecular species. Note that h and c are unknown functions; to determine the TCF and DCF, another relation between h and c, known as the closure relation, must be specified. The most general expression for the closure relation is

$$g_{ij} = \exp(-\beta w_{ij}) = \exp(-\beta u_{ij} + h_{ij} - c_{ij} + b_{ij}) \tag{5.4}$$

where the function arguments $(\mathbf{r}_1, \mathbf{\Omega}_1, \mathbf{r}_2, \mathbf{\Omega}_2)$ have been suppressed. w_{ij} is the total interaction potential between particles 1 and 2 of species i and j, respectively. $\beta = 1/k_b T$ where k_b is the Boltzmann constant and T is the temperature. u_{ij} is the pair interaction potential, and b_{ij} is known as the bridge function. We note again that $g_{ij} = 1 + h_{ij}$. The first equality is a result of statistical mechanics that is outside the scope of this background section. The second equality is a result of density functional and graph theory that is also outside the scope of this background. The bridge function is not known exactly in a closed form, and thus must be subject to an approximation. One particularly simple closure approximation is the hyper-netted chain equation, or HNC, in which the bridge function is set equal to zero. A discussion of the wide variety of closure approximations is outside the current scope.

5.1.3 1D reference interaction site model (1D-RISM)

We can consider a molecule to be composed of a number of "sites" that interact in a pairwise fashion. Typically, we consider the sites to be the constituent atoms. In this manner, we can apply Eq. 5.3 to non-spherical molecules. To make application of the OZ equation practical, the RISM formalism considers the sites to be rigid and orientationally averages out all site-site correlations. This leads to a one dimensional expression for the DCF between two molecules 1 and 2,

$$c(\mathbf{r}_1, \mathbf{\Omega}_1, \mathbf{r}_2, \mathbf{\Omega}_2) = c(\mathbf{r}_1, \mathbf{r}_2) = \sum_{\alpha_1, \gamma_2} c_{\alpha_1 \gamma_2}\left(|\mathbf{r}_1 - \mathbf{r}_2|\right), \tag{5.5}$$

where α_1 and γ_2 denote the interaction sites on molecules 1 and 2, respectively. Thus, the DCF is a function of only the distances between the intermolecular pairs of interaction sites. Using this expression for the intermolecular DCF, Eq. 5.3 can be rewritten as

$$\rho_\alpha h_{\alpha\gamma}(r)\rho_\gamma = \sum_{\lambda,\beta} \omega_{\alpha\lambda}(r) * c_{\lambda\beta}(r) * \omega_{\beta\lambda}(r) + \sum_{\lambda,\beta} \omega_{\alpha\lambda}(r) * c_{\lambda\beta}(r) * \rho_\beta h_{\beta\gamma}(r)\rho_\gamma \tag{5.6}$$

where $*$ is the convolution operator, the sums are performed over all interaction sites on all molecular species, and ω denotes factors that account for the molecular shape. ω is defined in reciprocal space by

$$\hat{\omega}_{\alpha\gamma}(k) = \delta_{\alpha\gamma} + (1 - \delta_{\alpha\gamma}) \frac{\sin(k l_{\alpha\gamma})}{k l_{\alpha\gamma}} \tag{5.7}$$

where δ is the Kronecker delta function and $l_{\alpha\gamma}$ is the real-space distance between sites on the same species. For $\alpha = \gamma$, $\hat{\omega}_{\alpha\alpha} = 1$, and for sites α and γ on different species, $\hat{\omega}_{\alpha\gamma} = 0$.

Equation 5.6 is typically written in the matrix form

$$\begin{aligned}\boldsymbol{\rho h \rho} &= \boldsymbol{\omega} * \boldsymbol{c} * \boldsymbol{\omega} + \boldsymbol{\omega} * \boldsymbol{c} * \boldsymbol{\rho h \rho} \\ &= \left(1 - \boldsymbol{\omega} * \boldsymbol{c}\right)^{-1} \boldsymbol{\omega} * \boldsymbol{c} * \boldsymbol{\omega}.\end{aligned} \tag{5.8}$$

In this form, all bolded quantities are $N_{site} \times N_{site}$ matrices, where N_{site} is the number of interaction sites across all molecular species. $\boldsymbol{\rho}$ is a diagonal matrix whose entries are related to bulk number densities of molecular species present, $\boldsymbol{c}$ contains the site-site direct correlation functions, and $\boldsymbol{\omega}$ is the intramolecular correlation matrix which accounts for molecular geometry.

5.1.4 3D reference interaction site model (3D-RISM)

In the context of a large solute macromolecule composed of many sites, the previous formalism must be modified to allow for distribution functions that are not radially symmetric. In the case of a solute macromolecule in solvent, the distribution functions of the solute U can be treated in full 3D while the distribution functions of the solvent V can be treated as radially symmetric, as in the 1D-RISM formalism. The 3D-RISM OZ equations then become

$$\begin{aligned}h_{ij}^{VV}(\mathbf{r}_i, \mathbf{r}_j) &= c_{ij}^{VV}(\mathbf{r}_i, \mathbf{r}_j) + \sum_k \rho_k^V \int c_{ik}^{VV}(\mathbf{r}_i, \mathbf{r}_k)\ h_{kj}^{VV}(\mathbf{r}_k, \mathbf{r}_j)\ d\mathbf{r}_k \\ h_i^{UV}(\mathbf{r}_1, \boldsymbol{\Omega}_1, \mathbf{r}_j) &= c_i^{UV}(\mathbf{r}_1, \boldsymbol{\Omega}_1, \mathbf{r}_i) + \sum_k \rho_k^V \int c_k^{UV}(\mathbf{r}_1, \boldsymbol{\Omega}_1, \mathbf{r}_k)\ h_{kj}^{VV}(\mathbf{r}_k, \mathbf{r}_j)\ d\mathbf{r}_k\end{aligned} \tag{5.9}$$

where h_{ij}^{VV} is the radially symmetric TCF containing only solvent interactions, and h_i^{UV} is the TCF of the solvent with respect to the solute. 1 denotes the solute, and i, j are orientationally averaged solvent sites. The first equation is used to obtain the TCF of the bulk solvent, which is then used in the next equation to obtain the TCF of the solvent around the solute. Figure 5.1(b) depicts sample radial distribution functions g^{UV} showing relative densities of water solvent sites V about a solute molecule U.

5.1.5 Long range components of correlation functions

When solving the RISM equations, the OZ relation given by Eqs. 5.8 and 5.9 is solved for the TCF by a Fast Fourier Transform, which turns the convolution integrals into multiplications. However, the long range components of the DCF and TCF present serious computational difficulties. Previously, renormalization procedures have been used in the XRISM formulation to handle these issues. More recently, the long range components have been handled analytically, by computing the asymptotic component of the correlation func-

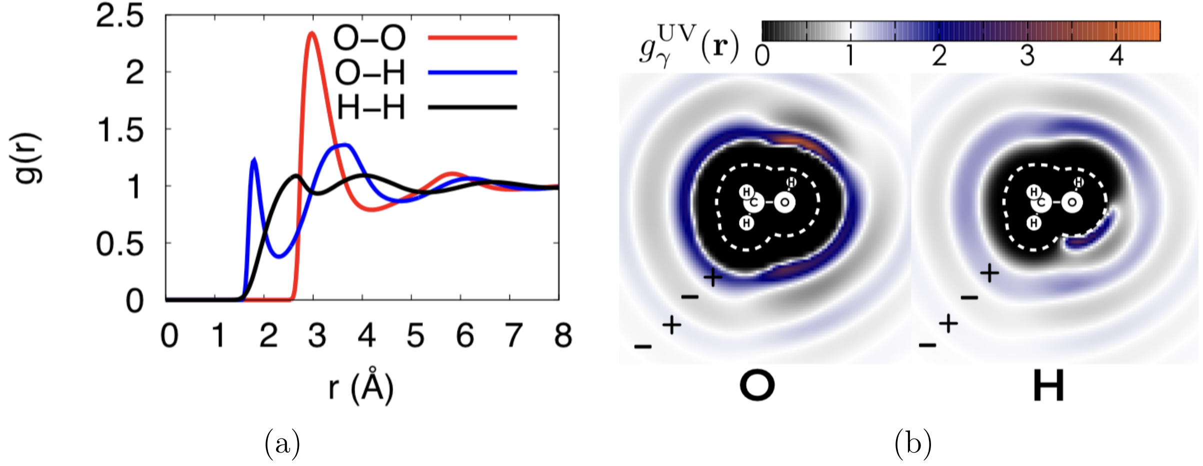


Figure 5.1: (a) Radial distribution functions (RDF) of water, depicting all three site-site interactions, O–O, O–H, and H–H. (b) Cross section of solute site-solvent site RDFs of a hydroxymethyl group surrounded by water, showing relative density of oxygen and hydrogen molecules in solution around the hydroxymethyl group. Figures courtesy of Tyler Luchko.

tions, removing them from correlation functions before an FFT or inverse FFT is performed, and adding them back after the transform. The expressions [123] for the asymptotic TCF and DCF in real and reciprocal space are

$$c_\gamma^{(\mathrm{lr})}(\mathbf{r}) = -\frac{1}{k_b T}\sum_a \frac{Q_a^U q_\gamma}{|\mathbf{r}-\mathbf{R}_a|}\operatorname{erf}\left(\frac{|\mathbf{r}-\mathbf{R}_a|}{\eta}\right) \tag{5.10}$$

$$\hat{c}_\gamma^{(\mathrm{lr})}(\mathbf{k}) = -\frac{4\pi}{k_b T}\sum_a \frac{Q_a^U q_\gamma}{k^2}\exp\left(-\frac{k^2\eta^2}{4} + i\mathbf{k}\cdot\mathbf{R}_a\right) \tag{5.11}$$

$$\begin{aligned} h_j^{(\mathrm{lr})}(\mathbf{r}) = &-\frac{1}{2\varepsilon k_b T}\sum_a \frac{Q_a^U q_\gamma}{|\mathbf{r}-\mathbf{R}_a|}\exp\left(\frac{\kappa_D^2\eta^2}{4}\right) \\ &\left[\exp\left(-\kappa_D|\mathbf{r}-\mathbf{R}_a|\right)\operatorname{erfc}\left(\frac{\kappa_D\eta}{2} - \frac{|\mathbf{r}-\mathbf{R}_a|}{\eta}\right)\right. \\ &\left.-\exp\left(\kappa_D|\mathbf{r}-\mathbf{R}_a|\right)\operatorname{erfc}\left(\frac{\kappa_D\eta}{2} + \frac{|\mathbf{r}-\mathbf{R}_a|}{\eta}\right)\right] \end{aligned} \tag{5.12}$$

$$\hat{h}_j^{(\mathrm{lr})}(\mathbf{k}) = -\frac{4\pi}{\varepsilon k_b T}\sum_a \frac{Q_a^U q_j}{k^2}\exp\left(-\frac{k^2\eta^2+\kappa_D^2}{4} + i\mathbf{k}\cdot\mathbf{R}_a\right) \tag{5.13}$$

where a are the solute sites with position $\mathbf{R}_a$ and partial charge Q_a^U and $\mathbf{r}$. k_b is the Boltzmann constant, T is temperature, ε is the dielectric constant, η is a charge smearing parameter, κ_D is the contribution to the inverse Debye length of ionic species j, and q_γ is the partial charge on solvent site γ.

5.1.6 The RISM procedure and its implementation

Given a form for the OZ relation and a closure relation, the equations are solved in an iterative manner to determine the TCF and DCF. The equations are solved on a regular grid, with grid spacings of approximately 0.25-0.5 Å, and an enclosing box that extends 20–60 Å beyond the biomolecular solute. There exist, of course, a large variety of methods for solving the coupled equations. The simplest process is Picard iteration, for which the general algorithmic form of solving the RISM equations is given in Algorithm 5.1. Additional approaches include generalized minimum residual, wavelet methods, multigrid methods, and dynamic relaxation.

Algorithm 5.1 An example of the RISM procedure with removal of asymptotic components using Picard iteration.

1: **while** $c_\gamma(\mathbf{r})$ is not converged **do**
2: given $c_\gamma(\mathbf{r})$, subtract out asymptotic component $c_\gamma^{(lr)}(\mathbf{r})$
3: transform $c_\gamma(\mathbf{r})$ to reciprocal space $c_\gamma(\mathbf{k})$
4: add back asymptotic component $c_\gamma^{(lr)}(\mathbf{k})$
5: solve OZ equation for $h_\gamma(\mathbf{k})$
6: subtract out asymptotic component $h_\gamma^{(lr)}(\mathbf{k})$
7: transform $h_\gamma(\mathbf{k})$ to real space $h_\gamma(\mathbf{r})$
8: add back asymptotic component $h_\gamma^{(lr)}(\mathbf{r})$
9: use closure relation to calculate new $c_\gamma(\mathbf{r})$
10: **end while**

One particularly popular implementation of 3D-RISM is available in the Amber molecular modeling suite as part of AmberTools [124, 116, 111]. As part of this package, 3D-RISM can also be used to calculate molecular dynamics [123]. This implementation uses modified direct inversion in the iterative subspace (MDIIS), a technique developed in quantum chemistry, to solve the coupled equations.

Note that, given M grid points on which to evaluate the asymptotic correlation functions, and N source charges, computing the TCF and DCF requires $O(NM)$ operations. For large solutes, this computation becomes prohibitively expensive, so we investigate the implementation of treecode fast summation methods to accurately and efficiently calculate these functions. The M grid points serve as the target sites and the N source charges serve as the source particles. Because M is typically much greater than N, we in particular investigate cluster-particle treecodes. In addition, RISM requires interaction potential at the M grid points from the N source charges. The interaction potential includes a long range Coulomb component for which we also implement a cluster-particle treecode.

In §5.2, we detail the application of Taylor expansion cluster-particle treecodes to accelerated 3D-RISM. In §5.3, we detail the ongoing development of GPU-accelerated treecodes to further accelerate computing TCF long-range asymptotics.

5.2 Project 1: Implementing treecodes for asymptotic correlation functions

5.2.1 Project description

In the evaluation of LRA functions in 3D-RISM, the solute is represented by N source particles and the solvent grid by M target sites. As described in Chapter 2, traditional particle-cluster treecodes build a tree on the source particles, with a computational cost that scales as $O(M \log N)$. For $M \gg N$, as is typically the case for the 3D-RISM solvent grid, these methods scale poorly. In this case, it is advantageous to consider an alternative cluster-particle treecode in which the tree is built on the targets, with a computational cost that scales as $O(N \log M)$ [37]. The results in this section follows the work of [38]; this paper also contains work on cut-off methods for other parts of the 3D-RISM calculation, but discussion of these methods is omitted here.

5.2.1.1 Direct correlation function

Writing the asymptotic direct correlation function from Eq. 5.10 in the cluster-particle form shown in Eq. 2.9 yields,

$$c_\gamma^{(\mathrm{lr})}(\mathbf{r}_i) = \frac{-q_\gamma}{k_b T}\left[\sum_{\mathbf{R}_a \in D} Q_a^U \phi_c^{(\mathrm{lr})}(\mathbf{r}_i, \mathbf{R}_a) + \sum_{l=1}^{L} \sum_{\mathbf{R}_a \in I_l} Q_a^U \phi_c^{(\mathrm{lr})}(\mathbf{r}_i, \mathbf{R}_a)\right], \tag{5.14}$$

where the DCF interaction potential is

$$\phi_c^{(\mathrm{lr})}(\mathbf{r}_i, \mathbf{R}_a) = \frac{1}{|\mathbf{r}_i - \mathbf{R}_a|} \operatorname{erf}\left(\frac{|\mathbf{r}_i - \mathbf{R}_a|}{\eta}\right). \tag{5.15}$$

Following [125], the Taylor coefficients of the DCF potential function in Eq. 5.15 are computed by the recurrence,

$$a_{\mathbf{k}}(\mathbf{x}, \mathbf{y}) = \frac{1}{|\mathbf{x} - \mathbf{y}|^2}\left[\left(2 - \frac{1}{\|\mathbf{k}\|}\right) \sum_{i=1}^{3} (x_i - y_i) a_{\mathbf{k}-\mathbf{e}_i} - \left(1 - \frac{1}{\|\mathbf{k}\|}\right) \sum_{i=1}^{3} a_{\mathbf{k}-2\mathbf{e}_i} + b_{\mathbf{k}}\right], \tag{5.16}$$

where the $b_{\mathbf{k}}(\mathbf{x}, \mathbf{y})$ are the Taylor coefficients of an auxiliary Gaussian function, $\exp\left(-\left|\mathbf{x}-\mathbf{y}\right|^2/\eta^2\right)$, whose recurrence is

$$b_{\mathbf{k}}(\mathbf{x}, \mathbf{y}) = \frac{2}{\eta^2 \|\mathbf{k}\|} \times \left(\sum_{i=1}^{3} (x_i - y_i) b_{\mathbf{k}-\mathbf{e}_i} - \sum_{i=1}^{3} b_{\mathbf{k}-2\mathbf{e}_i} \right). \tag{5.17}$$

5.2.1.2 Total correlation function

Similarly, writing the asymptotic total correlation function from Eq. 5.12 in the cluster-particle form shown in Eq. 2.9 yields,

$$h_{\gamma}^{(\mathrm{lr})}(\mathbf{r}_i) = \frac{-q_{\gamma}}{2\varepsilon k_b T} \exp\left(\frac{\kappa_D^2 \eta^2}{4}\right) \left[\sum_{\mathbf{R}_a \in D} Q_a^U \phi_h^{(\mathrm{lr})}(\mathbf{r}_i, \mathbf{R}_a) + \sum_{l=1}^{L} \sum_{\mathbf{R}_a \in I_l} Q_a^U \phi_h^{(\mathrm{lr})}(\mathbf{r}_i, \mathbf{R}_a) \right], \tag{5.18}$$

where the TCF interaction potential is

$$\begin{aligned} \phi_h^{(\mathrm{lr})}(\mathbf{r}_i, \mathbf{R}_a) = \frac{1}{|\mathbf{r}_i - \mathbf{R}_a|} \Bigg[& e^{(-\kappa_D |\mathbf{r}_i - \mathbf{R}_a|)} \operatorname{erfc}\left(\frac{\kappa_D \eta}{2} - \frac{|\mathbf{r}_i - \mathbf{R}_a|}{\eta} \right) \\ & - e^{(\kappa_D |\mathbf{r}_i - \mathbf{R}_a|)} \operatorname{erfc}\left(\frac{\kappa_D \eta}{2} + \frac{|\mathbf{r}_i - \mathbf{R}_a|}{\eta} \right) \Bigg]. \end{aligned} \tag{5.19}$$

The TCF potential function in Eq. 5.19 has a complicated form and computing its Taylor coefficients is a formidable task. Note however that the Taylor expansions are only used when a source particle and target cluster are well-separated, in other words when $|\mathbf{r}_i - \mathbf{R}_a|$ is large, and in that case we can take advantage of the asymptotic properties of the complementary error function. Thus for large values of $|\mathbf{r} - \mathbf{R}_a|$, we have

$$\operatorname{erfc}\left(\frac{\kappa_D \eta}{2} - \frac{|\mathbf{r} - \mathbf{R}_a|}{\eta} \right) \approx 2, \quad \operatorname{erfc}\left(\frac{\kappa_D \eta}{2} + \frac{|\mathbf{r} - \mathbf{R}_a|}{\eta} \right) \approx 0, \tag{5.20}$$

Using this observation, the TCF interaction potential in Eq. 5.19 is approximated by

$$\phi_h^{(\mathrm{lr})}(\mathbf{r}_i, \mathbf{R}_a) \approx \frac{2 \exp\left(-\kappa_D \left|\mathbf{r}_i - \mathbf{R}_a\right|\right)}{|\mathbf{r}_i - \mathbf{R}_a|}. \tag{5.21}$$

Solute	Number of Atoms	Net Charge
Phenol	13	$0e$
Cucurbit[7]uril (CB7)	122	$0e$
Adhiron	1324	$-1e$
Tubulin	13456	$-36e$

Table 5.1: Solutes used in this work.

Functionally, this is nothing more than a screened Coulomb interaction, so following [4], we may use the recurrence relation for its Taylor coefficients given in Eq. 5.22,

$$a_{\mathbf{k}}(\mathbf{x},\mathbf{y}) = \frac{1}{|\mathbf{x}-\mathbf{y}|^2}\Bigg[\left(2-\frac{1}{\|\mathbf{k}\|}\right)\sum_{i=1}^{3}(x_i-y_i)a_{\mathbf{k}-\mathbf{e}_i} - \left(1-\frac{1}{\|\mathbf{k}\|}\right)\sum_{i=1}^{3}a_{\mathbf{k}-2\mathbf{e}_i} + \kappa_D\left(\sum_{i=1}^{3}(x_i-y_i)b_{\mathbf{k}-\mathbf{e}_i} - \sum_{i=1}^{3}b_{\mathbf{k}-2\mathbf{e}_i}\right)\Bigg], \quad (5.22)$$

where the $b_{\mathbf{k}}(\mathbf{x},\mathbf{y})$ are the Taylor coefficients of an auxiliary function, $2\exp\left(-\kappa_D\left|\mathbf{x}-\mathbf{y}\right|\right)$, whose recurrence is

$$b_{\mathbf{k}}(\mathbf{x},\mathbf{y}) = \frac{\kappa_D}{\|\mathbf{k}\|}\left(\sum_{i=1}^{3}(x_i-y_i)a_{\mathbf{k}-\mathbf{e}_i} - \sum_{i=1}^{3}a_{\mathbf{k}-2\mathbf{e}_i}\right). \quad (5.23)$$

5.2.2 Methodology

5.2.2.1 System preparation

Four solutes were selected for benchmarking and testing, giving a range in the number of atoms of over four orders of magnitude, from 13 to 13,456 atoms (see Table 5.1 and Fig. 5.2). For each solute, the `tleap` program in AmberTools 17 [126] was used to assign the final parameters. OpenBabel [127] was used to create the 3D structure of phenol from the SMILES string “c1ccc(cc1)O”. The general Amber force field parameters (GAFF) [128] and AM1-BCC (AM1 with bond charge corrections) charges [129] were assigned using `antechamber`. The 3D structure of cucurbit[7]uril (CB7), a neutral host molecule, was obtained from the Statistical Assessment of the Modeling of Proteins and Ligands 4 (SAMPL4) exercise data set [130]. GAFF parameters were used with charges derived using the pyR.E.D. server [131, 132, 133, 134, 135]. Adhiron (PDB ID: 4N6T) is an engineered scaffold protein [136] and was parameterized using Amber FF14SB [137]. A complete crystal structure of tubulin, the main constituent protein of microtubules, does not exist. We constructed a 3D model from

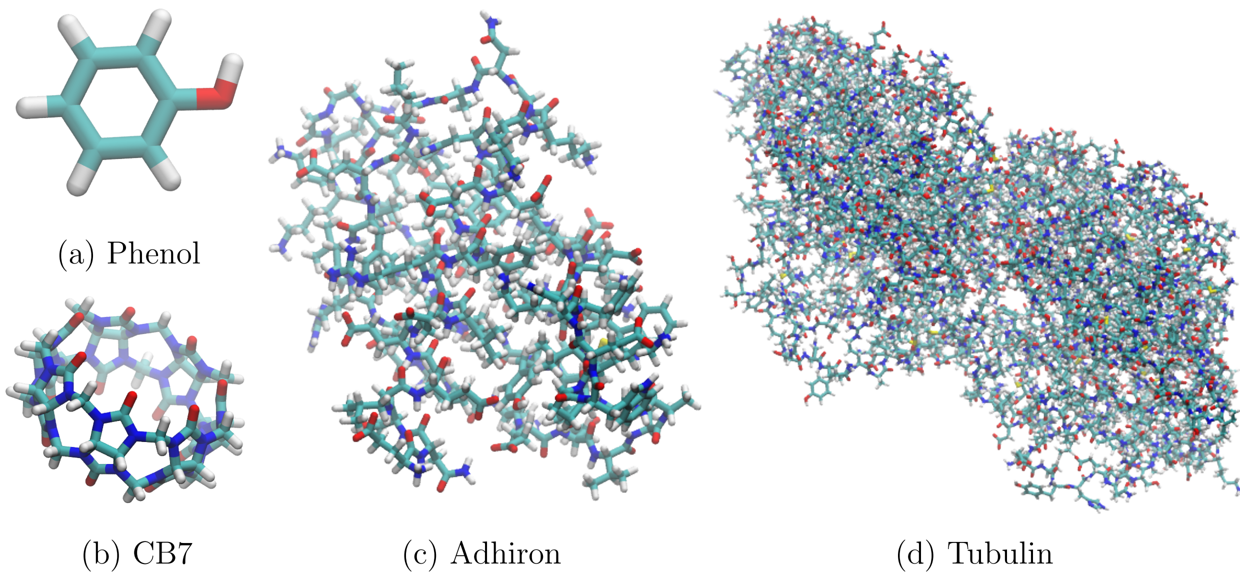


Figure 5.2: Stick representations of solutes used in this work.

PDB IDs 1TVK and 1SA0 [138, 139], using Modeller [140] to combine the structures and fill in residues missing from the H1-B2α-tubulin loop and the α- and β-tubulin N-termini. The C-terminal tails were not present in the crystal structures and replaced with N-methylamide (NME) caps. Amber FF14SB was used for the amino acids, the pyR.E.D. force field for GTP and GDP, and the MG^{2+}parameters for use with SPC/E water from Li et al. [141].

5.2.2.2 3D-RISM calculations

All RISM calculations were performed in AmberTools 19 [39].

The solvent was prepared for 3D-RISM using the `rism1d` program and consisted of 55.2 M modified SPC/E water [142, 116] with 0.1 M NaCl using the corresponding Joung-Cheatham parameters [143]. Dielectrically consistent RISM (DRISM) theory [144] was used with a dielectric constant of 78.44 and the Kovalenko-Hirata (KH) closure [32] at a temperature of 298.15 K. The solution was solved on 65536 grid points with 0.025 Å grid spacing using the default parameters for the modified direct inversion of the iterative subspace (MDIIS) solver [145].

The `rism3d.snglpnt` program was used for all 3D-RISM calculations. Default MDIIS settings, the KH closure, and a 0.5 Å grid spacing were used for all calculations. No cut-off was used for electrostatic interactions. The buffer distance between the solute and the edge of the solvent box was either explicitly set or determined from the requested LJ tolerance. In all cases, `rism3d.snglpnt` automatically increased the buffer distance to ensure that all grid

Solute	Tolerance	TCF		DCF		Coulomb		Reciprocal-Space
		MAC	Order	MAC	Order	MAC	Order	
Tubulin	1E−6	0.3	6	0.3	8	0.3	8	1E−8
Adhiron	1E−6	0.3	2	0.3	6	0.3	6	1E−7
CB7	1E−6	0.3	2	0.3	6	Direct		1E−7
Phenol	1E−6	0.3	2	Direct		Direct		1E−7

Table 5.2: Optimized 3D-RISM parameter settings. Treecode parameters MAC θ, order p. All LJ cutoffs were adjusted to fit inside the solvation box.

dimensions were divisible by factors of 2, 3, 5, and 7, and that the number of y- and z-grid points was divisible by the number of processes.

Performance and accuracy of the treecode summation was tested by performing calculations using direct summation for all calculations or using treecode for only one of DCF, TCF, or Coulomb calculations. The direct sum benchmark calculations use a buffer distance of 24 Å and were converged to a residual tolerance of 1E−13. All other 3D-RISM calculations detailed below were repeated five times to provide average timings. A buffer distance of 24 Å and grid spacing of 0.5 Å were selected as a compromise between precision and computational cost; obtaining a relative numerical error of 1E−10 would require a solvent grid much too large to be considered. When using treecode summation, all combinations of the MAC parameter θ from 0.2 to 0.7 in steps of 0.1, the Taylor series order p from 2 to 20 in steps of 2, and maximum leaf size N_0 values of 60, 500, and 4000 were used. In all cases, the 3D-RISM equations were solved to a residual tolerance of 1E−10. Optimized serial and parallel jobs were run with the settings in Table 5.2. To test the parallel scaling of treecode summation, calculations were performed on 1, 2, 4, 8, 16, 24, 32, 48, 64, 72, and 96 processes for all solutes.

Serial and parallel calculations for phenol, CB7, 4N6T, and tubulin were run on the Linux cluster Metropolis at California State University, Northridge, which has seven nodes connected by QDR Infiniband interconnects, each with 256 GB of memory and two 12 core Intel 2.4 GHz Xeon E5-2600 v2 (“Ivy Bridge-EP”) CPUs. AmberTools was compiled with the Intel Fortran and C++ compilers 19.1.053 and the OpenMPI 3.1.3 MPI library [146]. Additional parallel benchmarking was performed on the Skylake nodes of Stampede2 at the Texas Advanced Computing Center through the Extreme Science and Engineering Discovery Environment (XSEDE) [91], which each have two 24 core Intel Xeon Platinum 8160 CPUs, 192 GB of memory and are connected by a 100 Gb/sec Intel Omni-Path network. In this case, the software was compiled with the Intel Fortran and C++ compilers 17.0.4 and MVAPICH2 2.3 MPI library.

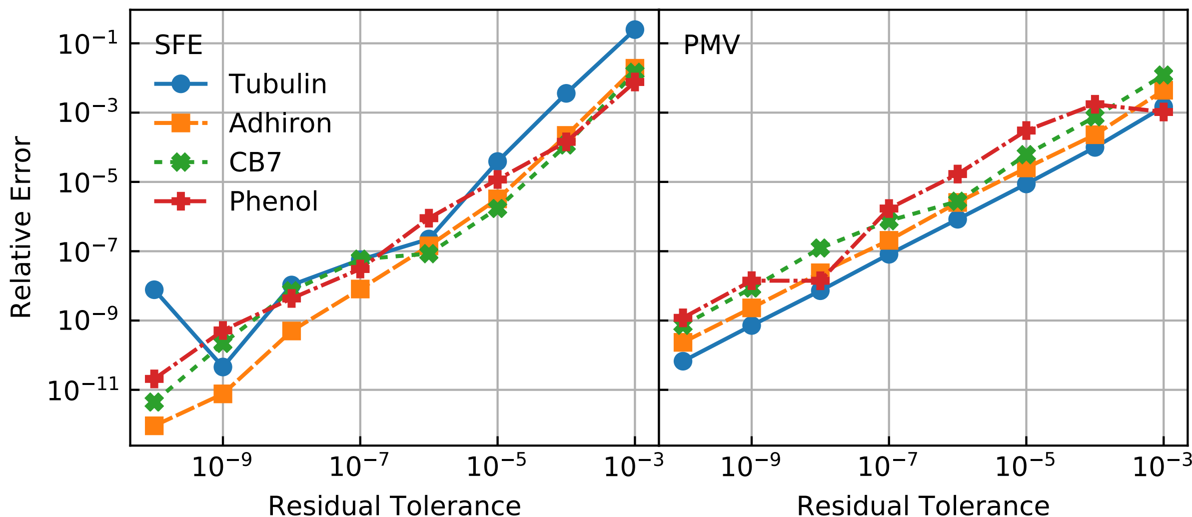


Figure 5.3: Dependence of the relative numerical error of the solvation free energy (SFE) and partial molar volume (PMV) on the 3D-RISM residual tolerance. Relative errors are calculated against a reference calculation converged to a residual tolerance of 1E−13.

5.2.3 Results

5.2.3.1 Numerical precision requirements

Computational efficiencies from treecode summation must not come at the cost of the numerical precision of computed thermodynamic observables. Generally, the solvation free energy (SFE) will be the most important value to be calculated with 3D-RISM. The numerical precision required depends on the application to be considered. For SFE calculations absolute errors up to 0.1 kcal/mol are generally acceptable. An absolute error < 0.1 kcal/mol typically means relative errors as large as 1E−3 for small molecules but may need to be less than 1E−5 or even 1E−6 for large proteins. To ensure stability, molecular dynamics simulations with 3D-RISM require relative errors less than 1E−5 to ensure sufficient agreement between SFEs and their derivatives [116]. Energy minimization is even more demanding, requiring relative errors less than 1E−10.

In practice, the convergence criterion for our iterative solver is to reach a given maximum allowable residual tolerance. Fig. 5.3 shows the relative error of SFE and PMV thermodynamic quantities as the residual tolerance of the 3D-RISM calculation is adjusted. Overall, we find that that residual tolerance and relative error are directly proportional for observables we have considered. In general, we can say that

$$\epsilon_{\mathrm{SFE}} \gtrsim 10 \times \text{tolerance}. \tag{5.24}$$

For the SFE, there is no apparent dependence on the size of the solute, though tubulin has an anomalously large relative error for a residual tolerance of 1E−10. There does appear to be a dependence on the solute size for the PMV, with larger solutes achieving smaller relative errors for the same residual tolerance. The vast majority of 3D-RISM calculations should use a residual tolerance of 1E−5 or 1E−6.

5.2.3.2 Treecode summation

To determine the impact on speed and numerical precision of the treecode parameters θ, p, and N_0 for TCF LRA, DCF LRA, or Coulomb potential energy, SFEs calculated from 3D-RISM for different size solutes with different treecode parameters were compared against direct sum calculations in Figs. 5.4, 5.5 and 5.6. Each data point in the plots represents a different value of p for a given θ, increasing from right to left. Only results for $N_0 = 500$ are shown, as we found that $N_0 = 60$ and $N_0 = 500$ performed almost identically, while $N_0 = 4000$ was generally slower for the same numerical precision. The cluster of data points in the lower left corner of each plot indicates that increasing p does not provide any additional precision. Though there is some noise in the timing, mostly due to interprocess interference, increasing p almost universally reduces the relative error, but also increases execution time. In all cases, a $\theta \leq 0.4$ was sufficient to obtain solutions with the smallest possible error, provided that the number of Taylor series terms was large enough. Results for $\theta = 0.7$ were omitted, as the performance was consistently worse for all calculations. Otherwise, the best choice of parameters depended on the quantity being summed, TCF LRA, DCF LRA, or Coulomb potential energy, and the size of the solute.

Treecode summation shows the largest relative speedups for the TCF LRA. In fact, treecode is faster than direct summation for all solutes at all precisions and is nearly two orders of magnitude faster than direct summation for tubulin and adhiron for relative errors of 1E−5, which is sufficient for most calculations. However, the treecode parameters that give the best performance vary with the relative error and the solute. For tubulin, $\theta = 0.4$ and 0.5 have the best performance, while $\theta = 0.3$ is close. $\theta = 0.3$ provides the best performance for both adhiron and CB7, except for the largest relative errors, where $\theta = 0.4$ and even 0.5 are slightly faster. Even phenol shows speedups relative to direct summation for all θ values with an appropriate p; however, the extreme values of $\theta = 0.2$ and 0.6 have the best performance.

The performance of treecode summation for the DCF LRA is still much better than direct summation for tubulin, adhiron, and CB7 but not for phenol. In contrast to TCF LRA, it is difficult to distinguish between the performance of different MAC values. $\theta = 0.3$, 0.4 and 0.5 have similar performance for tubulin and adhiron over almost the full range of relative errors. However, $\theta = 0.5$ is unable to achieve the lowest relative errors, even for $p = 20$, and

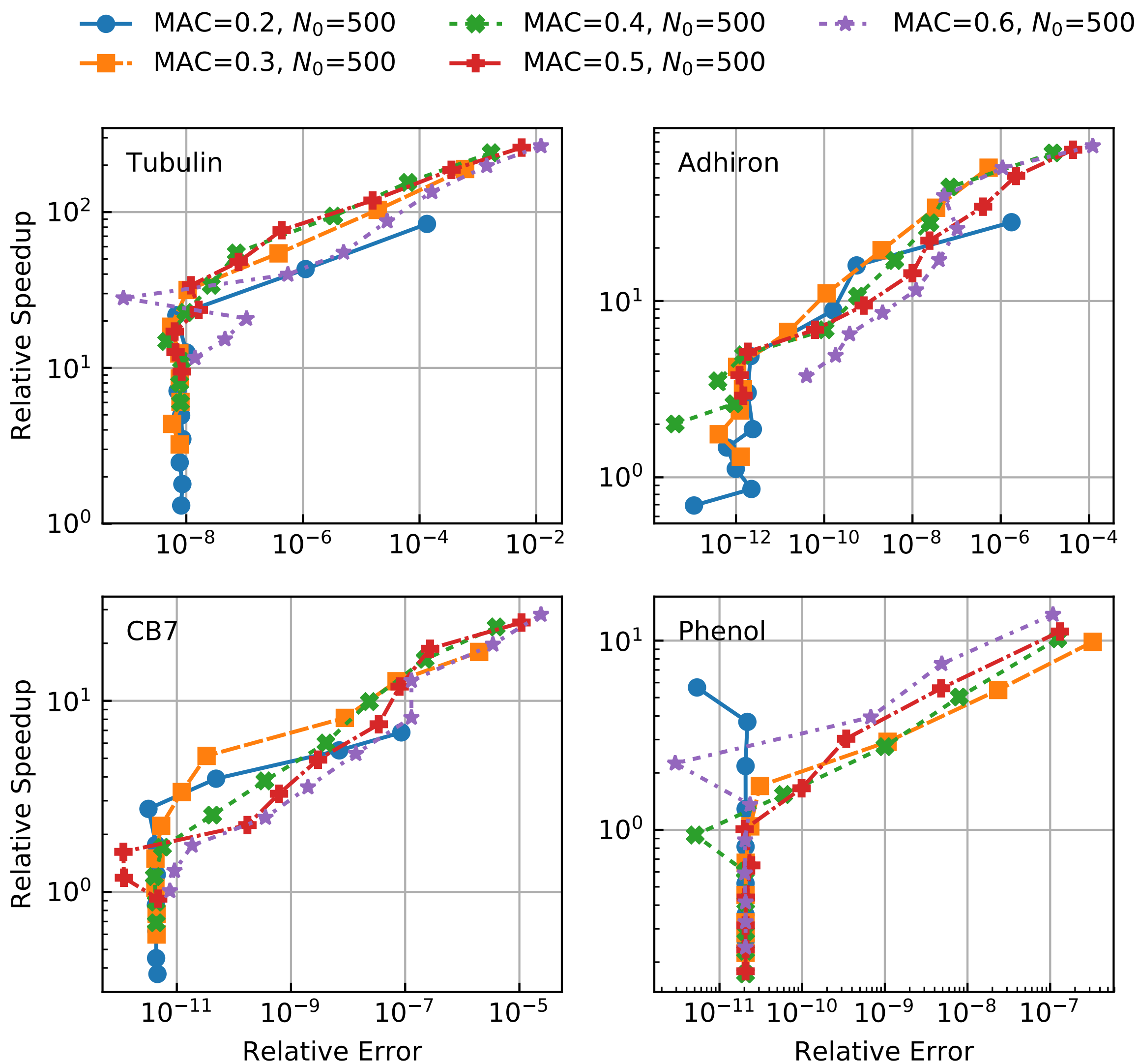


Figure 5.4: Relative speedup of treecode TCF LRA compared to direct summation versus relative error in $\mu_{\mathrm{ex,kh}}$ for tubulin, adhiron, CB7, and phenol, Taylor series order $p = 2k$, $k = 1, \ldots, 10$, increasing from right to left for each line. Simulations ran in serial on Metropolis cluster.

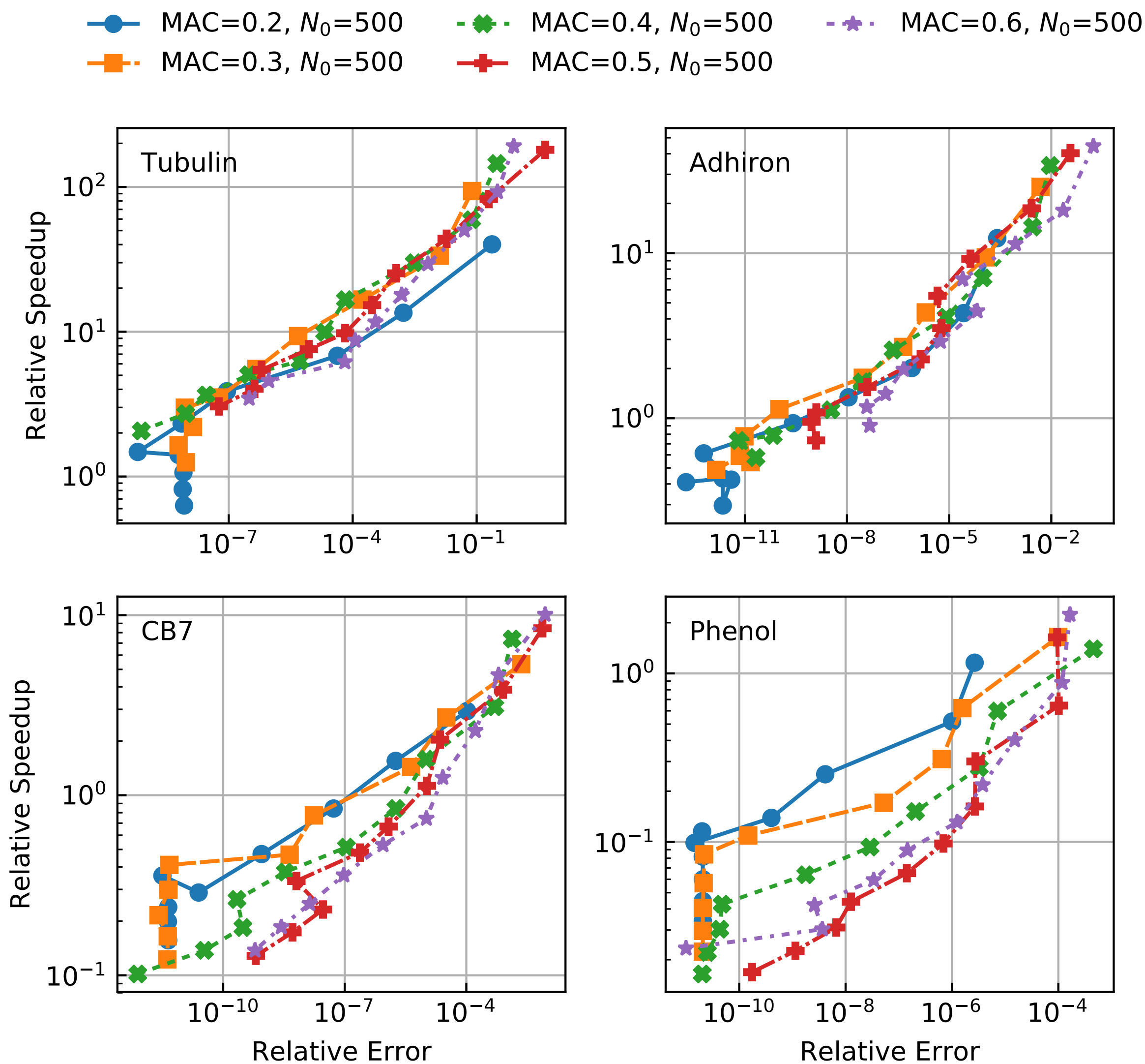


Figure 5.5: Relative speedup of treecode DCF LRA compared to direct summation versus relative error in $\mu_{\text{ex,kh}}$ for tubulin, adhiron, CB7, and phenol, Taylor series order $p = 2k$, $k = 1, \ldots, 10$, increasing from right to left for each line. Simulations ran in serial on Metropolis cluster.

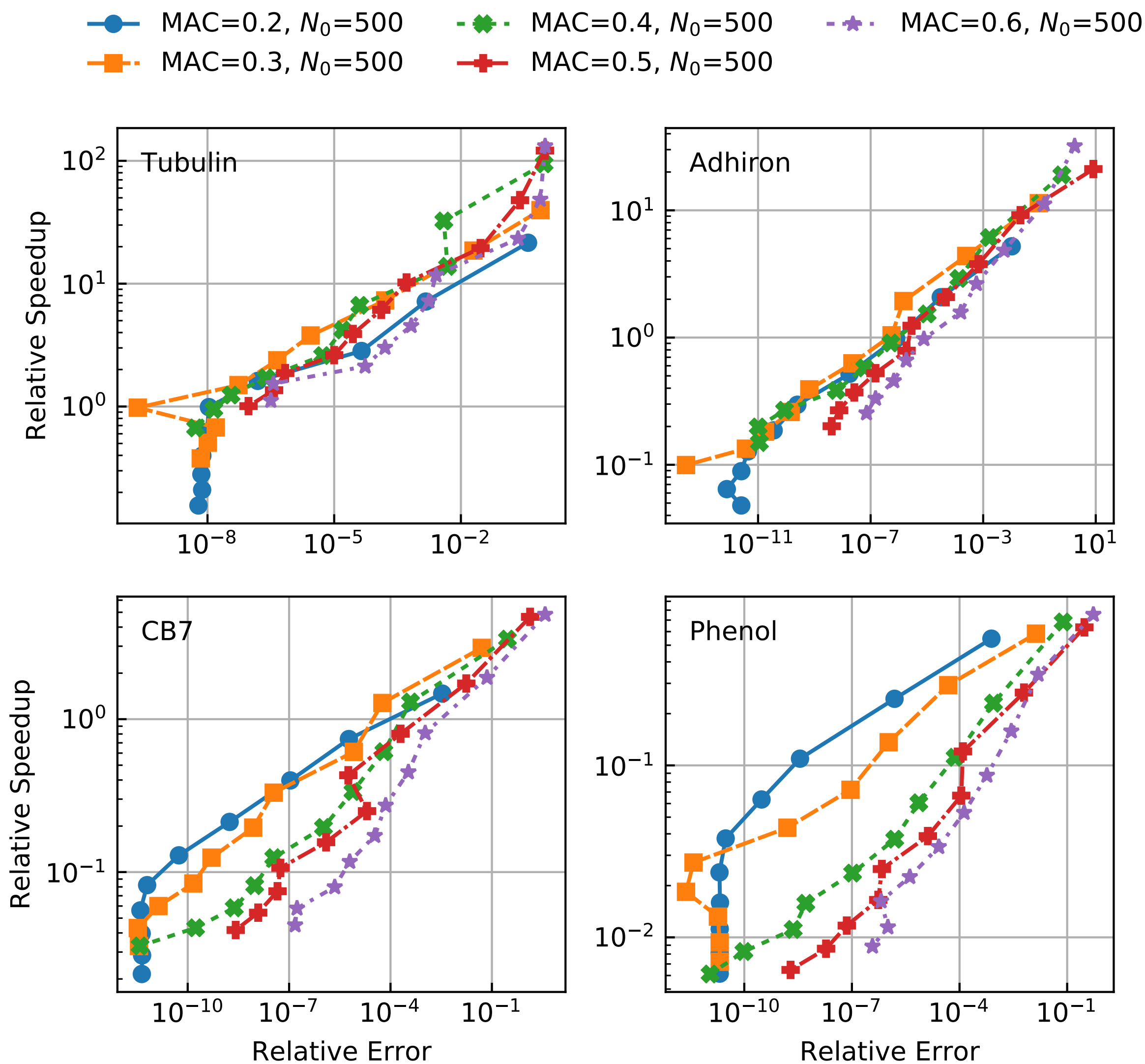


Figure 5.6: Relative speedup of treecode Coulomb potential energy compared to direct summation versus relative error in $\mu_{\mathrm{ex,kh}}$ for tubulin, adhiron, CB7, and phenol, Taylor series order $p = 2k$, $k = 1, \ldots, 10$, increasing from right to left for each line. Simulations ran in serial on Metropolis cluster.

	θ	p	N_0
TCF	0.3	$\max\left(2, \frac{\log_{10}(\text{tolerance})+5.7}{-0.7}\right)$	500
DCF	0.3	$\max\left(2, \frac{\log_{10}(\text{tolerance})+1.9}{-0.8}\right)$	500
Coulomb	0.3	$\max\left(2, \frac{\log_{10}(\text{tolerance})+1.4}{-0.8}\right)$	500

Table 5.3: Guide to selecting treecode parameters for a given residual tolerance. Recommended parameters should be tested before production use.

is generally slower than $\theta = 0.3$ and 0.4 to achieve the same relative error. For CB7, $\theta = 0.2$ and 0.3 have nearly identical results, outperforming larger MAC values. The trend towards better performance from smaller MAC values continues for phenol, though the tree code is generally slower than direct summation for this small solute.

The Coulomb potential energy has the simplest functional form and also shows the least benefit from treecode summation. Only tubulin has speedups at all relative errors. However, treecode summation is faster than direct summation for adhiron for relative errors $> 1\text{E}{-7}$ and for CB7 for relative errors $> 1\text{E}{-4}$. Treecode is slower than direct summation for all phenol calculations. Otherwise, the performance with different MAC values is similar to that observed for DCF LRA. The best performance for tubulin and adhiron is achieved with $\theta = 0.3$ and 0.4, while $\theta = 0.5$ has similar performance for larger relative errors but does not reach the lowest relative errors, even for $p = 20$. $\theta = 0.2$ and 0.3 again show similar performance for CB7, though they are faster than direct summation only for $p < 4$ and $p < 6$, respectively.

5.2.3.3 Treecode summation parameter selection

Even when considering just biological molecules, there is a wide range of shapes, sizes and charges for both the solutes and solvents that may be studied with 3D-RISM. As a result, it is not possible to prescribe a uniform set of parameters for treecode summation methods developed here; some testing will always need to be done before starting a large calculation. However, we can provide guidance to narrow the search for parameters that minimize computation time while preserving necessary numerical precision. Numerical precision is set by the user by specifying the residual tolerance at the beginning of the calculation. As shown in Fig. 5.3, relative error has a linear relationship with the residual tolerance. Therefore, we specify our guidelines relative to the residual tolerance.

Treecode summation requires the user to specify maximum leaf size N_0, MAC parameter θ, and Taylor series order p. Of these, N_0 and θ have clear best choices. $N_0 = 500$ is a safe and close to ideal choice for all calculations; $N_0 = 60$ provides almost identical performance

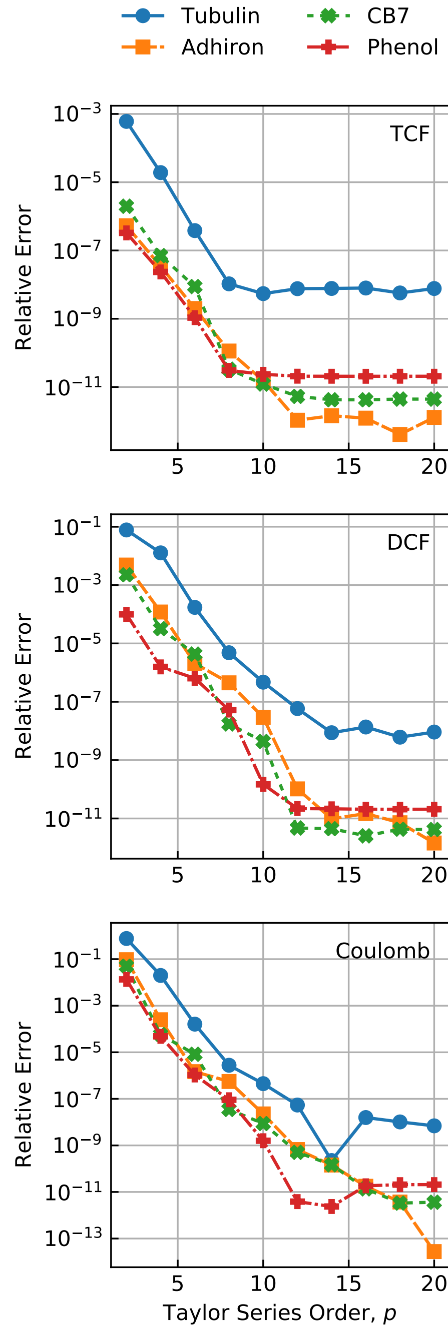


Figure 5.7: Relative error in $\mu_{\mathrm{ex,kh}}$ of 3D-RISM calculations with treecode parameters $\theta = 0.3$ and $N_0 = 500$ versus Taylor series order, p, for tubulin, adhiron, CB7, and phenol.

while $N_0 = 4000$ gives slower performance in some cases. If a smaller grid spacing of 0.25 Å is used, then a cluster of $N_0 = 500$ at this smaller grid spacing will occupy about the same volume as $N_0 = 60$ for a grid spacing of 0.5 Å and we would not expect a significant change in performance. We also recommend $\theta = 0.3$ for all calculations. While other values can be considered for the TCF LRA calculation, $\theta = 0.3$ performs well for all calculations where treecode is faster than direct summation. As observed for TCF and DCF LRA and Coulomb calculations, larger MAC values perform better for larger solutes; $\theta = 0.4$ may be a better choice for solutes larger than those considered here.

The Taylor series order is the most difficult parameter to select as it depends on both the size of the solute, the type of calculation being approximated, and the desired numerical precision. Fig. 5.7 shows the relationship between relative error and Taylor series order for $\theta = 0.3$ and $N_0 = 500$ from Figs. 5.4 to 5.6 grouped by calculation type across solutes. For all solutes and all calculations, we observe a linear relationship between $\log_{10}(\text{error})$ and p until the error due to the treecode is smaller than the error due to reaching the convergence criterion of the iterative solver, which is a residual tolerance of 1E−10 in this case. The slope in all cases appears similar, but there are different y-intercepts for the different solutes and calculation types. In addition, tubulin has systematically higher errors, likely due to the convergence anomaly shown in Fig. 5.3. In Table 5.3, we provide expressions for p-values based on the input residual tolerance, where we have used Eq. 5.24 to relate expected error to the input tolerance. As the case of tubulin demonstrates, these expressions are not exact. Rather, we recommend checking the relative error for a given p by performing a test calculation with the prescribed p and another with $p + 2$. If the error is sufficiently small, then other calculations can be performed with different conformations.

Note that treecode summation is not always faster than direct summation. In particular, for small molecules, it may be better to use direct summation for the DCF LRA and Coulomb potential.

5.2.3.4 Scaling with solute size

Using parameters determined in §5.2.3.3, we compare the compute time required for total cost of the calculation with treecode summation and cut-offs with the performance of direct summation (Fig. 5.8). For this comparison, we again use a residual tolerance of 1E−6, which is sufficient for most 3D-RISM calculations. For larger tolerances, more aggressive parameters can be used, resulting in potentially larger speedups. Combined, treecode summation and cut-off methods can significantly reduce the total calculation time, nearly 4× faster in the case of tubulin and 1.6× for adhiron. In the case of tubulin, computing the potential and asymptotics accounts for about 20% of the total runtime when using treecode and cut-offs

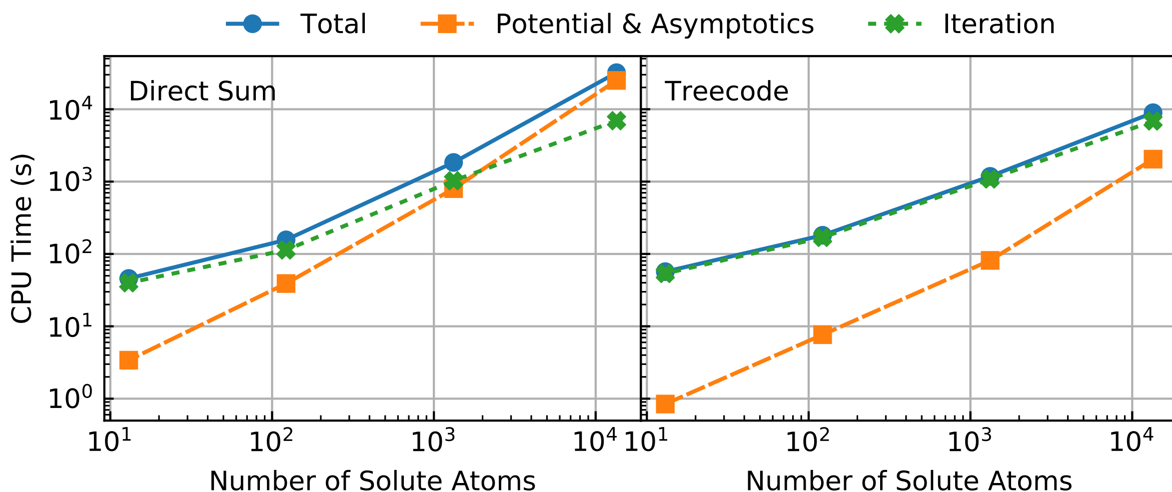


Figure 5.8: Total runtime of 3D-RISM converged to a tolerance of 1E−6, with potential and asymptotics calculated using direct and treecode summation. Required runtime is shown for setting up the calculations (potential and asymptotics) and iterating to a converged solution. Treecode and cut-off parameters can be found in Table 5.2. Simulations ran in serial on Metropolis cluster.

versus nearly 80% using direct summation. Smaller solutes obtain similar results; potential and asymptotics calculations are accelerated by a factor of 3× to 10× and, with the exception of tubulin, account for less than 10% of the total runtime when treecode summation is used. Overall, iteration time is now the dominant computational cost for all solute sizes.

To assess how the treecode summation methods perform, we have broken down the potential and asymptotics into their various components (Fig. 5.9). For the direct summation calculations, the real-space TCF LRA calculation dominates the runtime, followed by the real-space DCF LRA and the Coulomb potential energy calculations. After applying the treecode summation and cut-off methods, the real-space DCF LRA is the most expensive part of the calculation for all but tubulin while the real-space TCF LRA and Coulomb potential energy require about the same amount of time as the Lennard-Jones potential energy. Tubulin is an exception, as the reciprocal-space DCF and TCF LRA require the largest fraction of time, about 25% of the total time for the potential and asymptotics.

Using a cutoff for the reciprocal-space DCF and TCF LRA is the only optimization that does not improve the scaling with system size. As it is a cut-off in reciprocal space, only large values of k are omitted, which are determined by the grid spacing used and have nothing to do with solute size. As a result, we observe a performance improvement of 2.5-3.5× for all solutes and anticipate even greater speedups for finer grid-spacings. In fact, there should

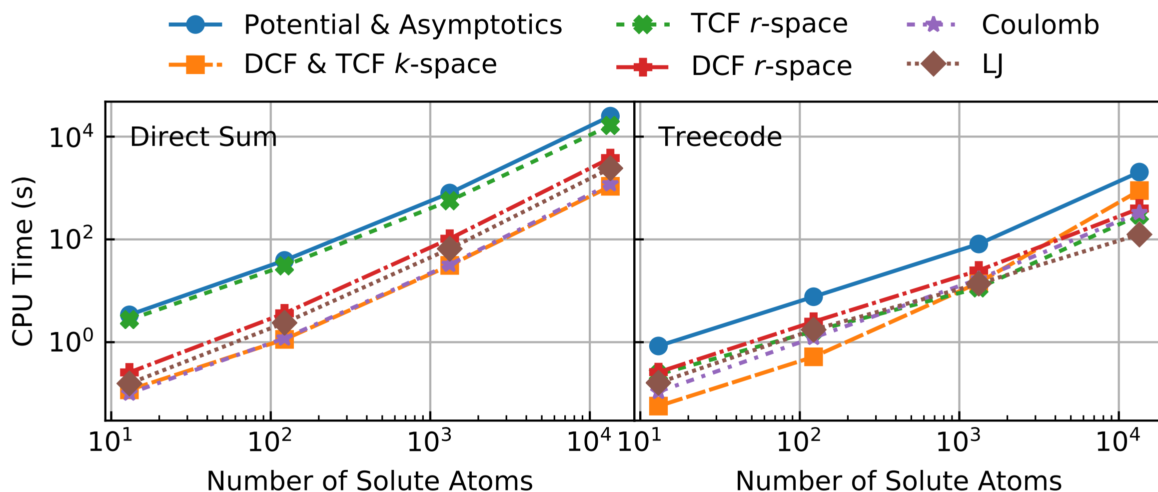


Figure 5.9: Runtime for different components of the potential and asymptotics calculation for Fig. 5.8 using direct and treecode summation. Calculations were solved to a residual tolerance of 1E−6. Simulations ran in serial on Metropolis cluster.

be little or no additional computation time for calculating the reciprocal-space DCF and TCF LRA on finer grids. Despite the fact that the scaling remains $O\left(N_{\text{atom}}N_{\text{grid}}\right)$, the use of cut-offs means that this part of the calculation remains a small fraction of the total and may be further reduced by other means, such as lookup-tables.

5.2.3.5 Parallel scaling

3D-RISM in AmberTools is parallelized using the message passing interface (MPI) with a distributed memory model. This allows 3D-RISM to make use of the aggregate memory of multiple nodes for large systems but means that the code must follow the memory model of the underlying FFT library for all of the solvation grids. We use the Fastest Fourier Transform in the West (FFTW) [147] library, which decomposes the memory in real-space along the z-axis into slabs. Each process gets one slab of each grid, whether or not that grid is directly processed by FFTW, and includes potential energy and LRA grids. In order to ensure adequate load balancing, 3D-RISM uses equal sized slabs for all nodes and will automatically increase the total grid size to ensure this if necessary. At the same time, each process gets a full copy of the solute information. This accounts for much less memory than the grids and is only a small fraction of the the aggregate memory footprint, even for 96 processes.

Treecode summation and cut-off methods have a small effect on the overall parallel scaling of 3D-RISM (Fig. 5.10). On the Metropolis cluster, with only 24 cores per node, calculations

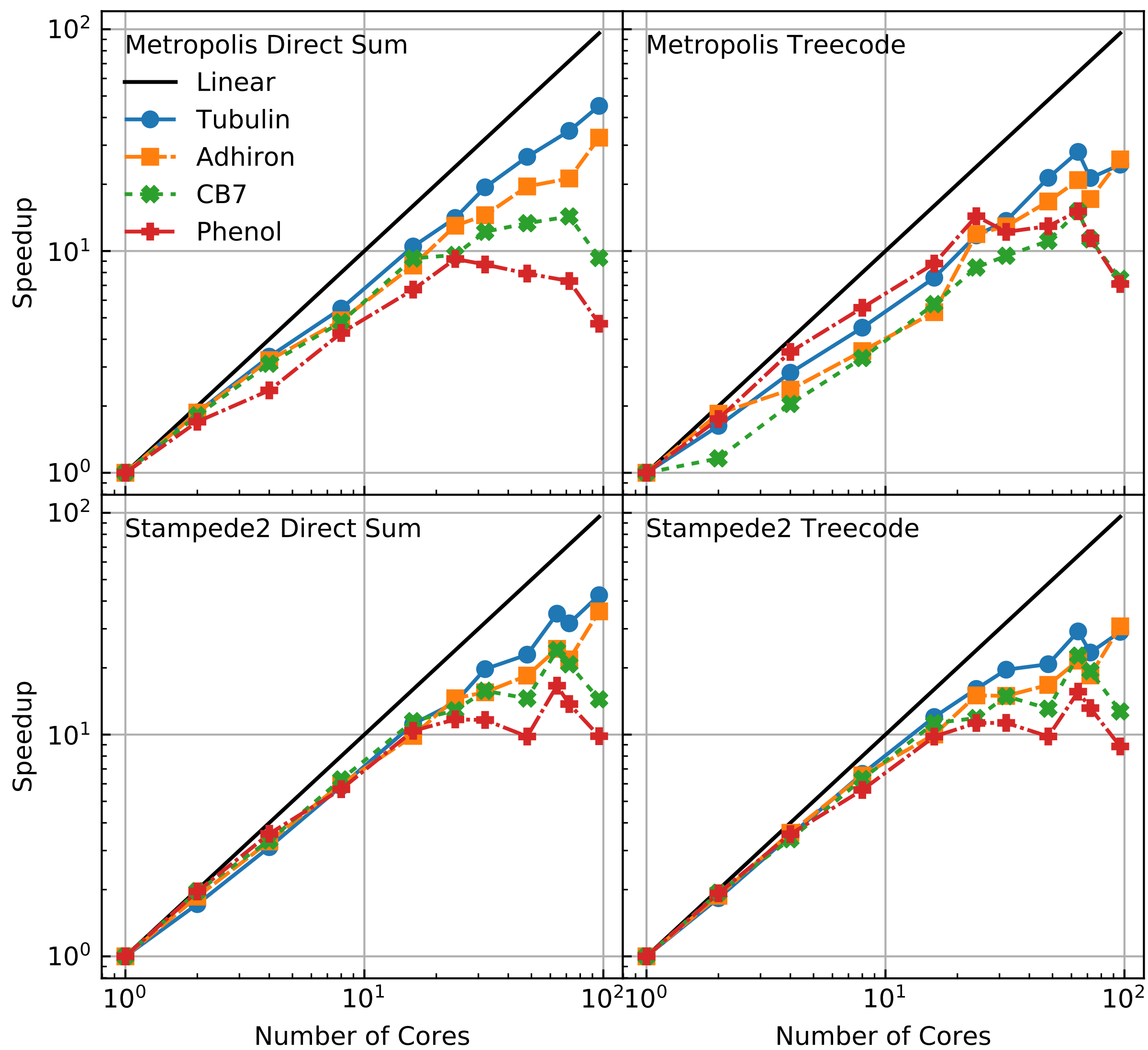


Figure 5.10: Speedup over multiple cores of the total calculation time for direct and treecode summation 3D-RISM calculations converged to a tolerance of 1E−6 on Metropolis and Stampede2 clusters. Treecode and cut-off parameters can be found in Table 5.2.

on all solutes scale well until 24 cores for both types of calculations. Adding resources beyond 24 cores causes the solution for phenol to slow down. CB7 is the next to saturate at about 72 cores for direct summation while adhiron and tubulin do not exhibit any slow down. As expected, large systems scale better than smaller systems. However, for the treecode summation and cut-off methods, 64 cores appears to be the limit for all solutes. In addition, phenol now exhibits the best scaling of all the systems until it saturates, while there is a notable decline in the scaling of CB7 and adhiron.

To investigate the role of hardware, we also ran calculations on Stampede2, which has double the cores and memory bandwidth of Metropolis (Fig. 5.10). As with Metropolis, all solutes scale well up to 24 cores for both direct summation and treecode methods. Unlike Metropolis, scaling is closer to linear and does not seem to be affected by solute size at these small core numbers. However, 24 cores remains the scaling limit for phenol, which indicates that this is a software limitation. After this point, larger solutes scale more efficiently and phenol and CB7 saturate at 24 and 64 cores respectively. Otherwise, treecode calculations scale as well as direct summation calculations until the high core counts are reached.

As we did with single-core performance, to better understand the contributions of different parts of the calculation, we have decomposed the calculation into various components for the potential and asympytotics calculations and the iteration time, the later of which we have not attempted to accelerate. We use tubulin for this discussion (Fig. 5.11), though the same behavior is observed for the other molecules as well.

For direct-sum calculations, the largest bottleneck to scaling is the iterative stage of the calculation. The scaling of this part of the code is sub-linearly and becomes the most expensive part of the calculation when eight or more processes are used. In contrast, all other parts of the calculation scale almost linearly. As each MPI process has a full copy of the solute, the direct sum calculation is trivially parallel, with no communication between the processes, and should scale linearly as observed. The cause of the sub-linear scaling of the iterative calculation is beyond the scope of this paper, but is likely hardware dependent as the iterative calculation performs much better on Stampede2. Profiling data (not shown) indicates that the iterative calculation has much higher memory bandwidth requirements than the direct summation, and the higher memory bandwidth of Stampede2 could account for these differences.

TCF and DCF LRA and Coulomb potential energy with treecode summation all scale well until around 32 cores on both Metropolis and Stampede2. The most likely reason for the scaling to plateau is that each process performs its own treecode decomposition on its own piece of the grid. Because a slab-decomposition memory layout is required by the FFTW3 library we use for the iterative part of the code, the memory that each process receives

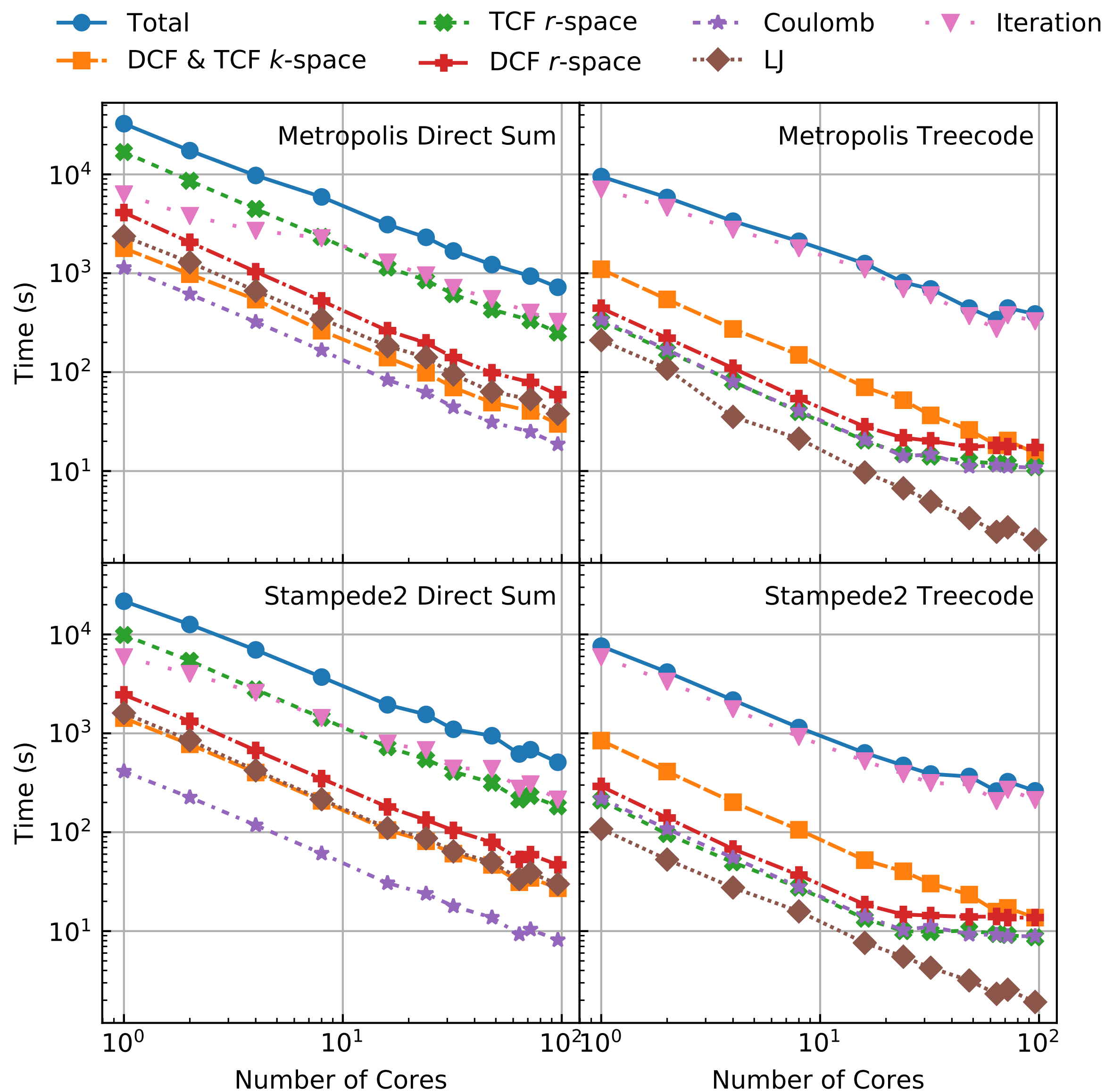


Figure 5.11: Computation time over multiple cores of the total calculation time and various components for direct and treecode summation 3D-RISM calculations on a tubulin dimer converged to a tolerance of 1E−6 on Metropolis and Stampede2 clusters. Treecode and cut-off parameters can be found in Table 5.2.

becomes narrower as the process count increases. As tree nodes narrow, it is more difficult to satisfy the MAC and the Taylor expansion becomes less efficient. To partially alleviate this constraint, when the tree is built, nodes are only subdivided along a given Cartesian direction if the node box length parallel to that direction is within a factor of $\sqrt{2}$ of the shortest box length. However, this can result in only two or four children in a given tree level, and the top levels of the tree will still have node boxes with uneven aspect ratios, so narrow tree root nodes may still affect performance. Additionally, slabs near the middle of the grid where the solute is located may end up doing more local source particle-target particle direct sums, while slabs near the edges of the grid will be able to use the Taylor expansion more often.

Overall, the performance of treecode summation for high process counts does not adversely affect the overall parallel scaling of the calculation as the total time and scaling is dominated by the iterative solver. Treecode summation is so much faster than direct summation that even at the highest node counts, it is an almost negligible part of the calculation.

5.3 Project 2: GPU-accelerated BLDTT 3D-RISM

5.3.1 Project description

The long-term vision of 3D-RISM is to have a fully GPU-accelerated molecular solvation package within Amber. A key piece of this effort is GPU acceleration of the LRA treecodes. BaryTree gives a model for building GPU-accelerated fast summation methods, and this section documents early implementation of a modified form of BaryTree and a cluster-particle barycentric Lagrange interpolation treecode (CP-BLTC) for calculating the TCF LRA, and its performance compared to the standard CP-BLTC in BaryTree. These efforts will, in future, be extended to the DCF LRA and Coulomb potentials in 3D-RISM.

5.3.2 Methodology and results

Instead of running a full 3D-RISM calculation, a modified version of the BaryTree CP-BLTC was developed for calculating the TCF LRA function. The 20s proteasome from yeast in complex with the proteasome activator PA26 (PDB ID: 1Z7Q) [148] was used as the solute biomolecule, and the system, including physical parameters, was prepared in the same manner as described in §5.2.2. Several gaps in the structure of 1Z7Q were present; rather than attempting to build the missing residues, the surrounding peptides were capped with acetyl (ACE) or NME groups and the final structure was parameterized with Amber FF14SB. The target particles were generated corresponding to a solvent grid with a 24Åbuffer. The final prepared solvent contains 148,724 source charges, and the solvent grid has dimensions $378 \times 432 \times 864$, for a total of 141,087,744 target particles.

The calculations are done in double precision arithmetic and the reported errors are the relative ℓ_2 error,

$$E = \left(\sum_{i=1}^{M} (\phi_i^{ds} - \phi_i^{fs})^2 \Bigg/ \sum_{i=1}^{M} (\phi_i^{ds})^2 \right)^{1/2}, \tag{5.25}$$

where ϕ_i^{ds} are the target grid TCF LRA values computed by direct summation and ϕ_i^{fs} are computed by fast summation. The error was sampled at a random subset of 0.01% of the target particles in all cases.

The computations were performed on the NVIDIA V100 GPU nodes on the University of Michigan Great Lakes computing cluster, where each node contains two GPUs, and each GPU has 16GB of memory. Each computation was run on a single GPU. The code was compiled with the PGI C compiler using the `-O3` optimization flag.

Beginning with the BaryTree implementation of the CP-BLTC with the TCF LRA interaction kernel, the implementation was iteratively improved and specialized for the 3D-RISM application. Figure 5.12 displays the performance of the original, unmodified BaryTree CP-BLTC (**blue**) in calculating the TCF LRA and a heavily specialized CP-BLTC for 3D-RISM (**green**). Connected curves represent constant MAC θ (0.7 $\times$, solid; 0.9 $\circ$, dashed), with interpolation degree $n = 1, 2, 4, 6, 8, 10$ increasing from right to left on each curve (original CP-BLTC run only to $n = 8$).

Note that the heavily modified CP-BLTC is nearly an order of magnitude more efficient than the original CP-BLTC. These changes were primarily implementation details, including replacing target particles in memory with a "virtual" grid of targets generated on the fly; more explicit memory management; "flattening" target cluster proxy points from their tensor product form to a decomposed form, with three arrays of the x, y, z coordinates necessary to generate the tensor product; and replacing the original $O(M \log M)$ CP-BLTC downpass with the BLDTT $O(M)$ downpass. More details on these successive improvements are given in Appendix B.

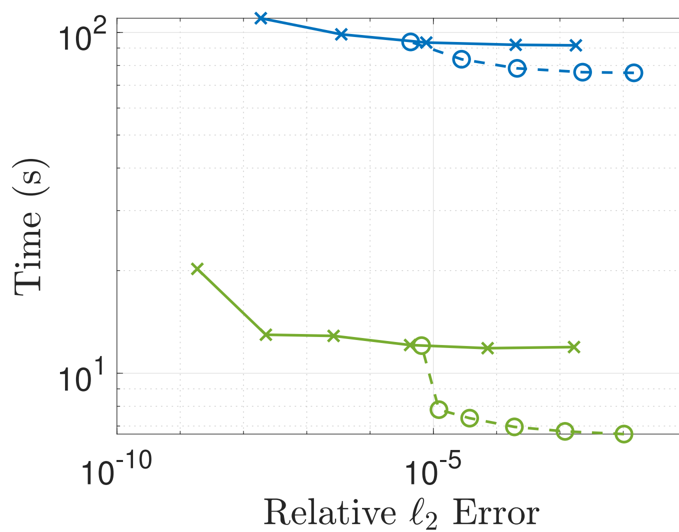


Figure 5.12: 1Z7Q TCF LRA calculation, original BaryTree CP-BLTC (**blue**), heavily specialized CP-BLTC for 3D-RISM (**green**), connected curves represent constant MAC θ (0.7 $\times$, solid; 0.9 $\circ$, dashed), interpolation degree $n = 1, 2, 4, 6, 8, 10$ increases from right to left on each curve (original CP-BLTC run only to $n = 8$), simulations ran on one NVIDIA P100 GPU.

CHAPTER 6

Conclusion

This work addressed the development of fast summation methods for long range particle interactions and their application to problems in biomolecular solvation, which describes the interaction of proteins or other biomolecules with their solvent environment. The primary result of this work was the development of an $O(N)$ dual tree traversal fast summation method based on barycentric Lagrange polynomial interpolation (BLDTT). This method was implemented to run across multiple GPU compute nodes in the software package BaryTree. Across different problem sizes, particle distributions, geometries, and interaction kernels, the BLDTT showed consistently better performance than the previously developed barycentric Lagrange treecode (BLTC).

The first major biomolecular solvation application of fast summation methods presented is to the Poisson–Boltzmann implicit solvent model, and in particular, the treecode-accelerated boundary integral Poisson–Boltzmann solver (TABI-PB). The work on TABI-PB consisted of three primary projects and an application. The first project investigated the impact of various biomolecular surface meshing codes on TABI-PB, and integrated the NanoShaper software into the package, resulting in significantly better performance. Second, a node patch method for discretizing the system of integral equations was introduced to replace the previous centroid collocation scheme, resulting in faster convergence of solvation energies. Third, a new version of TABI-PB with GPU acceleration based on the BLDTT was developed, resulting in even more scalability. An application investigating the binding of biomolecular complexes was undertaken using the previous Taylor treecode-based version of TABI-PB.

The second major application of fast summation methods is to the 3D reference interaction site model (3D-RISM), a statistical-mechanics based continuum solvation model. This work applied cluster-particle Taylor expansion treecodes to treat long-range asymptotic Coulomb-like potentials in 3D-RISM, and resulted in significant speedups and improved scalability to the 3D-RISM package implemented in AmberTools.

Next we summarize the major developments and current state of the BLDTT, TABI-PB, and 3D-RISM, and described several future directions for each.

6.1 BLDTT

6.1.1 Present work

We presented the barycentric Lagrange dual tree traversal (BLDTT) fast summation method for particle interactions, and its OpenACC implementation with MPI remote memory access for distributed memory parallelization running on multiple GPUs. The BLDTT builds adaptive trees of clusters on the target particles and source particles, where each cluster is the minimal bounding box of its particles, and a parent cluster may have 8, 4, or 2 children. The BLDTT uses a dual tree traversal strategy [42, 49] to determine well-separated target and source clusters that interact through particle-cluster (PC), cluster-particle (CP), or cluster-cluster (CC) approximations based on barycentric Lagrange interpolation at proxy particles defined by tensor product Chebyshev points of the 2nd kind in each cluster [65, 66]. The BLDTT has an upward pass and downward pass similar those in the FMM [43], although it relies on interlevel polynomial interpolation rather than translation of expansion coefficients. The BLDTT is kernel-independent, requiring only kernel evaluations. The distributed memory parallelization of the BLDTT uses recursive coordinate bisection for domain decomposition and MPI remote memory access to build locally essential trees on each rank [86]. The PP, PC, CP, and CC interactions all have a direct sum form that efficiently maps onto the GPU, where target particles provide an outer level of parallelism, and source particles provide an inner level of parallelism. The BLDTT code is part of the BaryTree library for fast summation of particle interactions available on GitHub at `github.com/Treecodes/BaryTree`.

The performance of the BLDTT was compared with that of the BLTC, an earlier particle-cluster barycentric Lagrange treecode [66]. For the systems of size N=1E5 to N=1E8 studied here running on a single GPU, the BLTC scales like $O(N \log N)$, while the BLDTT scales like $O(N)$. The BLDTT was applied to different random particle distributions (uniform, Gaussian, Plummer), different particle domains (thin slab, square rod, spherical surface), unequal sets of target and source particles, and different interaction kernels (oscillatory, Yukawa, singular and regularized Coulomb). The BLDTT had consistently better performance than the BLTC, showing its ability to adapt to different situations. Finally, the MPI strong scaling of the BLDTT and BLTC was demonstrated on up to 32 GPUs for N=64E6 particles with 7-8 digit accuracy. Across these simulations the parallel efficiency of both codes is better than 64%, while the BLDTT is 1.5-2.5× faster than the BLTC.

6.1.2 Future work

Future work to further improve the efficiency of the BLDTT could investigate overlapping communication and computation, building tree nodes that span multiple ranks, using

mixed-precision arithmetic, and employing barycentric Hermite interpolation [149]. We also anticipate applying the BLDTT to potentially speed up density functional theory calculations [150]. Additionally, we anticipate extending the BLDTT approach to high oscillatory interaction kernels and periodic boundary conditions.

6.2 TABI-PB

6.2.1 Present work

Over the course of three primary projects, a new TABI-PB solver with GPU acceleration using the NanoShaper software instead of the MSMS software for biomolecular surface meshing, a node patch method instead of centroid collocation for discretizing the integral equations, and the BLDTT fast summation method for computing matrix-vector products during GMRes iteration.

The treecode-accelerated boundary integral (TABI-PB) solver utilizes a well-conditioned boundary integral formulation and node patching on a biomolecular surface mesh triangulation. In these calculations, the linear system for the electrostatic potential and its normal derivative on the SES is solved by GMRes iteration. The matrix-vector product in each step of GMRES is computed by the BLDTT fast summation method which reduces the computational cost from $O(N^2)$ to $O(N)$, where N is the number of elements in the surface triangulation.

In previous versions of TABI-PB, centroid collocation was used instead of node patch for constructing the linear system of equations, including in the version used that compared MSMS and NanoShaper for surface meshing. Also in previous versions of TABI-PB, $O(N \log N)$ Taylor expansion particle-cluster treecodes were used instead of the BLDTT, including in the versions use that compared MSMS and NanoShaper for surface meshing and investigated the node patch method.

In the first project, the MSMS and NanoShaper codes triangulating the solvent excluded surface (SES) were compared for a test set of 38 biomolecules. The meshes produced by the two codes are qualitatively similar, although the MSMS meshes often contained triangles of exceedingly small area and high aspect ratio. The computed values of the surface area and solvation energy produced by MSMS and NanoShaper meshes often agree to within several percent. NanoShaper meshes were more computationally efficient, requiring less run time and fewer GMRES iterations than MSMS meshes. Furthermore, NanoShaper was consistently able to produce higher resolution meshes than MSMS, and NanoShaper solvation energies exhibited smoother convergence with increasing mesh resolution.

In the second project, the node patch method and centroid collocation were compared across a set of test biomolecules. In all cases, the node patch method showed faster convergence

than centroid collocation. Additionally, node patch in general requires less computational elements and less CPU time than collocation to reach the same level of accuracy.

In the third project, TABI-PB was rewritten from the ground up in object oriented C++, restructuring the integral kernels, implementing the BLDTT for calculating matrix-vector products and adding GPU acceleration. TABI-PB with BLDTT and TABI-PB with Taylor expansion treecodes are compared across four test biomolecules, and BLDTT TABI-PB displays clearly better scaling. For the densest meshes run, BLDTT TABI-PB is roughly 2.5× faster than Taylor treecode TABI-PB. Additionally, for the fast summation parameter choices run, BLDTT TABI-PB had 5× less fast summation error than Taylor treecode TABI-PB, suggesting that the BLDTT TABI-PB interpolation degree could be pushed even lower while maintaining the same accuracy. The performance of the BLDTT TABI-PB on a single CPU core was compared to its performance on a GPU; the speedups observed were respectable, but the NanoShaper surface meshing software, which runs only on the CPU, has become the largest component of the total run time for GPU-accelerated BLDTT TABI-PB.

A version of TABI-PB using NanoShaper was recently implemented as an option in APBS [29]. The most recent release of APBS includes TABI-PB with BLDTT, node patch, and NanoShaper. TABI-PB with GPU acceleration is available on GitHub at `github.com/Treecodes/TABI-PB`.

6.2.2 Future work

Future work on feature additions to TABI-PB will include the extension of the package to include non-polar energy calculation and molecular mechanics methods for more detailed binding free energy calculations of ligand docking and protein interactions. These proposed improvements are detailed here.

6.2.2.1 Non-Polar Solvation Energy Calculation

The Poisson–Boltzmann equation and our TABI-PB solver only calculate electrostatic solvation effects. However, if we wish to build a user-friendly biomolecular simulation package that has the potential to be widely accepted by the community, it is necessary to extend our calculation to include non-polar solvation effects when calculating solvation energies and binding free energies. The non-polar solvation energy $\Delta G_{\text{solv,non-polar}}$ is produced by repulsive cavity formation forces and attractive van der Waals forces between the solute and solvent [151, 152]. We described here several models for estimating non-polar energies.

- The SASA-Only Model, which ignores the attractive van der Waals contribution, approximates the non-polar contribution as a linearly dependent function of the solvent

accessible surface area (SASA) [151],

$$\Delta G_{\text{solv,non-polar}} \approx \Delta G_{\text{cavity}} \approx \gamma A + b, \tag{6.1}$$

where A is the surface area of the solvent accessible surface, b is a fitting parameter, and γ is a parameter related to the surface tension of the solvent.

- The SAV-Only Model, which also ignores the vdW contribution, approximates the non-polar contribution as a linearly dependent function of the solvent accessible volume [152],

$$\Delta G_{\text{solv,non-polar}} \approx \Delta G_{\text{cavity}} \approx pV + b, \tag{6.2}$$

  where V is the volume of the solvent accessible surface, b is a fitting parameter, and p is a parameter related to the solvent pressure.

- The SASA-SAV Model is a combination of the previous two models, again ignoring the vdW contribution [153, 152],

$$\Delta G_{\text{solv,non-polar}} \approx \Delta G_{\text{cavity}} \approx \gamma A + pV, \tag{6.3}$$

  where the parameters are as above.

- The SASA-SAV-WCA Model combines the above approach with the Weeks–Chandler–Anderson (WCA) theory [154] to quantify the vdW contribution [152],

$$\Delta G_{\text{solv,non-polar}} \approx \gamma A + pV + \Delta G_{\text{vdW}}, \tag{6.4}$$

  where the ΔG_{vdW} term is modeled with the attractive term of a WCA-like integral, corresponding to the integral of the attractive term of a Lennard-Jones interaction between the solute atoms and a uniformly distributed solvent outside of the cavity.

The optimal parameter values in all of the above models have been previously investigated [155, 152, 151]. We hope to implement all of the above models using standard parameter values into TABI-PB and investigate the accuracy of the resulting total free energies with respect to standard data sets.

6.2.2.2 MM-PBSA Method for Calculating Binding Free Energies

The Molecular Mechanics Poisson–Boltzmann Surface Area methodology is a popular approach for computing full binding free energies in continuum solvent [156, 157, 158, 159].

The binding free energy of a complex containing two subunits A and B is calculated by the difference in free energies

$$\Delta\Delta G^{\text{bind}} = \langle \Delta G^{\text{AB}} \rangle - \langle \Delta G^{\text{A}} \rangle - \langle \Delta G^{\text{B}} \rangle. \tag{6.5}$$

where the angle brackets denote averages calculated over multiple conformations. The subunits A and B could be, for instance, a protein and a ligand, a protein and a protein, or a protein and a nucleic acid. G^{AB} represents the free energy of the bound complex consisting of subunits A and B. Each free energy in the expression above can be decomposed into molecular mechanical, solvation, and entropic terms:

$$\langle \Delta G \rangle = \langle E_{\text{MM}} \rangle + \langle \Delta G_{\text{solv}} \rangle - TS \tag{6.6}$$

where E_{MM} is the potential energy generated by molecular mechanical interactions in vacuum, T is temperature, S is entropy, and ΔG_{solv} is the free energy of solvation. The ΔG_{solv} term contains, as explained above, polar and non-polar contributions. The molecular mechanics potential energy term E_{MM} contains bonding, electrostatic, and van der Waals interactions. The electrostatic effects are modeled with the Coulomb potential and the van der Waals effects are modeled with the Lennard-Jones potential. The thermodynamic loop associated with computing $\Delta\Delta G_{\text{bind}}$ is illustrated in Fig. 6.1.

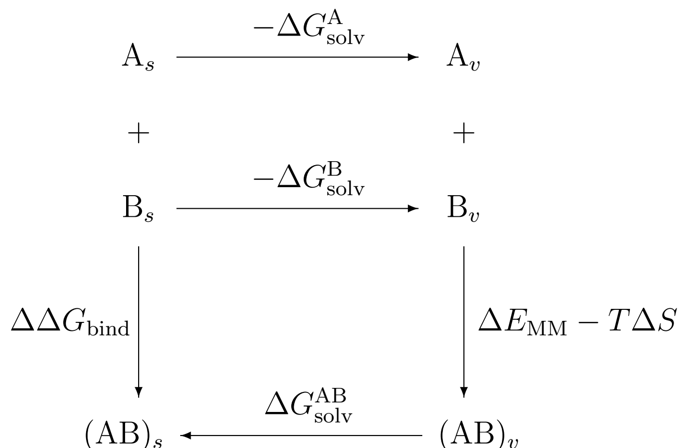


Figure 6.1: Thermodynamic loop illustrating the binding of two monomers A and B. Subscripts s, v denote solvent and vacuum, respectively. Binding free energy $\Delta\Delta G_{\text{bind}}$ is determined by the change in vacuum molecular mechanics energy ΔE_{MM}, containing van der Waals, electrostatic, and bonding terms, an entropic contribution $T\Delta S$, and binding solvation energy $\Delta\Delta G_{\text{solv}} = \Delta G^{\text{AB}}_{\text{solv}} - \Delta G^{\text{A}}_{\text{solv}} - \Delta G^{\text{B}}_{\text{solv}}$, containing polar and non-polar terms.

The averages for the A and B subunits and the complex AB in Eq. 6.5 should, in the most physically correct sense, be computed over three separate molecular dynamics trajectories for

A, B, and AB. This approach is typically denoted three-average MM-PBSA, or 3A-MM-PBSA. It is computationally more convenient to draw the trajectories from one simulation. Thus, we estimate the binding free energy as

$$\Delta\Delta G_{\text{bind}} \approx \langle \Delta G^{\text{AB}} - \Delta G^{\text{A}} - \Delta G^{\text{B}} \rangle_{\text{AB}} \tag{6.7}$$

where the AB subscript denotes that the trajectory over which the averages are computed is that of the complex. This approach is denoted 1A-MM-PBSA. Practically, this approach is often more accurate (and, of course, faster) than 3A-MM-PBSA because of the cancellation of intramolecular contributions [156]. In particular, all bonding terms cancel, so that E_{MM} contains only electrostatic and van der Waals interactions. For 1A-MM-PBSA, then, the free energy expressions G in Eq. (6.7) need only consist of the following terms,

$$\Delta G = E_{\text{electrostatic}} + E_{\text{vdW}} + \Delta G_{\text{solv,polar}} + \Delta G_{\text{solv,non-polar}} - TS, \tag{6.8}$$

where $\Delta G_{\text{solv,polar}}$ is computed by the Poisson–Boltzmann equation, $\Delta G_{\text{solv,non-polar}}$ is computed using one of the methods described in §6.2.2.1, $E_{\text{electrostatic}}$ is modeled as a Coulomb interaction in vacuum between the solute atoms, and E_{vdW} is modeled as a Lennard-Jones interaction between the solute atoms. The entropic TS term is often omitted as well; when this omission is made, binding free energies between various protein-ligand complexes can only be compared on a relative basis.

We propose to integrate an MM-PBSA binding calculation into TABI-PB. The user would input the typical TABI-PB parameters and PQR files for the subunits, in addition to a trajectory file generated by a molecular dynamics simulation of the complex, using a program such as GROMACS [160]. Our program would then extract snapshots from the trajectory file, generating an uncorrelated subset. For each snapshot, molecular surfaces would be generated for the entire complex and for each of the subunits, and solvation energies would be calculated for each of the three surfaces. The non-polar solvation energy definition would also be a user specifiable parameter. After calculation electrostatic and Lennard-Jones interactions for the three molecules for each snapshot, averages would be calculated. We would additionally use statistical methods to estimate the precision of the binding free energy.

To benchmark our method, we would use a standard set of 37 HIV-1 protease inhibitor complexes proposed in [161]. Note that, the inhibition constant K_i of the complex, which, for reversible competitive inhibitors is equivalent to the dissociation constant K_d, is related to the binding free energy by

$$\Delta\Delta G_{\text{bind}} = -RT \ln\left(1/K_i\right) \tag{6.9}$$

where R is the gas constant and T is the temperature [162]. Because experimental values for K_i have been measured for this set of protease inhibitor complexes, we can directly compare our results to experimental values. Additionally, we will compare the polar solvation energy components of the calculated binding free energies to our earlier static binding solvation energy computations discussed in §4.5.

6.3 RISM

6.3.1 Present work

In this work, we have developed and implemented treecode summation for long-range interactions to accelerate the potential and asymptotics calculations for non-periodic 3D-RISM calculations. The previous approach used direct sum calculations that scaled as $O\left(N_{\mathrm{grid}} N_{\mathrm{atom}}\right)$ and were a major impediment to studying large proteins and protein complexes. By implementing the numerical methods demonstrated here, we have reduced the computational complexity to at most $O((N_{\mathrm{grid}} + N_{\mathrm{atom}}) \log N_{\mathrm{grid}})$ for almost all parts of the calculation. For the largest protein we considered, tubulin, the total computation time was reduced by a factor of 4 and the potential and asymptotics now account for only 20% of the calculation time, compared to 80% when direct summation was used. Parallel calculations with these new methods scale almost linearly and the iterative solver remains the largest impediment to parallel scaling.

We additionally presented early work on GPU-accelerated treecodes for long range asymptotic calculations in 3D-RISM, in particular for the TCF LRA, based on a heavily specialized version of cluster-particle barycentric Lagrange interpolation treecodes (CP-BLTC) implemented in BaryTree.

6.3.2 Future work

Future work will focus on extending the GPU implementation to accelerate the other Coulomb and Coulomb-like LRA potentials in 3D-RISM. Due to the complexity of these kernels, the kernel-independent nature of barycentric treecodes is especially advantageous in this application. Extending the cluster-particle approach to a full BLDTT-based approach will also be investigated.

Further research will also develop a real-space method for solving RISM with free space boundary conditions. Singularity subtraction schemes for handling singularities in the resulting volume integrals will be investigated. The BLDTT has the potential to be a transformative option for accelerating the solution of real-space volume integral equations. Previous work from Nathan Vaughn has demonstrated the power of such an approach in

treating problems in electronic structure theory [150]. In contrast to Fourier transform-based approaches, this approach will be inherently more scalable and parallelizable, and more capable of accurately capturing solvent behavior for free space boundary conditions.

APPENDIX A

Implementation Details of BaryTree

This appendix describes some implementation details of the BaryTree software package discussed in this thesis, with the hopes of highlighting some lessons for future students.

A.1 BaryTree algorithm

This section expands upon BaryTree implementation details presented in §3.3. Algorithm A.1 gives a pseudocode implementation of the BLDTT in BaryTree. Operations involving MPI calls are marked with MPI, operations involving GPU compute kernels are marked with GPU, and data transfers are marked Data, HtD for host (CPU) to device (GPU) and Data, DtH for device (GPU) to host (CPU). Data, Dev marks data operations on the GPU that do not involve a transfer, such as allocating and deleting memory. Data operations on the CPU are not marked. Braces are used to emphasize quantities corresponding to *sets* of objects, like clusters, and *arrays* of values, like potentials and charges. This algorithm is described in more detail, with some comments, below.

- **Lines 1–2.** Each rank begins with a set of target particles $\{\mathbf{x}_i\}$ and source particles $\{\mathbf{y}_j\}$ which may or may not be coincident or localized. The particles are load balanced using the Zoltan library, which also guarantees domain localization. The Zoltan library executes MPI calls between all ranks to perform the load balancing.
- **Lines 3–4.** Each rank builds trees of source clusters $\{C_s\}$ and target clusters $\{C_t\}$ on its new localized sets of particles. Note that these correspond to *sets* of clusters. The tree building process reorganizes the local $\{\mathbf{x}_i\}$ and $\{\mathbf{y}_j\}$ arrays so that, for any cluster, the particles it contains are contiguous in memory. Specifically, this tree building process produces multiple arrays whose entries correspond to each cluster. Two arrays contain the starting and ending indices in the particle arrays, corresponding to the particles that each cluster contains. Two arrays contain parent and child information, effectively providing a "linked list" structure for tree traversal. One array contains the radius of the cluster,

Algorithm A.1 BaryTree BLDTT implementation, with special attention paid to MPI calls, GPU data transfers, and GPU compute kernels.

1: **procedure** BaryTree BLDTT
2: MPI: load balance target and source particles $\{\mathbf{x}_i\}$, $\{\mathbf{y}_j\}$ among all ranks
3: build tree of target clusters $\{C_t\}$ from local $\{\mathbf{x}_i\}$
4: build tree of source clusters $\{C_s\}$ from local $\{\mathbf{y}_j\}$
5: Data, HtD: copy local $\{\mathbf{x}_i\}$ and $\{\mathbf{y}_j\}$
6: Data, HtD: copy proxy particles $\{\mathbf{s_k}\}$ of $\{C_s\}$ and $\{\mathbf{t}_{\boldsymbol{\ell}}\}$ of $\{C_t\}$
7: Data, Dev: allocate space for local potential $\{\phi\}$
8: Data, Dev: allocate space for proxy charges $\{\widehat{q}_{\mathbf{k}}\}$ of $\{C_s\}$ and potentials $\{\widehat{\phi}_{\boldsymbol{\ell}}\}$ of $\{C_t\}$
9: GPU: UpwardPass($\{\mathbf{y}_j\}, \{C_s\}$), which computes $\{\widehat{q}_{\mathbf{k}}\}$
10: Data, DtH: copy $\{\widehat{q}_{\mathbf{k}}\}$
11: MPI: create RMA windows to local data
12: **for** each remote rank r **do**
13: MPI: get source cluster arrays from r necessary to form skeleton tree
14: determine needed remote particle and cluster data for LET_r with tree traversal
15: **end for**
16: **for** each remote rank r **do**
17: MPI, async: get needed data from r and fill LET_r
18: **end for**
19: MPI, wait
20: construct interaction lists $\mathcal{L}(\{C_t\}, \{C_s\})$ with dual tree traversal
21: GPU: ComputePotential($\mathcal{L}(\{C_t\}, \{C_s\})$)
22: Data, Dev: delete local $\{\mathbf{y}_j\}$, $\{\mathbf{s_k}\}$, and $\{\widehat{q}_{\mathbf{k}}\}$
23: **for** each remote rank r **do**
24: Data, HtD: copy LET_r data (particles, proxy particles and charges)
25: construct interaction lists $\mathcal{L}(\{C_t\}, \text{LET}_r)$ with dual tree traversal
26: GPU: ComputePotential($\mathcal{L}(\{C_t\}, \text{LET}_r)$)
27: Data, Dev: delete LET_r data
28: **end for**
29: GPU: DownwardPass($\{\phi\}, \{C_t\}$), which increments $\{\phi\}$ with $\{\widehat{\phi}_{\boldsymbol{\ell}}\}$
30: Data, DtH: copy $\{\phi\}$
31: Data, Dev: delete remaining memory
32: **end procedure**

and three contain the x, y, z components of the cluster center. Three arrays contain the x, y, z components of the proxy particles of the cluster; for an n degree interpolation, these arrays contain $(n+1)^3$ entries for each cluster. We denote these by $\{\mathbf{s_k}\}$ for $\{C_s\}$ and $\{\mathbf{t}_\ell\}$ for $\{C_t\}$ For the source clusters, one array allocates space for storing proxy charges at each proxy particle, denoted $\{\widehat{q}_\mathbf{k}\}$, and for the target clusters, one array allocates space for storing proxy potentials at each proxy particle, denoted $\{\widehat{\phi}_\ell\}$.

- **Lines 5–6.** After the trees are built, a host-to-device date transfer copies the particle data $\{\mathbf{x}_i\}$ and $\{\mathbf{y}_j\}$ and proxy particles $\{\mathbf{s_k}\}$ and $\{\mathbf{t}_\ell\}$ to the GPU.

- **Lines 7–8.** Space is allocated on the GPU for the local potential array $\{\phi\}$, with each entry corresponding to a local source particle. Space is additionally allocated on the GPU for the proxy charges $\{\widehat{q}_\mathbf{k}\}$ and the proxy potentials $\{\widehat{\phi}_\ell\}$. Importantly, no actual data *transfer* occurs on lines 7–8. The corresponding arrays in CPU memory are either zeroed out or junk at this point, so there is no need to perform a transfer. The space is simply allocated. This is good—transfers are expensive, and should be limited as much as possible.

- **Lines 9–10.** The proxy charges $\{\widehat{q}_\mathbf{k}\}$ are then computed on the GPU using the upward pass, and are copied back out to CPU memory. This procedure is given by Algorithm A.3. The $\{\widehat{q}_\mathbf{k}\}$ must be copied back to CPU memory, because other ranks could possibly require this data during formation of locally essential trees (LETs).

- **Line 11.** MPI Remote Memory Access (RMA) windows are then formed on the source particle data $\{\mathbf{y}_j\}$ and all arrays corresponding to source clusters, including $\{\mathbf{s_k}\}$ and $\{\widehat{q}_\mathbf{k}\}$.

- **Lines 12–15.** We then perform the first series of `get` operations. Each rank allocates space for and `get`s only the source cluster arrays from every other rank necessary to form a "skeleton tree". Essentially, this means only the arrays containing the cluster centers, radii, and child information; at this point there is no need to communicate data about the remote $\{\mathbf{s_k}\}$, $\{\widehat{q}_\mathbf{k}\}$, or $\{\mathbf{y}_j\}$. For each remote rank, a dual tree traversal is performed between the local target tree $\{C_t\}$ and the remote rank's skeleton source tree. This traversal determines which remote $\{\mathbf{s_k}\}$, $\{\widehat{q}_\mathbf{k}\}$, or $\{\mathbf{y}_j\}$ will be needed to fill the LET, denoted LET_r for rank r.

- **Lines 16–19.** Another loop is performed over the remote ranks to actually perform the `get` operations on the needed remote $\{\mathbf{s_k}\}$, $\{\widehat{q}_\mathbf{k}\}$, and $\{\mathbf{y}_j\}$ data. In this loop, the MPI `get` operations are performed *asynchronously*, and an MPI `wait` on line 19 blocks further operations until all `get`s have been performed. In the MPI parlance, these MPI `get` operations are "non-blocking" calls. This squeezes out some additional performance, in that there is no need to wait for a `get` operation to be completed before another one is

launched. The networking fabric can instead optimize the execution of the `get`s between all r ranks. LET_r is a collection of source particles and clusters, like $\{C_s\}$, except its source particle and proxy particle and charge arrays are truncated to only include needed data.

- **Line 20.** The first set of interaction lists are formed by a dual tree traversal between the local target tree $\{C_t\}$ and local source tree $\{C_s\}$. These interaction lists are denoted $\mathcal{L}(\{C_t\}, \{C_s\})$, to emphasize that they are functions of $\{C_t\}$ and $\{C_s\}$. The full algorithm for building the interaction lists is given by the recursive dual tree traversal in Algorithm 3.2, which begins at the roots of $\{C_t\}$ and $\{C_s\}$. There are really *four* interaction lists contained within $\mathcal{L}$, one for all target cluster-source cluster pairs (C_t, C_s) that interact directly, i.e., with PP interactions, one for all pairs with PC interactions, one for all pairs with CP interactions, and one for all pairs with CC interactions.

- **Line 21.** ComputePotential is called on the interaction lists $\mathcal{L}(\{C_t\}, \{C_s\})$, to increment the local potential array $\{\phi\}$ and the proxy potentials $\{\widehat{\phi}_{\boldsymbol{\ell}}\}$. This procedure is given by Algorithm A.2.

- **Line 22.** The local $\{\mathbf{s}_{\mathbf{k}}\}$, $\{\widehat{q}_{\mathbf{k}}\}$, and $\{\mathbf{y}_j\}$ data are deleted from the GPU.

- **Line 23–28.** Mirroring the local computations on lines 21 and 22, for each remote rank r, the source particle, proxy particle, and proxy charge data in LET_r is copied to the GPU, the interaction lists $\mathcal{L}(\{C_t\}, \text{LET}_r)$ are formed, ComputePotential($\mathcal{L}(\{C_t\}, \text{LET}_r)$) is called, and the LET_r data is deleted from the GPU.

- **Line 29.** The local potential $\{\phi\}$ is incremented with the proxy potentials $\{\widehat{\phi}_{\boldsymbol{\ell}}\}$ of $\{C_t\}$ on the GPU using the downward pass. This procedure is given by Algorithm A.4.

- **Lines 30–31.** The local potential $\{\phi\}$ is copied back out to CPU memory, and all remaining data on the GPU is deleted.

A.2 BaryTree potential computation

Algorithm A.2 gives a pseudocode implementation of the BLDTT's potential computation, corresponding to lines 21 and 26 of Algorithm A.1. For p ranks, this procedure is called p times on each of the ranks: one time for the interactions between local target tree $\{C_t\}$ and local source tree $\{C_s\}$, and $p-1$ times for the interactions between $\{C_t\}$ and LET_r, once each for the $p-1$ remote ranks r.

For a given set of interaction lists $\mathcal{L}$, this procedure launches asynchronous GPU compute kernels for each (C_t, C_s) pair. The pairs with either PP or PC interactions increment the

local potential array $\{\phi\}$, and the pairs with either CP or CC interactions increment the proxy potentials $\{\widehat{\phi}_{\boldsymbol{\ell}}\}$ of $\{C_t\}$. A GPU `wait` on the last line blocks further computation until all of the launched GPU compute kernels return. This strategy allows us to saturate the GPU with as much work as possible; the GPU scheduler can determine which compute kernels to launch when to keep the device optimally busy.

Algorithm A.2 Computing potential and proxy potential contributions due to the interaction lists $\mathcal{L}$.

1: **procedure** COMPUTEPOTENTIAL(interaction lists $\mathcal{L}$)
2:     **for** each PP interaction pair (C_t, C_s) in $\mathcal{L}$ **do**
3:         GPU, async: increment potential $\{\phi\}$ with (C_t, C_s) PP (direct) interaction
4:     **end for**
5:     **for** each PC interaction pair (C_t, C_s) in $\mathcal{L}$ **do**
6:         GPU, async: increment potential $\{\phi\}$ with (C_t, C_s) PC interaction
7:     **end for**
8:     **for** each CP interaction pair (C_t, C_s) in $\mathcal{L}$ **do**
9:         GPU, async: increment proxy potential $\{\widehat{\phi}_{\boldsymbol{\ell}}\}$ of C_t with (C_t, C_s) CP interaction
10:     **end for**
11:     **for** each CC interaction pair (C_t, C_s) in $\mathcal{L}$ **do**
12:         GPU, async: increment proxy potential $\{\widehat{\phi}_{\boldsymbol{\ell}}\}$ of C_t with (C_t, C_s) CC interaction
13:     **end for**
14:     GPU, wait
15: **end procedure**

Listing A.1 shows an example OpenACC GPU compute kernel for computing a PP Coulomb interaction between a target cluster C_t and source cluster C_s, corresponding to line 3 of Algorithm A.2. Note that this procedure is essentially identical for other interaction kernels beyond Coulomb, with nothing more than some changes on lines 20–21, and caching some kernel parameters, such as κ for screened Coulomb, at the beginning of the procedure.

This procedure is also largely identical to the ones for computing PC, CP, and CC interactions. In the PC case, the source particle arrays are replaced with the source proxy particle arrays. In the CP case, the target particle arrays are replaced with the target proxy particle arrays, and the potential array is replaced with the target proxy potential array. Similarly, in the CC case, all arrays are replaced with their proxy variants.

This procedure is described in more detail, with some comments, below.

- **Lines 1–7.** This procedure takes in target particle arrays `tx`, `ty`, `tz`, corresponding to the x, y, z coordinates of the targets, and source particle arrays `sx`, `sy`, `sz`, `sq`, corresponding to the x, y, z coordinates of the sources and the charges q, as well as the local potential array `potential`, corresponding to $\{\phi\}$. `num_targets` and `num_sources` mark the number of

target and source particles contained in C_t and C_s, respectively, and `target_idx_start` and `source_idx_start` mark the target and source array indices where the particles contained in C_t and C_s begin.

```
1  void K_Coulomb_PP(int num_targets, int num_sources,
2      int target_idx_start, int source_idx_start,
3      double *tx, double *ty, double *tz,
4      double *sx, double *sy, double *sz,
5      double *sq, struct RunParams *run_params,
6      double *potential, int gpu_stream_id)
7  {
8      #pragma acc kernels async(gpu_stream_id) \
9              present(tx, ty, tz, sx, sy, sz, sq, potential)
10     {
11     #pragma acc loop gang independent
12     for (int i = 0; i < num_targets; i++) {
13         int ii = target_idx_start + i;
14         double x = tx[ii], y = ty[ii], z = tz[ii];
15         double temp_potential = 0.0;
16         #pragma acc loop vector reduction(+:temp_potential)
17         for (int j = 0; j < num_sources; j++) {
18             int jj = source_idx_start + j;
19             double dx = x - sx[jj], dy = y - sy[jj], dz = z - sz[jj];
20             double rr  = dx*dx + dy*dy + dz*dz;
21             if (rr > DBL_MIN) temp_potential += sq[jj] / sqrt(rr);
22         }
23         #pragma acc atomic
24         potential[ii] += temp_potential;
25     }
26     } // end kernel
27 }
```

Listing A.1: An OpenACC GPU compute kernel for computing the direct interaction between a target cluster and source cluster.

- **Line 8.** The `#pragma acc kernels` directive marks the beginning of a GPU compute kernel. The `async(gpu_stream_id)` clause marks that this compute kernel is launched asynchronously, in the stream given by the integer `gpu_stream_id`, which is a randomly chosen integer between 0 and 2. Asynchronous launching means that control is returned to the host CPU as soon as the kernel has been launched. Launching the compute kernel in one of three streams allows the GPU to better hide kernel launch latency; while a compute kernel in one stream is executing, a compute kernel in another stream can simultaneously

be in the process of launching or returning. Testing suggests that using three streams decreases total compute time by about 30%, and additional streams provide no more performance gain.

- **Line 9.** The `present(tx, ty, tz, sx, sy, sz, sq, potential)` clause marks that these eight arrays must already be present in the GPU's memory for this compute kernel to execute. Otherwise, the kernel will fail.

- **Lines 11–12.** The `#pragma acc loop` directive marks the beginning of a loop which will be parallelized on the GPU. This loop iterates over the targets in C_t. The `gang independent` clause marks that the following loop will be parallelized at the `gang` level, i.e., loop iterations will be mapped to thread blocks. The `independent` clause simply tells the compiler that the results of the loop iterations are in no way dependent on each other. The `independent` clause is likely not actually required, but in this case is a signal to the programmer. In fact, the compiler would likely parallelize this loop at the `gang` level even without the `gang` clause, but it's best practice to be as explicit as possible. Not every compiler is as good as the PGI/ NVIDIA HPC compiler with OpenACC.

- **Lines 13–15.** Each loop iteration over the targets stores the target coordinates in temporary local variables. Likely, the compiler would do this on its own, but again, it's good to be as explicit as possible. A temporary variable `temp_potential` is also created to store the contribution from the sources to the target's potential.

- **Lines 16–17.** This `#pragma acc loop` directive marks yet another loop which will be parallelized on the GPU. The `vector` clause marks that this loop will be parallelized at the `vector` level, i.e., loop iterations will be mapped to threads within a thread block. Note, of course, that each loop iteration of the loop on line 12 over the targets in C_t will launch an instance of this loop over the sources in C_s. The `reduction(+:temp_potential)` clause marks that each iteration of this loop increments the `temp_potential` variable. Because each loop iteration touches this variable, the `reduction` clause ensures proper loop scheduling so that the values in `temp_potential` are not overwritten.

- **Lines 18–20.** Each loop iteration over the sources computes the difference between the stored target coordinates and the source coordinates. The squared Cartesian distance between the source and target is then computed and stored in `rr`.

- **Line 21.** `temp_potential` is incremented with the contribution to the potential value at the given target due to the given source. This implementation omits singular contributions, so `temp_potential` is only incremented if the square distance `rr` is greater than the smallest

positive normal double precision number. For different interaction kernels, this is really the only block of lines that would be modified.

- **Lines 23–24.** After the completion of the inner loop nest, the `potential` for the given target is incremented with the contribution from `temp_potential`. The `#pragma acc atomic` directive ensures that only one thread at a time can update the memory location at `potential[ii]`. Since the GPU compute kernels which update the potential are launched asynchronously, the `atomic` directive is required to prevent multiple compute kernels from attempting to update `potential[ii]` at the same time.

As stated above, effectively this exact same procedure is used for PC, CP, and CC interactions. On close inspection of the implications of this, a natural question arises: why should we explicitly build out the proxy particles like the actual particles? Recall that the proxy particles are formed by a tensor product of Chebyshev points in the x, y, z directions. Instead of explicitly building the tensor product, we could just use three arrays containing the x, y, z coordinates from which the tensor product is built. For a PC interaction, this would modify lines 16–22 of Listing A.1 to the form shown in Listing A.2.

```
1  ...
2      int proxy_source_idx_start = source_cluster_idx * (degree+1);
3      int proxy_charge_idx_start = source_cluster_idx
4                                 * (degree+1) * (degree+1) * (degree+1);
5      #pragma acc loop vector collapse(3) reduction(+:temp_potential)
6      for (int kx = 0; kx < degree+1; kx++) {
7      for (int ky = 0; ky < degree+1; ky++) {
8      for (int kz = 0; kz < degree+1; kz++) {
9          double dx = x - proxy_source_x[proxy_source_idx_start + kx];
10         double dy = y - proxy_source_y[proxy_source_idx_start + ky];
11         double dz = z - proxy_source_z[proxy_source_idx_start + kz];
12         double rr  = dx*dx + dy*dy + dz*dz;
13         int kk = proxy_charge_idx_start
14                + kx * (degree+1) * (degree+1) * (degree+1)
15                + ky * (degree+1) * (degree+1) + kz;
16         if (rr > DBL_MIN) temp_potential += proxy_q[kk] / sqrt(rr);
17     }
18     }
19     }
20  ...
```

Listing A.2: A modified inner loop for PC interactions if the tensor product of the Chebyshev points for the source proxy particles were explicitly constructed.

This strategy replaces one inner loop over source proxy particles with three loops over the x, y, z cooredinates. The `collapse3` clause of the directive on line 5 essentially turns the three loops into a single loop which is parallelized at the vector level. Notice also that the proxy charges themselves cannot be decomposed in such a manner, so we have to maintain two indexing schemes, one for the coordinates, and another for the proxy charges.

My initial assumption was that this strategy would be overall more efficient, because it requires the creation of significantly less memory. The x, y, z coordinate arrays, instead of containing (number of clusters) $\times$ (degree+1)^3 entries, only contain (number of clusters) $\times$ (degree+1). This requires significantly less work to copy or fill, and data management on GPUs is always a potential source of inefficiency.

However, as it turns out, this strategy yields markedly worse performance, for one important reason. The data accesses to the x, y, z coordinate arrays are no longer *strided*. In the first implementation, there exists a one-to-one correspondence between the threads and the accesses to the proxy x, y, z, q arrays such that two threads whose IDs differ by n will also have the memory locations of their fetches differ by n. GPU memory fetches are more efficient with strided accesses like these. In the decomposed three loop case shown above, this is no longer the case, and multiple threads will attempt to access the same entries of the x, y, z arrays. Even the q array might not display strided accesses, if OpenACC's strategy for collapsing the three loops results in a different indexing strategy than the one used for the q array. The tradeoff between these costs and reduced memory management and copying turns out to not work in this strategy's favor.

An important caveat is that this strategy actually is effective for the *target* clusters, because target particles or proxy particles are iterated over in the outer loop, at the `gang` level. The memory accesses to the target particle or proxy particle arrays are much less frequent than those to the source particle or proxy particle arrays, so in this case, it is in fact advantageous to flatten the target proxy particles. However, flattening the target proxy particles while maintaining the full tensor product source proxy particle arrays introduces a significant amount of complication into the code, so pursuing this idea was abandoned. Flattening the target clusters for the cluster-particle GPU RISM treecode described in §5.3, where no source proxy particles are needed, does end up providing significant performance improvements, as shown in Appendix B.

A.3 Upward and downward passes

Algorithm A.3 gives a pseudocode implementation of the BLDTT's upward pass, corresponding to line 9 of Algorithm A.1. Level L refers to the top level of the tree, containing

only the root source cluster, and level 1 refers to the bottom level, which contains only leaf clusters. First, for each leaf cluster, the procedure asynchronously launches a GPU compute kernel that interpolates the source charges to the proxy source charges. A GPU `wait` blocks further computation until all of these launched GPU compute kernels return. Then, for each level l of the tree beginning from the second level, a GPU compute kernel is asynchronously launched for each child of each cluster on level l, incrementing the parent cluster proxy charges by interpolating the child cluster proxy charges. At each level, a GPU `wait` blocks further computation until all launched GPU compute kernels for level l return.

Algorithm A.3 The upward pass computes proxy charges of source clusters.

1: **procedure** UPWARDPASS(source particles $\{\mathbf{y}_j\}$, source clusters $\{C_s\}$)
2:     **for** each leaf cluster C_s **do**
3:         GPU, async: interpolate charges $\{q_j\}$ of $\{\mathbf{y}_j\}$ to $\{\widehat{q}_{\mathbf{k}}\}$ of C_s
4:     **end for**
5:     GPU, wait
6:     **for** level $l = 2$ to L **do**
7:         **for** each cluster C_s on level l **do**
8:             **for** each child cluster C_s^i of C_s **do**
9:                 GPU, async: increment $\{\widehat{q}_{\mathbf{k}}\}$ of C_s by interpolating $\{\widehat{q}_{\mathbf{k}}\}$ of C_s^i
10:             **end for**
11:         **end for**
12:         GPU, wait
13:     **end for**
14: **end procedure**

Algorithm A.4 gives a pseudocode implementation of the BLDTT's downward pass, corresponding to line 29 of Algorithm A.1. Note that this procedure has a very similar structure to the downward pass. For each level l of the target tree beginning from the top level L, which contains only the root target cluster, a GPU compute kernel is asynchronously launched for each child of each cluster on level l, incrementing proxy potentials of the child cluster by interpolating the parent cluster proxy potentials. At each level, a GPU `wait` blocks further computation until all launched GPU compute kernels for level l return. Then, for each leaf cluster, the procedure asynchronously launches a GPU compute kernel that increments the potential by interpolating the proxy potentials. A GPU `wait` blocks further computation until all of these launched GPU compute kernels return.

Note that leaf clusters in both the UPWARDPASS and DOWNWARDPASS procedures can actually occur on any level of the tree. While level 1 is defined to be the level containing only leaf clusters, for a sufficiently uniform distribution, all leaf clusters will be on level 1. Careful inspection of these two procedures shows that they are, in fact, agnostic to the

tree level on which the leaf clusters occur. For further details on the implementation of the GPU compute kernels launched by these procedures, refer to the GitHub repository at `github.com/Treecodes/BaryTree`.

Algorithm A.4 The downward pass increments the local potential with proxy potentials of target clusters.

1: **procedure** DownwardPass(potential $\{\phi\}$, target clusters $\{C_t\}$)
2: **for** level $l = L$ to 2 **do**
3: **for** each cluster C_t on level l **do**
4: **for** each child cluster C_t^i of target cluster C_t **do**
5: GPU, async: increment $\{\widehat{\phi}_{\boldsymbol{\ell}}\}$ of C_t^i by interpolating $\{\widehat{\phi}_{\boldsymbol{\ell}}\}$ of C_t
6: **end for**
7: **end for**
8: GPU, wait
9: **end for**
10: **for** each leaf cluster C_t **do**
11: GPU, async: increment $\{\phi\}$ by interpolating $\{\widehat{\phi}_{\boldsymbol{\ell}}\}$ of C_t
12: **end for**
13: GPU, wait
14: **end procedure**

APPENDIX B

Implementation Details of GPU-Accelerated TCF LRA Treecodes

This appendix describes some implementation details of the GPU-accelerated treecodes for computing TCF LRAs in 3D-RISM, described in §5.3, with the hopes of highlighting some lessons for future students.

Figure B.1 expands upon Fig. 5.12 by showing three intermediate improvements in performance for the 1Z7Q TCF LRA calculation from the original BaryTree implementation of the cluster-particle barycentric Lagrange treecode (CP-BLTC) (**blue**) to the heavily specialized and optimized CP-BLTC for 3D-RISM (**green**). For the five treecode versions, this figure shows relative ℓ_2 error vs total compute time (s), where connected curves represent constant MAC θ (0.7 $\times$, solid; 0.9 $\circ$, dashed), and interpolation degree $n = 1, 2, 4, 6, 8, 10$ increases from right to left on each curve (first three versions run only to $n = 8$).

Figure B.2 gives the run time component breakdown of the five codes (labeled by color) shown in Fig. B.1 for MAC $\theta = 0.9$, across interpolation degrees $n = 1, 2, 4, 6, 8$. The components shown are the target tree building (blue), in which the target particles are reordered for the hierarchical tree of target clusters; cluster building (green), in which the target proxy particles are computed; potential computation (grey), in which the potential and proxy potentials are incremented due to PP and CP interactions between cluster pairs; downpass (yellow), in which the proxy potentials are interpolated to the final potential values; particle reordering (red), in which the target particles and potentials are unordered to their original ordering, before tree building; and other (purple), including source particle batching, interaction list building, and various data movement operations. This figure shows the portion of code that benefited from each successive improvement.

These successive improvements between each code version are detailed below. The five versions are labelled by a **bolded color** corresponding to the curve colors in Figure B.1.

- **Blue to Orange.** The **blue** curves are the original CP-BLTC as implemented in BaryTree. The **orange** curves introduce a virtualized target grid. Because the target particles are on a uniform Cartesian grid, the targets do not have to be explicitly constructed. This saves

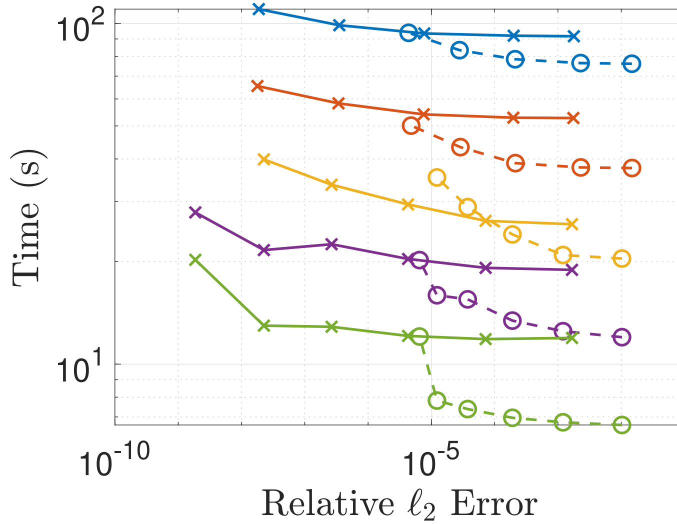


Figure B.1: 1Z7Q TCF LRA calculation, original BaryTree CP-BLTC (blue), heavily specialized CP-BLTC for 3D-RISM (green), connected curves represent constant MAC θ (0.7 $\times$, solid; 0.9 $\circ$, dashed), interpolation degree $n = 1, 2, 4, 6, 8, 10$ increases from right to left on each curve (first three versions run only to $n = 8$), simulations ran on one NVIDIA P100 GPU.

a significant amount of time in the target tree building process, because the targets do not have to be rearranged in memory. Whenever targets are needed during the downward pass and potential computation, they are generated on the fly. In Fig. B.2, this improvement completely shrinks the tree building (blue) and particle reordering (red) phases.

Additionally, the downpass is only performed on clusters whose proxy potential arrays were actually used in the potential computation. This greatly reduces unnecessary computation in the downpass phase (yellow).

- **Orange to Yellow.** Two changes were introduced from the **orange** curves to the **yellow** curves. First, the maximum leaf size parameter $M0$ was changed from 2000 to 256.

  Additionally, the target cluster proxy particles were flattened in the manner described in Listing A.1. In this case, the strategy is actually advantageous, because target cluster proxy particles are iterated over in the *outer* `gang` loop, not in the *inner* `vector` loop. Thus, there are significantly fewer non-strided memory accesses that would decrease the efficiency of the GPU compute kernel as with source cluster proxy particle flattening. Note that after virtualizing the targets, the full tensor product arrays of target cluster proxy particles become a significant majority of the total memory usage, and have significant costs associated with copying to the GPU. In Fig. B.2, this improvement completely shrinks the cluster building phase (green).

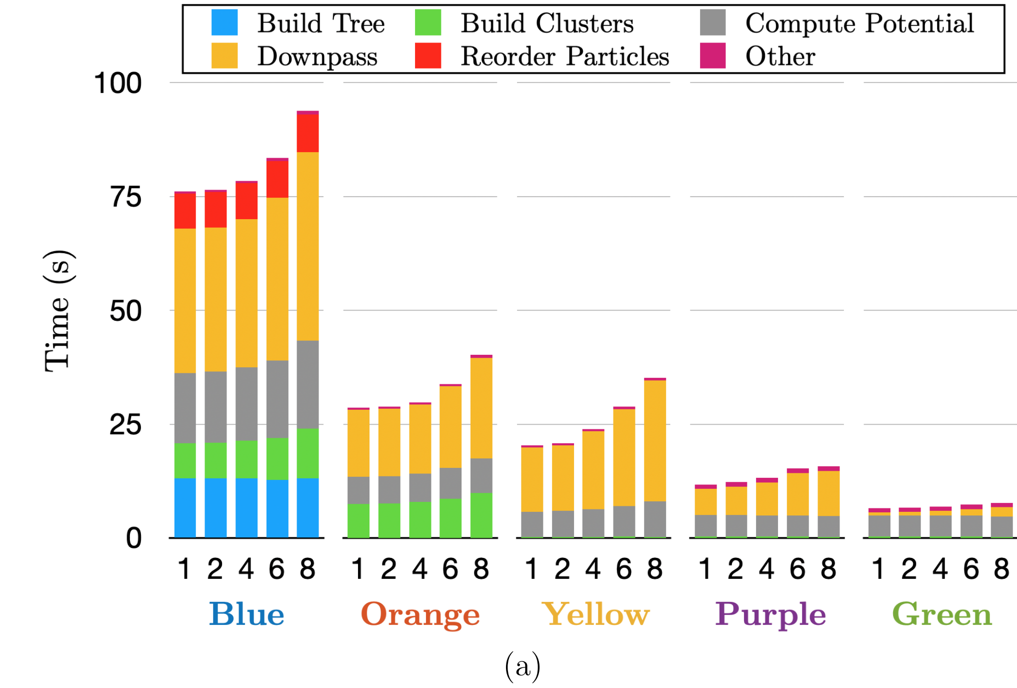
- **Yellow to Purple.** Three changes were introduced from the yellow curves to the purple curves. First, the downpass algorithm was "flattened" to take advantage of the grid structure of the targets. Since both the targets and the target proxy particles are tensor products, the downpass can be modified to calculate the interpolation coefficients in a 1D manner. Instead, for each interpolation of proxy potentials to potentials, interpolation coefficients need only be computed for the x direction, y direction, and z direction, separately.

  Second, `vector`ized loops in the downward pass which only run over a single Cartesian coordinate were modified with the `vector(32)` clause. Because the interpolation degree is almost certainly smaller than 32 (typically no bigger than 10 in practical use cases), and the vast majority of target clusters contain less than 32 grid points along a given Cartesian direction ($32 \times 32 \times 32$ corresponds to 32,768 target grid points in a cluster), the default value of 128 threads for a `vector`ized loop in OpenACC results in a significant amount of wasted computation on these loops. The `vector(32)` clause means that the loop is `vector`ized across only 32 threads. In Fig. B.2, both this and the previous improvement shrink the downpass phase (yellow).

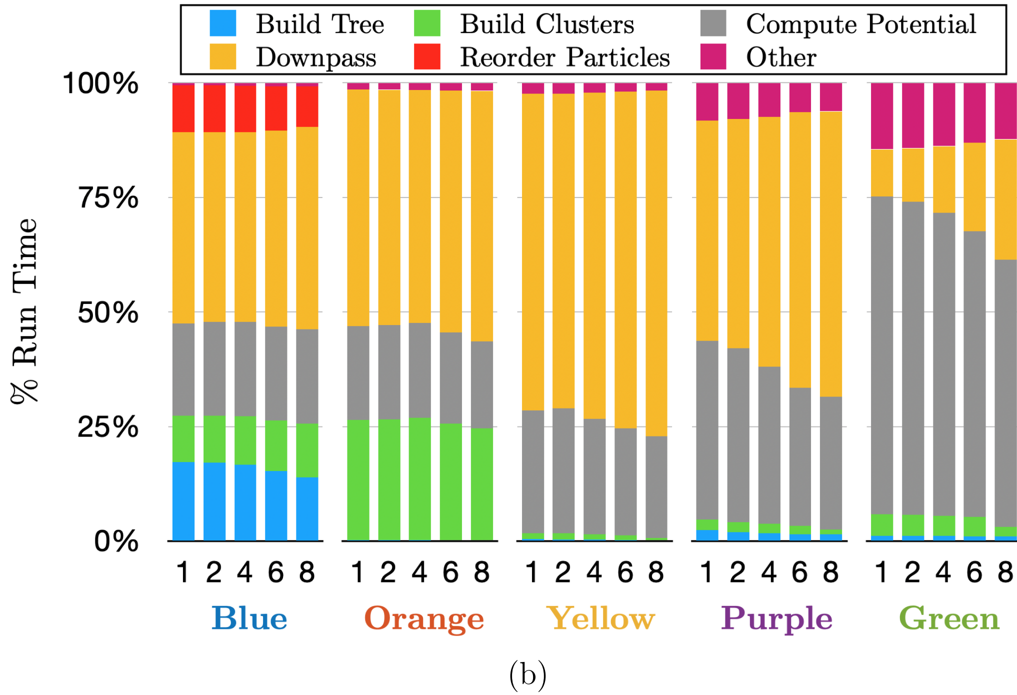
  Third, data movement is more explicitly managed. Initially, data regions were "structured" and corresponded to kernel regions, which resulted in data being moved on and off the GPU multiple times. "Unstructured" data regions minimize data movement by explicitly marking when data should be moved between host and device. This improvement decreases growth in the potential computation phase (grey) as degree increases.

- **Purple to Green.** Two changes were introduced from the purple curves to the green curves. Most importantly, the original $O(M \log M)$ downpass, in which the proxy potentials of each target cluster are directly interpolated to the potentials, was replaced with the BLDTT's $O(M)$ downpass in which the proxy potentials of each target cluster are interpolated to the proxy potentials of their child target clusters. In the actual development history of BaryTree, this was the first implementation of the $O(M)$ downward pass, which was later extended to an $O(N)$ upward pass and implementation in the BLDTT algorithm. Among all of the improvements listed in this section, this is the only algorithmic change.

  Additionally, the parts of the downpass routine which generate the interpolation coefficients were moved from the GPU to the CPU, and the parts which actually interpolate the proxy potentials remained on the GPU. The parts moved to the CPU were essentially those optimized in the previous bullet point. Essentially, the calculation of interpolation coefficients involves such small loops after flattening, that it's more efficient to compute them on the CPU and then copy the coefficients to the GPU for the downpass. In Fig. B.2, both this and the previous improvement shrink the downpass phase (yellow).



(a)



(b)

Figure B.2: Run time component breakdown of the five codes (labeled by color) shown in Fig. B.1 for MAC $\theta = 0.9$, across interpolation degrees $n = 1, 2, 4, 6, 8$, (a) absolute time breakdown, (b) proportional percentage breakdown.

APPENDIX C

Advice for Scientific Computing Software Projects

This appendix describes some general advice for building scientific software, with the hopes of highlighting some lessons for future students.

- **Use CMake to manage your build system**, and learn how to use it well. As you build out a software package, using Makefiles becomes a completely inflexible, unsustainable solution, particularly if you plan to run it on multiple machines, or make it available publicly for other users.

  In particular, use modern CMake techniques, such as explicit creation of compile targets, and never use a CMake version less than 3.0. The BaryTree and TABI-PB software packages are solid examples of how to write good modern CMake. One particularly good introduction to the topic, with links to other resources, can be found at [163].

- **Version control your work.** I have saved myself from great pain on multiple occasions by using git repos to save and version control all of my software projects. Using git, and particularly GitHub or GitLab, also makes collaborative projects significantly easier. GitHub also provides a natural way to distribute your software packages.

  To get the most out of git, think carefully about your branching policy. One common and effective strategy is to make separate branches for each feature addition, and merge them into a develop branch through pull requests. Then, only merge into the master or main branch when preparing for a release version.

  The GitHub Guides provide resources on using GitHub and git for version control [164].

- **Use C++ and modern C++ design patterns** (i.e., don't just write C-style code and call it C++). If I could do everything over, I would've used C++ for BaryTree. Using C++ to write the new version of TABI-PB was relatively easy, and created much more concise and self-documenting code.

  One particularly important C++ design pattern is RAII (Resource Acquisition Is Initialization), which essentially means that the life of an object should be bound to the life of

its resources. For the purposes of any computationally heavy software, probably the most salient example of this is the use of the `vector` class instead of C-style arrays. There is almost no good reason to use naked pointers to memory with explicit `alloc` and `free` calls when storing arrays of numbers. It's no faster than just using `vector` when actually operating on the array elements.

Also, the standard template library (STL) has many very powerful algorithms. It's likely that if you have need for, say, sorting or partitioning algorithms, highly optimized versions already exist in the STL.

- **Learn to use profilers and debuggers.** A significant number of the opportunities for optimization identified when building BaryTree, TABI-PB, and the treecodes for RISM were identified using profiling tools. Building out the MPI distributed memory parallelization of BaryTree would have been nearly impossible without a parallel debugger.

  One great set of tools is ARM Forge, previously known as Allinea, which includes the DDT debugger and the MAP profiler [165]. While these tools can be used effectively for serial applications, they are particularly useful for MPI parallelized applications. Note that many MPI bugs or scaling bottlenecks may not become immediately apparently until the number of ranks increases well beyond two!

  For profiling GPU kernels targeting NVIDIA machines, whether the kernels were written with OpenACC directives or CUDA (or possibly OpenMP), NVIDIA Nsight Compute is the best tool available. `nvprof` is a great command line tool for quick and dirty profiling of GPU compute kernels. Nsight Systems is a tool for system-wide performance analysis of applications scaling any number of CPUs or GPUs. I have little experience with this tool, but have heard good things. More on the NVIDIA tools is available at [166].

- **MPI and OpenMP are essential tools for scaling your application.** Understand the complementary roles of MPI and OpenMP in building an HPC application. MPI targets distributed memory systems across multiple compute nodes, while OpenMP provides a high level method for multithreading a program across a shared memory machine.

  A common design pattern is to use one MPI rank per socket or compute node, and within each MPI rank, use one OpenMP thread per CPU core in the socket or node. Note that it's possible, for some applications, that using one MPI rank per core and no multithreading is the most efficient solution, so experimenting is imperative.

  Also, understanding OpenMP thread binding and affinity and its interaction with MPI parallelization is important to achieving high performance.

BIBLIOGRAPHY

[1] Joshua E. Barnes and Piet Hut. A hierarchical $O(N \log N)$ force-calculation algorithm. *Nature*, 324:446–449, 1986.

[2] Zhong-Hui Duan and Robert Krasny. An adaptive treecode for computing nonbonded potential energy in classical molecular systems. *J. Comput. Chem.*, 22(2):184–195, 2001.

[3] Zhong-Hui Duan and Robert Krasny. A treecode algorithm for computing Ewald summation of dipolar systems. *Proc. 2003 ACM Symp. Appl. Comput. (SAC)*, pages 172–177, 2003.

[4] Peijun Li, Hans Johnston, and Robert Krasny. A Cartesian treecode for screened Coulomb interactions. *J. Comput. Phys.*, 228:3858–3868, 2009.

[5] Nathan Vaughn. *GPU Accelerated Barycentric Treecodes and their Application to Kohn-Sham Density Functional Theory.* PhD thesis, University of Michigan, 2020.

[6] Nathan Vaughn, Leighton Wilson, and Robert Krasny. A GPU-accelerated barycentric Lagrange treecode. *Proc. 21st IEEE Int. Workshop Parallel Distrib. Sci. Eng. Comput. (PDSEC 2020)*, pages 701–710, 2020.

[7] Leighton Wilson, Nathan Vaughn, and Robert Krasny. A GPU-accelerated fast summation method based on barycentric Lagrange interpolation and dual tree traversal. *arXiv e-prints*, page arXiv:2012.06925, 2020.

[8] Benoît Roux and Thomas Simonson. Implicit solvent models. *Biophys. Chem.*, 78:1–20, 1999.

[9] Zhe Zhang, Shawn Witham, and Emil Alexov. On the role of electrostatics in protein–protein interactions. *Phys. Biol.*, 8(3):035001, 2011.

[10] Jacopo Tomasi. Thirty years of continuum solvation chemistry: a review, and prospects for the near future. *Theor. Chem. Acc.*, 112:184–203, 2004.

[11] Nathan A. Baker. Poisson–Boltzmann methods for biomolecular electrostatics. In Ludwig Brand and Michael L. Johnson, editors, *Numerical Computer Methods, Part D*, volume 383 of *Methods Enzymol.*, chapter 5, pages 94–118. Academic Press, 1 edition, 2004.

[12] Benzhuo Lu, Y. C. Zhou, Michael J. Holst, and J. Andrew McCammon. Recent progress in numerical methods for the Poisson–Boltzmann equation in biophysical applications. *Commun. Comput. Phys.*, 3(5):973–1009, 2008.

[13] Michael J. Holst and Faisal Saied. Numerical solution of the nonlinear Poisson–Boltzmann equation: developing more robust and efficient methods. *J. Comput. Chem.*, 16(3):337–364, 1995.

[14] Nathan A. Baker, David Sept, Simpson Joseph, Michael J. Holst, and J. Andrew McCammon. Electrostatics of nanosystems: application to microtubules and the ribosome. *Proc. Natl. Acad. Sci. U.S.A.*, 98(18):10037–10041, 2001.

[15] Ray Luo, Laurent David, and Michael K. Gilson. Accelerated Poisson–Boltzmann calculations for static and dynamic systems. *J. Comput. Chem.*, 23:1244–1253, 2002.

[16] Jun Wang and Ray Luo. Assessment of linear finite-difference Poisson–Boltzmann solvers. *J. Comput. Chem.*, 31(8):1689–1698, 2010.

[17] Minxin Chen and Benzhuo Lu. TMSmesh: a robust method for molecular surface mesh generation using a trace technique. *J. Chem. Theory Comput.*, 7(1):203–212, 2011.

[18] Alexander H. Boschitsch and Marcia O. Fenley. A fast and robust Poisson–Boltzmann solver based on adaptive Cartesian grids. *J. Comput. Chem.*, 7(5):1524–1540, 2011.

[19] Weihua Geng and Shan Zhao. Fully implicit ADI schemes for solving the nonlinear Poisson–Boltzmann equation. *Mol. Based Math. Biol.*, 1:109–123, 2012.

[20] Leighton Wilson and Shan Zhao. Unconditionally stable time splitting methods for the electrostatic analysis of solvated biomolecules. *Int. J. Numer. Anal. Model.*, 13(6):852–878, 2016.

[21] Michael J. Holst, Nathan A. Baker, and F. Wang. Adaptive multilevel finite element solution of the Poisson–Boltzmann equation I: Algorithms and examples. *J. Comput. Chem.*, 21(15):1319–1342, 2000.

[22] Nathan A. Baker, Michael J. Holst, and F. Wang. Adaptive multilevel finite element solution of the Poisson–Boltzmann equation II: Refinement at solvent-accessible surfaces in biomolecular systems. *J. Comput. Chem.*, 21(15):1343–1352, 2000.

[23] Jie Liang and Shankar Subramaniam. Computation of molecular electrostatics with boundary element methods. *Biophys. J.*, 73(4):1830–1841, 1997.

[24] André H. Juffer, Eugen F. F. Botta, Bert A. M. van Keulen, Auke van der Ploeg, and Herman J. C. Berendsen. The electric potential of a macromolecule in a solvent: a fundamental approach. *J. Comput. Phys.*, 97:144–171, 1991.

[25] Alexander H. Boschitsch, Marcia O. Fenley, and Huan-Xiang Zhou. Fast boundary element method for the linear Poisson–Boltzmann equation. *J. Phys. Chem. B*, 106:2741–2754, 2002.

[26] Weihua Geng and Robert Krasny. A treecode-accelerated boundary integral Poisson–Boltzmann solver for electrostatics of solvated biomolecules. *J. Comput. Phys.*, 247:62–78, 2013.

[27] Christopher D. Cooper, Jaydeep P. Bardhan, and Lorena A. Barba. A biomolecular electrostatics solver using Python, GPUs and boundary elements that can handle solvent-filled cavities and Stern layers. *Comput. Phys. Commun.*, 185(3):720–729, 2014.

[28] Weihua Geng. A boundary integral Poisson–Boltzmann solver package for solvated bimolecular simulations. *Mol. Based Math. Biol.*, 3:43–58, 2015.

[29] Elizabeth Jurrus, Dave Engel, Keith Star, Kyle Monson, Juan Brandi, Lisa E. Felberg, David H. Brookes, Leighton Wilson, Jiahui Chen, Karina Liles, Minju Chen, Peter Li, David W. Gohara, Todd Dolinsky, Robert Konecny, David R. Koes, Jens E. Nielsen, Teresa Head-Gordon, Weihua Geng, Robert Krasny, Guo Wei Wei, Michael J. Holst, J. Andrew McCammon, and Nathan A. Baker. Improvements to the APBS biomolecular solvation software suite. *Protein Sci.*, 27(1):112–128, 2017.

[30] L. S. Ornstein and F. Zernike. Accidental deviations of density and opalescence at the critical point of a single substance. *Proc. Akad. Sci. (Amsterdam)*, 17:793, 1914.

[31] Dmitrii Beglov and Benoît Roux. An integral equation to describe the solvation of polar molecules in liquid water. *J. Phys. Chem. B*, 101(39):7821–7826, 1997.

[32] Andriy Kovalenko and Fumio Hirata. Self-consistent description of a metal–water interface by the Kohn–Sham density functional theory and the three-dimensional reference interaction site model. *J. Chem. Phys.*, 110(20):10095–10112, 1999.

[33] J Johnson, D A Case, T Yamazaki, S Gusarov, A Kovalenko, and T Luchko. Small molecule hydration energy and entropy from 3D-RISM. *J. Phys. Condens. Matter*, 28(34):344002, 2016.

[34] David S. Palmer, Andrey I. Frolov, Ekaterina L. Ratkova, and Maxim V. Fedorov. Towards a universal method for calculating hydration free energies: a 3D reference interaction site model with partial molar volume correction. *J. Phys. Condens. Matter*, 22(49):492101, 2010.

[35] Volodymyr P. Sergiievskyi, Guillaume Jeanmairet, Maximilien Levesque, and Daniel Borgis. Fast computation of solvation free energies with molecular density functional theory: Thermodynamic-ensemble partial molar volume corrections. *J. Phys. Chem. Lett.*, pages 1935–1942, 2014.

[36] Jean-François Truchon, B. Montgomery Pettitt, and Paul Labute. A cavity corrected 3D-RISM functional for accurate solvation free energies. *J. Chem. Theory Comput.*, 10(3):934–941, 2014.

[37] Henry A. Boateng and Robert Krasny. Comparison of treecodes for computing electrostatic potentials in charged particle systems with disjoint targets and sources. *J. Comput. Chem.*, 34(25):2159–2167, 2013.

[38] Leighton Wilson, Tyler Luchko, and Robert Krasny. Accelerating the 3D-RISM theory of molecular solvation with treecode summation and cut-offs. In preparation.

[39] D.A. Case, I.Y. Ben-Shalom, S.R. Brozell, D.S. Cerutti, T.E. Cheatham III, V.W.D. Cruzeiro, T.A. Darden, R.E. Duke, D. Ghoreishi, G. Giambaşu, T. Giese, M.K. Gilson, H. Gohlke, A.W. Goetz, D. Greene, R. Harris, N. Homeyer, S. Izadi, A. Kovalenko, R. Krasny, T. Kurtzman, T.S. Lee, S. LeGrand, P. Li, C. Lin, J. Liu, T. Luchko, R. Luo, V. Man, D.J. Mermelstein, K.M. Merz, Y. Miao, G. Monard, C. Nguyen, H. Nguyen, A. Onufriev, F. Pan, R. Qi, D.R. Roe, A. Roitberg, C. Sagui, S. Schott-Verdugo, J. Shen, C.L. Simmerling, J. Smith, J. Swails, R.C. Walker, J. Wang, H. Wei, L. Wilson, R.M. Wolf, X. Wu, L. Xiao, Y. Xiong, D.M. York, and P.A. Kollman. AMBER 2019. University of California, San Francisco, 2019.

[40] R. W. Hockney and J. W. Eastwood. *Computer Simulation Using Particles*. Taylor & Francis, 1988.

[41] U. Essmann, L. Perera, M. Berkowitz, T. Darden, H. Lee, and L. Pederson. A smooth particle mesh Ewald method. *J. Chem. Phys.*, 103:8577–8593, 1995.

[42] Andrew W. Appel. An efficient program for many-body simulation. *SIAM J. Sci. Stat. Comput.*, 6(1):85–103, 1985.

[43] Leslie Greengard and Vladimir Rokhlin. A fast algorithm for particle simulations. *J. Comput. Phys.*, 73:325–348, 1987.

[44] H. Cheng, Leslie Greengard, and Vladimir Rokhlin. A fast adaptive multipole algorithm in three dimensions. *J. Comput. Phys.*, 155(2):468–498, 1999.

[45] W. Hackbusch and Z. P. Nowak. On the fast matrix multiplication in the boundary element method by panel clustering. *Numer. Math.*, 54:463–491, 1989.

[46] Wolfgang Hackbusch. *Hierarchical Matrices: Algorithms and Analysis*. Spring, 2015.

[47] David J. Hardy, Zhe Wu, James C. Phillips, John E. Stone, Robert D. Skeel, and Klaus Schulten. Multilevel summation method for electrostatic force evaluation. *J. Chem. Theory Comput.*, 11:766–779, 2015.

[48] Keith Lindsay and Robert Krasny. A particle method and adaptive treecode for vortex sheet motion in three-dimensional flow. *J. Comput. Phys.*, 172(2):879–907, 2001.

[49] Walter Dehnen. A hierarchical $O(N)$ force calculation algorithm. *J. Comput. Phys.*, 179(1):27–42, 2002.

[50] Klaas Esselink. The order of Appel's algorithm. *Inf. Process. Lett.*, 41(3):141–147, 1992.

[51] Michael S. Warren and John K. Salmon. A portable parallel particle program. *Comput. Phys. Commun.*, 87(1–2):266–290, 1995.

[52] Shang-Hua Teng. Provably good partitioning and load balancing algorithms for parallel adaptive N-body simulation. *SIAM J. Sci. Comput.*, 19(2):635–656, 1998.

[53] Rio Yokota. An FMM based on dual tree traversal for many-core architectures. *J. Algorithm Comput. Technol.*, 7(3):301–324, 2013.

[54] Kenjiro Taura, Jun Nakashima, Rio Yokota, and Naoya Maruyama. A task parallel implementation of fast multipole methods. In *2012 SC Companion: High Performance Computing, Networking Storage and Analysis*, pages 617–625, 2012.

[55] Walter Dehnen. A fast multipole method for stellar dynamics. *Comput. Astrophys. Cosmol.*, 1(1):1–23, 2014.

[56] Benoit Lange and Pierre Fortin. Parallel dual tree traversal on multi-core and many-core architectures for astrophysical N-body simulations. *Proc. 20th Int. Eur. Conf. Parallel Distrib. Comput. (Euro-Par 2014)*, pages 716–727, 2014.

[57] Konstantin Lorenzen, Magnus Schwörer, Philipp Tröster, Simon Mates, and Paul Tavan. Optimizing the accuracy and efficiency of fast hierarchical multipole expansions for md simulations. *J. Chem. Theory Comput.*, 8(10):3628–3636, 2012.

[58] Jonathan Coles and Michel Masella. The fast multipole method and point dipole moment polarizable force fields. *J. Chem. Phys.*, 142(2):024109, 2015.

[59] Pierre Fortin and Maxime Touche. Dual tree traversal on integrated GPUs for astrophysical N-body simulations. *Int. J. High Perform. Comput. Appl.*, pages 1–13, 2019.

[60] Leslie Greengard and Jingfang Huang. A new version of the fast multipole method for screened Coulomb interactions in three dimensions. *J. Comput. Phys.*, 180(2):642 – 658, 2002.

[61] Christopher R. Anderson. An implementation of the fast multipole method without multipoles. *SIAM J. Sci. Stat. Comput.*, 13:923–947, 1992.

[62] Junichiro Makino. Yet another fast multipole method without multipoles–pseudoparticle multipole method. *J. Comput. Phys.*, 151:910–920, 1999.

[63] Lexing Ying, George Biros, and Denis Zorin. A kernel-independent adaptive fast multipole algorithm in two and three dimensions. *J. Comput. Phys.*, 196(2):591–626, 2004.

[64] William Fong and Eric Darve. The black-box fast multipole method. *J. Comput. Phys.*, 228(23):8712–8725, 2009.

[65] Jean-Paul Berrut and Lloyd N. Trefethen. Barycentric Lagrange interpolation. *SIAM Rev.*, 46(3):501–517, 2004.

[66] Lei Wang, Robert Krasny, and Svetlana Tlupova. A kernel-independent treecode based on barycentric Lagrange interpolation. *Commun. Comput. Phys.*, 28:1415–1436, 2020.

[67] Léopold Cambier and Eric Darve. Fast low-rank kernel matrix factorization using skeletonized interpolation. *SIAM J. Sci. Comput.*, 41(3):A1652–A1680, 2019.

[68] Xin Xing and Edmond Chow. Interpolative decomposition via proxy points for kernel matrices. *SIAM J. Matrix Anal. Appl.*, 41(1):221–243, 2020.

[69] Hua Huang, Xin Xing, and Edmond Chow. H2Pack: High-performance $\mathcal{H}^2$ matrix package for kernel matrices using the proxy point method. *ACM Trans. Math. Softw.*, 2020.

[70] Lexing Ying, George Biros, Denis Zorin, and Harper Langston. A new parallel kernel-independent fast multipole method. *Proc. 2003 ACM/IEEE Conf. Supercomput. (SC03)*, page 14, 2003.

[71] Ilya Lashuk, Aparna Chandramowlishwaran, M. Harper Langston, Tuan-Anh Nguyen, Rahul Sampath, Aashay Shringarpure, Richard Vuduc, Lexing Ying, Denis Zorin, and George Biros. A massively parallel adaptive fast multipole method on heterogeneous architectures. *Commun. ACM*, 55:101–109, 05 2012.

[72] William B. March, Bo Xiao, Sameer Tharakan, Chenhan D. Yu, and George Biros. A kernel-independent FMM in general dimensions. *Proc. Int. Conf. High Perf. Comput. Networking, Storage Anal. (SC15)*, pages 24:1–24:12, 2015.

[73] Dhairya Malhotra and George Biros. Pvfmm: A parallel kernel independent FMM for particle and volume potentials. *Commun. Comput. Phys.*, 18(3):808–830, 2015.

[74] Dhairya Malhotra and George Biros. Algorithm 967: A distributed-memory fast multipole method for volume potentials. *ACM Trans. Math. Softw.*, 43:1–27, 2016.

[75] Toru Takahashi, Cris Cecka, and Eric Darve. Optimization of the parallel black-box fast multipole method on CUDA. *2012 Innov. Parallel Comput. (InPar)*, pages 1–14, May 2012.

[76] NVIDIA Corporation. Kepler GK110/210 Whitepaper, 2014. Available at `nvidia.com/content/dam/en-zz/Solutions/Data-Center/tesla-product-literature/NVIDIA-Kepler-GK110-GK210-Architecture-Whitepaper.pdf`.

[77] Erich Elsen, M Houston, Vikrant Vishal, Eric Darve, P Hanrahan, and Vijay Pande. N-body simulation on GPUs. *Proc. 2006 ACM/IEEE Conf. Supercomput. (SC06)*, page 188, 2006.

[78] Lars Nyland, M Harris, and Jan Prins. Fast N-body simulation with CUDA. *GPU Gems, Vol. 3*, pages 677–695, 2009.

[79] Hu Jiang and Qianni Deng. Barnes–Hut treecode on GPU. *2010 IEEE Int. Conf. Prog. Inf. Comput. (PIC)*, 2:974–978, 2010.

[80] Jeroen Bédorf, Evghenii Gaburov, and Simon Portegies Zwart. A sparse octree gravitational n-body code that runs entirely on the GPU processor. *J. Comput. Phys.*, 231(7):2825–2839, 2012.

[81] Jeroen Bédorf, Evghenii Gaburov, Michiko S. Fujii, Keigo Nitadori, Tomoaki Ishiyama, and Simon Portegies Zwart. 24.77 Pflops on a gravitational tree-code to simulate the Milky Way Galaxy with 1860 GPUs. *Proc. Int. Conf. High Perf. Comput. Networking, Storage Anal. (SC14)*, pages 54–65, 2014.

[82] Tsuyoshi Hamada, Tetsu Narumi, Rio Yokota, Kenji Yasuoka, Keigo Nitadori, and Makato Taiji. 42 TFlops hierarchical N-body simulations on GPUs with applications in both astrophysics and turbulence. *Proc. Int. Conf. High Perf. Comput. Networking, Storage Anal. (SC09)*, pages 1–12, 2009.

[83] Martin Burtscher and Keshav Pingali. An efficient CUDA implementation of the tree-based Barnes Hut n-body algorithm. In Wen-Mei Hwu, editor, *GPU Computing Gems Emerald Edition*, chapter 6, pages 75–92. Morgan Kaufmann/Elsevier, 2011.

[84] Rio Yokota and Lorena A. Barba. Comparing the treecode with FMM on GPUs for vortex particle simulations of a leapfrogging vortex ring. *Comput. Fluids.*, 45(1):155–161, 2011.

[85] Herbert E. Salzer. Lagrangian interpolation at the Chebyshev points $x_{n,\nu} \equiv \cos(\nu\pi/n), \nu = 0(1)n$; some unnoted advantages. *Comput. J.*, 15:156–159, 1972.

[86] M. S. Warren and J. K. Salmon. Astrophysical N-body simulations using hierarchical tree data structures. *Proc. 1992 ACM/IEEE Conf. Supercomput. (SC92)*, pages 570–576, 1992.

[87] The Zoltan Project Team. The Zoltan Project, 2020. Available at `trilinos.github.io/zoltan.html`.

[88] E. G. Boman, U. V. Catalyurek, C. Chevalier, and K. D. Devine. The Zoltan and Isorropia parallel toolkits for combinatorial scientific computing: Partitioning, ordering, and coloring. *Sci. Program.*, 20(2):129–150, 2012.

[89] H. C. Plummer. On the problem of distribution in globular star clusters. *Mon. Not. R. Astr. Soc.*, 71:460–470, 1911.

[90] H. Dejonghe. A completely analytical family of anisotropic Plummer models. *Mon. Not. R. Astr. Soc.*, 224:13–39, 1987.

[91] John Towns, Timothy Cockerill, Maytal Dahan, Ian Foster, Kelly Gaither, Andrew Grimshaw, Victor Hazlewood, Scott Lathrop, Dave Lifka, Gregory D. Peterson, Ralph Roskies, J. Ray Scott, and Nancy Wilkins-Diehr. XSEDE: Accelerating scientific discovery. *Comput. Sci. Eng.*, 16(5):62–74, 2014.

[92] Henry A. Boateng. *Cartesian treecode algorithms for electrostatic interactions in molecular dynamics simulations*. PhD thesis, University of Michigan, 2010.

[93] Leighton Wilson and Robert Krasny. Comparison of the MSMS and NanoShaper molecular surface triangulation codes in the TABI Poisson–Boltzmann solver. *arXiv e-prints*, page arXiv:2010.10570, 2020.

[94] Leighton Wilson, Jingzhen Hu, Jiahui Chen, Robert Krasny, and Weihua Geng. Computation of electrostatic binding free energy with the TABI-PB solver. In preparation.

[95] Frederic M. Richards. Areas, volumes, packing, and protein structure. *Annu. Rev. Biophys. Bioeng.*, 6:151–176, 1977.

[96] M. L. Connolly. Molecular surface triangulation. *J. Appl. Crystallogr.*, 18:499–505, 1985.

[97] Sergio Decherchi, José Colmenares, Chiara Eva Catalano, Michela Spagnuolo, Emil Alexov, and Walter Rocchia. Between algorithm and model: different molecular surface definitions for the Poisson–Boltzmann based electrostatic characterization of biomolecules in solution. *Commun. Comput. Phys.*, 13:61–89, 2013.

[98] Tiantian Liu, Minxin Chen, and Benzhuo Lu. Parameterization for molecular Gaussian surface and a comparison study of surface mesh generation. *J. Mol. Model.*, 21(5):113, 2015.

[99] Michel F. Sanner, Arthur J. Olson, and Jean-Claude Spehner. Fast and robust computation of molecular surfaces. *Proc. 11th ACM Symp. Comput. Geom. (SoCG)*, pages C6–C7, 1995.

[100] Sergio Decherchi and Walter Rocchia. A general and robust ray-casting-based algorithm for triangulating surfaces at the nanoscale. *PLoS ONE*, 8(4):e59744, 2013.

[101] Todd J. Dolinsky, Jens E. Nielsen, J. Andrew McCammon, and Nathan A. Baker. PDB2PQR: an automated pipeline for the setup, execution, and analysis of Poisson–Boltzmann electrostatics calculations. *Nucleic Acids Res.*, 32(Web Server):W665–W667, 2004.

[102] Tolga Can, Chao-I Chen, and Yuan-Fang Wang. Efficient molecular surface generation using level-set methods. *J. Mol. Graph. Model.*, 25:442–454, 2006.

[103] Robert C. Harris, Alexander H. Boschitsch, and Marcia O. Fenley. Influence of grid spacing in Poisson–Boltzmann equation binding energy estimation. *J. Chem. Theory Comput.*, 9(8):3677–3685, 2013. PMID: 23997692.

[104] Robert C. Harris, Travis Mackoy, and Marcia O. Fenley. Problems of robustness in Poisson–Boltzmann binding free energies. *J. Chem. Theory Comput.*, 11(2):705–712, 2015. PMID: 26528091.

[105] Duc D. Nguyen, Bao Wang, and Guo Wei Wei. Accurate, robust, and reliable calculations of Poisson–Boltzmann binding energies. *Nature Commun.*, 38(13):941–948, 2017.

[106] Duan Chen, Zhan Chen, Changjun Chen, Weihua Geng, and Guo Wei Wei. MIBPB: A software package for electrostatic analysis. *J. Comput. Chem.*, 32:756–770, 2011.

[107] Y. C. Zhou, Shan Zhao, Michael Feig, and Guo Wei Wei. High order matched interface and boundary method for elliptic equations with discontinuous coefficients and singular sources. *J. Comput. Phys.*, 213(1):1–30, 2006.

[108] Sining Yu, Weihua Geng, and Guo Wei Wei. Treatment of geometric singularities in implicit solvent models. *J. Chem. Phys.*, 126:244108, 2007.

[109] Weihua Geng, Sining Yu, and Guo Wei Wei. Treatment of charge singularities in implicit solvent models. *J. Chem. Phys.*, 127(11):114106, 2007.

[110] Guilherme Duarte Ramos Matos, Daisy Y. Kyu, Hannes H. Loeffler, John D. Chodera, Michael R. Shirts, and David L. Mobley. Approaches for calculating solvation free energies and enthalpies demonstrated with an update of the FreeSolv database. *J. Chem. Eng. Data*, 62(5):1559–1569, 2017.

[111] George M. Giambaşu, Tyler Luchko, Daniel Herschlag, Darrin M. York, and David A. Case. Ion counting from explicit-solvent simulations and 3D-RISM. *Biophys. J.*, 106:883–894, 2014.

[112] Zoe Cournia, Bryce Allen, and Woody Sherman. Relative binding free energy calculations in drug discovery: Recent advances and practical considerations. *J. Chem. Inf. Model.*, 57(12):2911–2937, 2017.

[113] R. Evans. The nature of the liquid-vapour interface and other topics in the statistical mechanics of non-uniform, classical fluids. *Adv. Phys.*, 28(2):143–200, 1979.

[114] Yu Liu, Shuangliang Zhao, and Jianzhong Wu. A site density functional theory for water: Application to solvation of amino acid side chains. *J. Chem. Theory Comput.*, 9(4):1896–1908, 2013.

[115] Shuangliang Zhao, Rosa Ramirez, Rodolphe Vuilleumier, and Daniel Borgis. Molecular density functional theory of solvation: From polar solvents to water. *J. Chem. Phys.*, 134(19):194102, 2011.

[116] Tyler Luchko, Sergey Gusarov, Daniel R. Roe, Carlos Simmerling, David A. Case, Jack Tuszynski, and Andriy Kovalenko. Three-dimensional molecular theory of solvation coupled with molecular dynamics in Amber. *J. Chem. Theory Comput.*, 6(3):607–624, 2010.

[117] Sergey Gusarov, Tom Ziegler, and Andriy Kovalenko. Self-consistent combination of the three-dimensional RISM theory of molecular solvation with analytical gradients and the Amsterdam density functional package. *J. Phys. Chem. A*, 110(18):6083–6090, 2006.

[118] Norio Yoshida and Fumio Hirata. A new method to determine electrostatic potential around a macromolecule in solution from molecular wave functions. *J. Comput. Chem.*, 27(4):453–462, 2006.

[119] Thomas Kloss, Jochen Heil, and Stefan M. Kast. Quantum chemistry in solution by combining 3D integral equation theory with a cluster embedding approach. *J. Phys. Chem. B*, 112(14):4337–4343, 2008.

[120] Tatsuhiko Miyata and Fumio Hirata. Combination of molecular dynamics method and 3D-RISM theory for conformational sampling of large flexible molecules in solution. *J. Comput. Chem.*, 29(6):871–882, 2008.

[121] Andriy Kovalenko and Fumio Hirata. Potentials of mean force of simple ions in ambient aqueous solution. ii. solvation structure from the three-dimensional reference interaction site model approach, and comparison with simulations. *J. Chem. Phys.*, 112(23):10403–10417, 2000.

[122] Jean-Pierre Hansen and Ian R. McDonald. *Theory of Simple Liquids*. Elsevier, 4th edition, 2013.

[123] Tyler Luchko, In Suk Joung, and David A. Case. Integral equation theory of biomolecules and electrolytes. In Tamar Schlick, editor, *Innovations in Biomolecular Modeling and Simulations: Volume 1*, volume 23 of *RSC Biomolecular Sciences*, chapter 4, pages 51–86. Royal Society of Chemistry, 2012.

[124] D.A. Case, I.Y. Ben-Shalom, S.R. Brozell, D.S. Cerutti, T.E. Cheatham III, V.W.D. Cruzeiro, T.A. Darden, R.E. Duke, D. Ghoreishi, M.K. Gilson, H. Gohlke, A.W. Goetz, D. Greene, R. Harris, N. Homeyer, S. Izadi, A. Kovalenko, T. Kurtzman, T.S. Lee, S. LeGrand, P. Li, C. Lin, J. Liu, T. Luchko, R. Luo, D.J. Mermelstein, K.M. Merz, Y. Miao, G. Monard, C. Nguyen, H. Nguyen, I. Omelyan, A. Onufriev, F. Pan, R. Qi, D.R. Roe, A. Roitberg, C. Sagui, S. Schott-Verdugo, J. Shen, C.L. Simmerling, J. Smith, R. Salomon-Ferrer, J. Swails, R.C. Walker, J. Wang, H. Wei, R.M. Wolf, X. Wu, L. Xiao, D.M. York, and P.A. Kollman. AMBER 2018. University of California, San Francisco, 2018.

[125] Zhong-Hui Duan and Robert Krasny. An Ewald summation based multipole method. *J. Chem. Phys.*, 113(9):3492–3495, 2000.

[126] D. A. Case, D. S. Cerutti, T. E. Cheatham, III, T. A. Darden, R. E. Duke, T. J. Giese, H. Gohlke, A. W. Goetz, D. Greene, N. Homeyer, S. Izadi, A. Kovalenko, T. S. Lee, S. LeGrand, P. Li, C. Lin, J. Liu, T. Luchko, R. Luo, D. Mermelstein, K. M. Merz, G. Monard, H. Nguyen, I. Omelyan, A. Onufriev, F. Pan, R. Qi, D. R. Roe, A. Roitberg, C. Sagui, C. L. Simmerling, W. M. Botello-Smith, J. Swails, R. C. Walker, J. Wang, R. M. Wolf, X. Wu, L. Xiao, D. M. York, and P. A. Kollman. AMBER 2017. University of California, San Francisco, 2017.

[127] Noel M. O'Boyle, Michael Banck, Craig A. James, Chris Morley, Tim Vandermeersch, and Geoffrey R. Hutchison. Open Babel: An open chemical toolbox. *J. Cheminformatics*, 3(1):33, oct 2011.

[128] Junmei Wang, Romain M. Wolf, James W. Caldwell, Peter A. Kollman, and David A. Case. Development and testing of a general Amber force field. *J. Comput. Chem.*, 25(9):1157–1174, 2004.

[129] Araz Jakalian, Bruce L. Bush, David B. Jack, and Christopher I. Bayly. Fast, efficient generation of high-quality atomic charges. AM1-BCC model: I. Method. *J. Comput. Chem.*, 21(2):132–146, 2000.

[130] Hari S. Muddana, Andrew T. Fenley, David L. Mobley, and Michael K. Gilson. The SAMPL4 host–guest blind prediction challenge: an overview. *J. Comput. Aided Mol. Des.*, 28(4):305–317, 2014.

[131] Christopher I. Bayly, Piotr Cieplak, Wendy Cornell, and Peter A. Kollman. A well-behaved electrostatic potential based method using charge restraints for deriving atomic charges: the RESP model. *J. Phys. Chem.*, 97(40):10269–10280, 1993.

[132] François-Yves Dupradeau, Adrien Pigache, Thomas Zaffran, Corentin Savineau, Rodolphe Lelong, Nicolas Grivel, Dimitri Lelong, Wilfried Rosanski, and Piotr Cieplak. The R.E.D. tools: advances in RESP and ESP charge derivation and force field library building. *Phys. Chem. Chem. Phys.*, 12(28):7821–7839, 2010.

[133] Enguerran Vanquelef, Sabrina Simon, Gaelle Marquant, Elodie Garcia, Geoffroy Klimerak, Jean Charles Delepine, Piotr Cieplak, and François-Yves Dupradeau. R.E.D. Server: a web service for deriving RESP and ESP charges and building force field libraries for new molecules and molecular fragments. *Nucleic Acids Res.*, 39:W511–W517, 2011.

[134] F. Wang, J.-P. Becker, Piotr Cieplak, and François-Yves Dupradeau. R.E.D. Python: Object oriented programming for Amber force fields, 2013.

[135] M. J. Frisch, G. W. Trucks, H. B. Schlegel, G. E. Scuseria, M. A. Robb, J. R. Cheeseman, G. Scalmani, V. Barone, G. A. Petersson, H. Nakatsuji, X. Li, M. Caricato, A. Marenich, J. Bloino, B. G. Janesko, R. Gomperts, B. Mennucci, H. P. Hratchian, J. V. Ortiz, A. F. Izmaylov, J. L. Sonnenberg, D. Williams-Young, F. Ding, F. Lipparini, F. Egidi, J. Goings, B. Peng, A. Petrone, T. Henderson, D. Ranasinghe, V. G. Zakrzewski, J. Gao, N. Rega, G. Zheng, W. Liang, M. Hada, M. Ehara, K. Toyota, R. Fukuda, J. Hasegawa, M. Ishida, T. Nakajima, Y. Honda, O. Kitao, H. Nakai, T. Vreven, K. Throssell, J. A. Montgomery, Jr., J. E. Peralta, F. Ogliaro, M. Bearpark, J. J. Heyd, E. Brothers, K. N. Kudin, V. N. Staroverov, T. Keith, R. Kobayashi, J. Normand, K. Raghavachari, A. Rendell, J. C. Burant, S. S. Iyengar, J. Tomasi, M. Cossi, J. M. Millam, M. Klene, C. Adamo, R. Cammi, J. W. Ochterski, R. L. Martin, K. Morokuma, O. Farkas, J. B. Foresman, and D. J. Fox. Gaussian 09. Gaussian, Inc.

[136] Christian Tiede, Anna A. S. Tang, Sarah E. Deacon, Upasana Mandal, Joanne E. Nettleship, Robin L. Owen, Suja E. George, David J. Harrison, Raymond J. Owens, Darren C. Tomlinson, and Michael J. McPherson. Adhiron: a stable and versatile peptide display scaffold for molecular recognition applications. *Protein Eng. Des. Sel.*, 27(5):145–155, 2014.

[137] James A. Maier, Carmenza Martinez, Koushik Kasavajhala, Lauren Wickstrom, Kevin E. Hauser, and Carlos Simmerling. ff14SB: Improving the accuracy of protein side chain

and backbone parameters from ff99SB. *J. Chem. Theory Comput.*, 11(8):3696–3713, 2015.

[138] James H. Nettles, Huilin Li, Ben Cornett, Joseph M. Krahn, James P. Snyder, and Kenneth H. Downing. The binding mode of epothilone A on α,ß-tubulin by electron crystallography. *Science*, 305(5685):866–869, 2004.

[139] Raimond B.G. Ravelli, Benoît Gigant, Patrick A. Curmi, Isabelle Jourdain, Sylvie Lachkar, André Sobel, and Marcel Knossow. Insight into tubulin regulation from a complex with colchicine and a stathmin-like domain. *Nature*, (6979):198–202, 2004.

[140] Andrej Šali and Tom L. Blundell. Comparative protein modelling by satisfaction of spatial restraints. *J. Mol. Biol.*, 234(3):779–815, 1993.

[141] Pengfei Li, Benjamin P. Roberts, Dhruva K. Chakravorty, and Kenneth M. Merz. Rational design of particle mesh Ewald compatible Lennard–Jones parameters for +2 metal cations in explicit solvent. *J. Chem. Theory Comput.*, 9(6):2733–2748, 2013.

[142] H. J. C. Berendsen, J. R. Grigera, and T. P. Straatsma. The missing term in effective pair potentials. *J. Phys. Chem.*, 91(24):6269–6271, 1987.

[143] In Suk Joung and Thomas E. Cheatham. Determination of alkali and halide monovalent ion parameters for use in explicitly solvated biomolecular simulations. *J. Phys. Chem. B*, 112(30):9020–9041, 2008.

[144] John Perkyns and B. Montgomery Pettitt. A site–site theory for finite concentration saline solutions. *J. Chem. Phys.*, 97(10):7656–7666, 1992.

[145] Andriy Kovalenko, Seiichiro Ten-no, and Fumio Hirata. Solution of three-dimensional reference interaction site model and hypernetted chain equations for simple point charge water by modified method of direct inversion in iterative subspace. *J. Comput. Chem.*, 20(9):928–936, 1999.

[146] Edgar Gabriel, Graham E. Fagg, George Bosilca, Thara Angskun, Jack J. Dongarra, Jeffrey M. Squyres, Vishal Sahay, Prabhanjan Kambadur, Brian Barrett, Andrew Lumsdaine, Ralph H. Castain, David J. Daniel, Richard L. Graham, and Timothy S. Woodall. Open MPI: Goals, Concept, and Design of a Next Generation MPI Implementation. In Dieter Kranzlmüller, Péter Kacsuk, and Jack Dongarra, editors, *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, Lecture Notes in Computer Science, pages 97–104, Berlin, Heidelberg, 2004. Springer.

[147] Matteo Frigo and Steven G. Johnson. The design and implementation of FFTW3. *Proc. IEEE*, 93(2):216–231, 2005.

[148] Andreas Förster, Eugene I. Masters, Frank G. Whitby, Howard Robinson, and Christopher P. Hill. The 1.9 Å structure of a proteasome-11S activator complex and implications for proteasome-PAN/PA700 interactions. *Mol. Cell*, 18(5):589–599, 2005.

[149] Robert Krasny and Lei Wang. A treecode based on barycentric Hermite interpolation for electrostatic particle interactions. *Comput. Math. Biophys.*, 7:73–84, 2019.

[150] Nathan Vaughn, Vikram Gavini, and Robert Krasny. Treecode-accelerated Green iteration for Kohn–Sham density functional theory. *J. Comput. Phys.*, 430:110101, 2021.

[151] Chunhu Tan, Yu-Hong Tan, and Ray Luo. Implicit nonpolar solvent models. *J. Phys. Chem. B*, 111(42):12263–12274, 2007.

[152] Jason A. Wagoner and Nathan A. Baker. Assessing implicit models for nonpolar mean solvation forces: The importance of dispersion and volume terms. *Proc. Natl. Acad. Sci. U.S.A.*, 103(22):8331–8336, 2006.

[153] Robert A. Pierotti. A scaled particle theory of aqueous and nonaqueous solutions. *Chem. Rev.*, 76(6):717–726, 1976.

[154] John D. Weeks, David Chandler, and Hans C. Anderson. Role of repulsive forces in determining the equilibrium structure of simple liquids. *J. Chem. Phys.*, 54(12):5237–5247, 1971.

[155] Wei Wang and Peter A. Kollman. Free energy calculations on dimer stability of the HIV protease using molecular dynamics and a continuum solvent model. *J. Mol. Biol.*, 303(4):567–582, November 2000.

[156] Nadine Homeyer and Holger Gohlke. Free energy calculations by the molecular mechanics Poisson–Boltzmann surface area method. *Mol. Inform.*, 31:114–122, 2012.

[157] Samuel Genheden and Ulf Ryde. The MM/PBSA and MM/GBSA methods to estimate ligand-binding affinities. *Expert Opin. Drug Discov.*, 10(5):449–461, 2015.

[158] Changhao Wang, Peter H. Nguyen, Kevin Pham, Danielle Huynh, Thanh-Binh Nancy Le, Hongli Wang, Pengyu Ren, and Ray Luo. Calculating protein-ligand binding affinities with MMPBSA: Method and error analysis. *J. Comput. Chem.*, 37:2436–2446, 2016.

[159] Changhao Wang, D'Artagnan Greene, Li Xiao, Ruxi Qi, and Ray Luo. Recent developments and applications of the MMPBSA method. *Front. Biosci.*, 4:87, 2018.

[160] Mark James Abraham, Teemu Murtolad, Roland Schulz, Szilárd Páll, Jeremy C. Smith, Berk Hess, and Erik Lindahl. GROMACS: High performance molecular simulations through multi-level parallelism from laptops to supercomputers. *SoftwareX*, 1–2:19–25, 2015.

[161] Rashmi Kumari, Rajendra Kumar, Open Source Drug Discovery Consortium, and Andrew Lynn. g_mmpbsa–a GROMACS tool for high-throughput MM-PBSA calculations. *J. Chem. Inf. Model.*, 54:1951–1962, 2014.

[162] Cheng Yung-Chi and William H. Prusoff. Relationship between the inhibition constant (K_I) and the concentration of inhibitor which causes 50 per cent inhibition (I_{50}) of an enzymatic reaction. *Biochem. Pharmacol.*, 22(23):3099–3108, 1973.

[163] Henry Schreiner. An introduction to modern CMake, 2021. Available at `cliutils.gitlab.io/modern-cmake`.

[164] GitHub, Inc. GitHub Guides, 2021. Available at `guides.github.com`.

[165] Arm Ltd. Arm Forge, 2021. Available at `https://developer.arm.com/tools-and-software/server-and-hpc/debug-and-profile/arm-forge`.

[166] NVIDIA Corporation. NVIDIA Developer Tools, 2021. Available at `developer.nvidia.com`.