Communication Reducing Approaches and Shared-Memory Optimizations for the Hierarchical Fast Multipole Method on Distributed and Many-core Systems

Dissertation by
Mustafa Abduljabbar

In Partial Fulfillment of the Requirements
For the Degree of
Doctor of Philosophy

King Abdullah University of Science and Technology
Thuwal, Kingdom of Saudi Arabia

November, 2018
© November, 2018
Mustafa Abduljabbar
All Rights Reserved
EXAMINATION COMMITTEE PAGE

The dissertation of Mustafa Abduljabbar is approved by the examination committee.

Committee Chairperson: Prof. David Keyes
Committee Members: Prof. Hakan Bagci, Prof. Markus Hadwiger, Prof. William Gropp
ABSTRACT

Communication Reducing Approaches and Shared-Memory Optimizations for the Fast Multipole Method on Distributed and Many-core Systems

Mustafa Abduljabbar

We present algorithms and implementations that overcome obstacles in the migration of the Fast Multipole Method (FMM), one of the most important algorithms in computational science and engineering, to exascale computing. Emerging architectural approaches to exascale computing are all characterized by data movement rates that are slow relative to the demand of aggregate floating point capability, resulting in performance that is bandwidth limited. Practical parallel applications of FMM are impeded in their scaling by irregularity of domains and dominance of collective tree communication, which is known not to scale well. We introduce novel ideas that improve partitioning of the $N$-body problem with boundary distribution through a sampling-based mechanism that hybridizes two well-known partitioning techniques, Hashed Octree (HOT) and Orthogonal Recursive Bisection (ORB). To reduce communication cost, we employ two methodologies. First, we directly utilize features available in parallel runtime systems to enable asynchronous computing and overlap it with communication. Second, we present Hierarchical Sparse Data Exchange (HSDX), a new all-to-all algorithm that inherently relieves communication by relaying sparse data in a few steps of neighbor exchanges. HSDX exhibits superior scalability and improves relative performance compared to the default MPI_alltoall and other relevant literature implementations. We test this algorithm alongside others on a Cray XC40 tightly coupled with the Aries network and on Intel Many Integrated Core Architecture (MIC) represented by Intel Knights Corner (KNC) and
Intel Knights Landing (KNL) as modern shared-memory CPU environments. Tests include comparisons of thoroughly tuned handwritten versus auto-vectorization of FMM Particle-to-Particle (P2P) and Multipole-to-Local (M2L) kernels. Scalability of task-based parallelism is assessed with FMM’s tree traversal kernel using different threading libraries. The MIC tests show large performance gains after adopting the prescribed techniques, which are inevitable in a world that is moving towards many-core parallelism.
ACKNOWLEDGEMENTS

I present my sincere and heart-felt gratitude to

Allah, who made me who I am, and gave me both wisdom and patience to deliver this piece of work.

My father, Abdulmajeed, who surrounded me with love and continuous prayers, and my supportive brothers, Mohammad, Ahmad, Mahmoud, and my loving sister, Dima.

My mother, Feryal, may she rest in peace, whose memory and values gave me reasons to stay determined.

My wife and lifetime friend, Ataa Al-lahham, who was wonderful enough to support me during all the ups and downs of the Ph.D. journey, and my little sons, Abdulmajeed and Owais, whose smiles and gestures added colors and flavors to my life.

Mohammad Al Farhan for his remarkable contributions in many parts of this work, and all my ECRC fellows who shared their valuable thoughts and expertise.

My second advisor, Prof. Rio Yokota whose ideas and deep insight enhanced my knowledge and helped me through many obstacles.

Last but not least, my first advisor and source of inspiration, Prof. David Keyes, whose mesmerizing knowledge, wisdom and virtues cannot be put in words.
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Copyright</strong></td>
<td>2</td>
</tr>
<tr>
<td><strong>Examination Committee Page</strong></td>
<td>3</td>
</tr>
<tr>
<td><strong>Abstract</strong></td>
<td>4</td>
</tr>
<tr>
<td><strong>Acknowledgements</strong></td>
<td>6</td>
</tr>
<tr>
<td><strong>List of Abbreviations</strong></td>
<td>11</td>
</tr>
<tr>
<td><strong>List of Figures</strong></td>
<td>12</td>
</tr>
<tr>
<td><strong>List of Tables</strong></td>
<td>15</td>
</tr>
<tr>
<td><strong>1 Introduction</strong></td>
<td>16</td>
</tr>
<tr>
<td>1.1 FMM in Computational Science</td>
<td></td>
</tr>
<tr>
<td>1.1.1 Sample FMM Matrices (Green’s Functions)</td>
<td>19</td>
</tr>
<tr>
<td>1.1.2 Factorization and Complexity</td>
<td>21</td>
</tr>
<tr>
<td>1.2 Applications of the N-body Method</td>
<td>22</td>
</tr>
<tr>
<td>1.3 Literature Review</td>
<td>23</td>
</tr>
<tr>
<td>1.3.1 FMM and N-body Algorithms</td>
<td>25</td>
</tr>
<tr>
<td>1.3.2 Runtime Systems and Skeletons</td>
<td>27</td>
</tr>
<tr>
<td>1.3.3 Communication Reduction</td>
<td>29</td>
</tr>
<tr>
<td>1.4 Objectives and Contributions</td>
<td>31</td>
</tr>
<tr>
<td><strong>2 Direct N-body Kernel on Many-core Architecture</strong></td>
<td>32</td>
</tr>
<tr>
<td>2.1 Direct N-body Code</td>
<td>32</td>
</tr>
<tr>
<td>2.2 SIMD and Shared-Memory Approaches</td>
<td>33</td>
</tr>
<tr>
<td>2.3 Task-based Traversal of ExaFMM</td>
<td>34</td>
</tr>
<tr>
<td>2.4 NUMA-aware Communication Reducing Algorithms</td>
<td>38</td>
</tr>
<tr>
<td>2.5 Results and Discussions</td>
<td>39</td>
</tr>
<tr>
<td>2.5.1 SIMD Optimizations of the N-body Kernel</td>
<td>40</td>
</tr>
<tr>
<td>2.5.2 Dual Tree Traversal with Task-Based Threading</td>
<td>44</td>
</tr>
</tbody>
</table>
## 6 FMM-Accelerated Wave Scattering Solver on Emerging HPC Architectures

### 6.1 Introduction

### 6.2 Background

- **6.2.1 BIE Formulation**
- **6.2.2 Discretization via Nyström Method**
- **6.2.3 FMM as an Accelerator for the BIE**

### 6.3 BEMFMM System Architecture

### 6.4 Thread-level Parallelism

### 6.5 Distributed-memory Optimizations

- **6.5.1 Partitioning and Load Balancing**
- **6.5.2 Communication Reduction**

### 6.6 Experimental Setup and Workload Characterization

- **6.6.1 Software Stack and Hardware Configuration**
- **6.6.2 Dataset Description**
- **6.6.3 Experimental Setup**

### 6.7 Performance Evaluation Results

- **6.7.1 Data-level Parallelism Results**
- **6.7.2 Thread-level Parallelism Results**
- **6.7.3 Distributed-memory Parallelism Results**
- **6.7.4 Convergence Aspects and Numerical Error**

### 6.8 Concluding Remarks and Future Work

## 7 Concluding Remarks

### 7.1 Summary

### 7.2 Future Work

## References

## Appendices

### A.1 Helmholtz Equation

- **A.1.1 Simple Acoustics**
- **A.1.2 Green’s Function of the Helmholtz Equation**
- **A.1.3 Divergence Theorem**
- **A.1.4 Green’s Integral Theorem and Formula**

### A.2 Fast Multipole for Oscillatory Kernels

- **A.2.1 Room Acoustics for Monopole Source**
### LIST OF ABBREVIATIONS

\[ I \text{ Sources} \]
\[ J \text{ Targets} \]

- HSDX: Hierarchical Sparse Data Exchange
- NBX: Non-blocking Exchange
- AVX: Advanced Vector Extensions
- BIE: Boundary Integral Equation
- BSP: Bulk Synchronous Parallel
- BVP: Boundary Value Problem
- DAG: Direct Acyclic Graph
- FFT: Fast Fourier Transform
- FT: Fourier Transform
- HOT: Hashed Octree
- IFFT: Inverse Fast Fourier Transform
- KNC: Intel Knights Corner
- KNL: Intel Knights Landing
- L2L: Local-to-Local
- L2P: Local-to-Particle
- LET: Local Essential Tree
- M2L: Multipole-to-Local
- M2M: Multipole-to-Multipole
- MIC: Intel Many Integrated Core Architecture
- MPI: Message Passing Interface
- ORB: Orthogonal Recursive Bisection
- P2M: Particle-to-Multipole
- P2P: Particle-to-Particle
- PDE: Partial Differential Equation
- RDMA: Remote Direct Memory Access
- SIMD: Single Instruction Multiple Data
- SSE: Streaming SIMD Extensions
- STAPL: Standard Template Adaptive Parallel Library
## LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Direct $N$-body kernel running on two Intel architectures, KNL (quadrant cluster and flat memory modes) and Broadwell [Problem Size: 1 million particles].</td>
</tr>
<tr>
<td>2.2</td>
<td>FLOP/s performance across the 2 Intel many-core generations, KNL (quadrant cluster and flat memory modes) and KNC.</td>
</tr>
<tr>
<td>2.3</td>
<td>Dual tree traversal using different task threading models. [Problem Size: 100 million particles].</td>
</tr>
<tr>
<td>2.4</td>
<td>Strong scaling of FMM tree traversal on KNL vs. Haswell using different number of MPI processes [Problem Size: 100 million particles].</td>
</tr>
<tr>
<td>2.5</td>
<td>Comparison of different MPI communication algorithms of LET communication kernel running on KNL (quadrant cluster/flat memory modes) [Problem Size: 80 million particles].</td>
</tr>
<tr>
<td>2.6</td>
<td>Comparison of different cluster modes of KNL running Distgraph communication algorithm with 64 MPI ranks and 1 TBB Thread per MPI rank [problem size is 80 million particles].</td>
</tr>
<tr>
<td>3.1</td>
<td>Illustration of the flow of FMM calculation, and the interaction between source and target particles.</td>
</tr>
<tr>
<td>3.2</td>
<td>Schematic of different partitioning schemes. (a) shows the hashed octree with Morton keys. (b) shows the hashed octree with Hilbert keys. (c) shows the orthogonal recursive bisection with an underlying global tree. (d) is the present method using an orthogonal recursive bisection with independent local trees and tight bounding boxes.</td>
</tr>
<tr>
<td>3.3</td>
<td>Histogram-based partitioning scheme.</td>
</tr>
<tr>
<td>3.4</td>
<td>A Hilbert partition from a boundary spherical distribution viewed from different perspectives. A space discontinuity exists even though partitions are in correct Hilbert order due to the existence of hollow space in orthogonal dimensions.</td>
</tr>
<tr>
<td>3.5</td>
<td>Shows the different granularity at which the partitioning phase can take place.</td>
</tr>
</tbody>
</table>
3.6 Schematic of the local essential tree (LET) for conventional HOT/ORB partitioning and present method. 59

4.1 A generic bulk synchronous Charm++-mapped version of global communication. 62

4.2 An example for the composition in the STAPL Skeleton Framework: inner-product. 66

4.3 The graphical representation of the bucket-sort skeleton’s port-mappings. 68

4.4 Different distributions of particles. 73

4.5 Strong scaling for $N = 10^8$ particles. 75

4.6 Breakdown of strong scaling for $N = 10^8$ particles. 77

4.7 Load-balance for the Plummer distribution. 78

4.8 Different distributions of particles. $N = 10^8$ on 1,025 (not 1,024) cores. 79

4.9 Performance comparison of class D of NAS EP benchmark with the reference implementation. 80

4.10 Weak scaling of FMM using MPI vs Skeleton-based Implementations. 81

4.11 Strong scaling of FMM using MPI vs Skeleton-based Implementations. 82

5.1 The different granularity at which the FMM communication can take place. The bulk-synchronous model can be viewed as an extreme case of communication aggregation. 86

5.2 Implemented communication patterns in FMM as visualized by Extrae. 87

5.3 The underlying data exchange graph of central process (3,3) within a uniform 2D grid of processes. 88

5.4 Average communication time for different sizes and distributions as grain-size is varied. 92

5.5 Strong scaling across different communication protocols with $10^{10}$ particles. 93

5.6 Communication scaling for big and small examples. 95

5.7 Strong scaling $2^{31}$ unknowns for sphere distribution and $P = 4$ and comparing exaFMM while using HSDX communication and PVFMM. 97

6.1 FMM workflow. 105

6.2 Particles in a spherical shell 2D domain. 106

6.3 BEMFMM’s system overview. 107

6.4 Geometric FMM tree partitioning. 110
6.5 Granularity of partitioning. ............................................... 113
6.6 How LET is grafted. .......................................................... 114
6.7 HSDX: A neighborhood-based communication protocol. .......................... 115
6.8 Used curved-triangle meshes. .................................................. 117
6.9 Floating point performance of the P2P Helmholtz kernel with different
vectorization means. ............................................................ 120
6.10 Tuning the threading parameters performance of the traversal Helmholtz
kernel. Performance is normalized by the arithmetic mean of the runtime. 121
6.11 Tuning the threading parameters performance of the traversal Helmholtz
kernel. Performance is normalized by the runtime and the size of the
L1/L2/L3 caches on Skylake, and the size of the L1/L2 caches on KNL. 122
6.12 Scalability results on 1,024 compute nodes of Shaheen. Performance is
normalized by the FMM time per linear iteration and the total number
of GMRES iterations. ........................................................... 124
6.13 Communication reduction and balancing ..................................... 125
6.14 Breakdown of computation and communication times across 2 different
problem sizes on a single SKL node and 4,096 nodes of Shaheen XC40.
\[ \lambda = 5h, P = 30 \text{ and } \epsilon_{FMM} = 1.0 e^{-63}. \] .......................... 129
6.15 Incident solution velocity field. ............................................. 131
6.16 Weak and strong scalability results on Shaheen from 1 to 6,144 Intel
Haswell compute nodes. Performance is normalized by the FMM time
per linear iteration and the total number of GMRES iterations. Numbers along the graph lines (red) indicate efficiency with respect to the
ideal speedup (efficiency baseline is the smallest compute node count). 135
A.1 Variable transformation by Duffy’s method. ................................ 170
A.2 Convergence effects of self- and near-singularity treatments. .......... 176
LIST OF TABLES

2.1 MPI-Based Communication Paradigms ........................................ 39
4.1 Provides skeletons and compositional operators .......................... 64
4.2 The grammar for the compositional operators .............................. 66
4.3 The grammar for the mini-language used by the compose operator. 68
4.4 FMM parameters ..................................................................... 72
5.1 *HSDX* Algorithm communication symbols ............................... 89
5.2 Communication complexity of FMM ........................................... 90
5.3 *HSDX* strong scalability analysis with FMM ............................ 91
6.1 BEMFMM control parameters ................................................... 109
6.2 Hardware specifications ............................................................ 116
6.3 Spherical mesh dataset descriptions ......................................... 118
6.4 Weak Scalability results on Shaheen from 1 to 4,096 Intel Haswell compute nodes. $T_{mv}$ is the single MatVec time. FMM is configured with $e^{-03}$ accuracy. $\epsilon_{solution}$ is the solution error, whereas $\epsilon_{far}$ is the scattered field’s error at $\infty$ when sampling 361 points at $(\theta = [0^\circ : 0.5^\circ : 180^\circ])$.
[Note: In the last two cases, we configure GMRES with max_iter=860 to avoid running out of core hours, since the solver takes great deal of time-to-solution in order to converge.] ............................................. 126
6.5 Characteristics of Shaheen’s XC40 dragonfly network .................. 127
6.6 Convergence results of various problem sizes and wavelengths on a single Intel Skylake node. $\alpha$ is the number of iterations without quadrature refinement, as opposed to $\beta$ ............................................ 130
6.7 Multiple flower shape [Figure 6.8(b)] self-convergence results for various problem sizes and a fixed wavelength on a single Intel Skylake node. Convergence is achieved when $\|\bar{b} - Ax\|_2 \leq 1.0e^{-6}$ .............................................. 131
Chapter 1

Introduction

The \( N \)-body problem is a kernel in many scientific simulations where the behavior of the system is defined from mutual interactions between discrete entities (e.g., molecules, charges, astrophysical bodies). The \( N \)-body algorithm sums up contributions due to all particles on one another and results in quadratic complexity. The Barnes-Hut treecode, which subdivides the 2D or 3D domain in quadtree or octrees, respectively, brings the complexity down to \( O(N \log N) \) by hierarchically clustering the sources into multipole expansions. FMM clusters the targets into local expansions to bring the complexity further down to \( O(N) \). For mathematical foundations of the multipole expansions, see [2], [3], and [4]. The direct \( N \)-body kernel calculates the all-pairs interaction of \( N \)-bodies against \( N \)-bodies, which results in an operation count of \( O(N^2) \). Fast approximation methods that use hierarchical domain decomposition of the bodies, along with truncated series expansions of the kernel can drive the operation count lower to \( O(N \log N) \) or even \( O(N) \) as described by Barnes et al. in [5]. The rationale behind these fast \( N \)-body methods is that bodies in the far-field need not be considered individually, but can be grouped into multipoles. However, bodies in the near-field must still be calculated accurately by using the direct \( N \)-body kernel. Therefore, the performance of the direct \( N \)-body kernel is critical for these fast \( N \)-body methods, as will be presented in Chapter 2. Among the applications of FMM are [6] and [7] where protein-protein encounters within a biomolecular dynamics solver are accelerated by using FMM to solve the boundary integral equation, which is used to discretize the linearized Poisson-Boltzmann equation. In [8] all-atom
molecular dynamics is performed to simulate the conditions of living cells by calculating energy at target proteins in a solvent and a molecular crowder using FMM. It is also used to speedup the matrix-vector multiplication, which arises from electromagnetic scattering problems [9]. Other applications include celestial dynamics simulations [10] [11].

When applying data-driven execution models to parallel hierarchical $N$-body methods, it is important first to understand the dynamic load-balancing and data prefetching mechanisms that have existed in them for over two decades. Parallel $N$-body methods start by partitioning the particles in a way that maximizes data locality while balancing the workload among the partitions. This is done by using the workload from the previous time step as weights when splitting a space filling curve that connects all particles [12]. Parallel $N$-body methods also have a mechanism for prefetching the data on remote processes by communicating all necessary parts of the remote trees up front. The resulting tree is a subset of the entire global tree, which is called the Local Essential Tree (LET) [12]. Any data-driven execution model that provides features such as dynamic load-balancing and data prefetching/caching must augment these existing tailored mechanisms rather than compete with them.

Due to its increased importance in large-scale simulations, there is now a considerable literature on implementing parallel hierarchical $N$-body solvers. Also, since FMM is among Berkeley’s seven dwarfs, the numerical methods that are believed to be the most impactful in science and engineering according to [13], it is important to address issues arising at exascale especially the increasing cost of data movement (through memory hierarchy or network) as opposed to floating point operations. Even though many of the current FMM implementations are scalable to the full machine they run on, a communication reducing approach that works on at least an order of magnitude more nodes tends to be rarely the emphasis of these implementations. This tendency is justified in accordance to the trend in enhancing a node with multi/many-core
capabilities. However, even within a many-core node, more sophisticated methods should be used to place and exchange data to get the maximum performance reported by the vendor. This is already implied in equipping the second generation of Intel® Xeon Phi™ processors code-named Knights Landing (KNL) with memory ‘clustering modes’. Therefore, ideas presented in this thesis complement the literature although they mainly target distributed memory. Chapter 2 visits exaFMM, the open source FMM library, in light of highly tuned shared-memory parallelization and detailed performance analysis on the new highly parallel Intel many-core architecture, Knights Landing (KNL). The importance of Chapter 2 stems from the fact that many-core optimizations are essential for achieving performance worthy of anticipated exascale systems, and utilization of many-core chips is inevitable to attain the desired floating point performance of these energy-austere systems.

Moreover, Chapter 2 assesses scalability and performance gain using task-based parallelization of the FMM tree traversal. It also provides an in-depth analysis of the most computationally intensive part of the traversal kernel (i.e., P2P), by comparing its performance across KNL and Broadwell architectures. Chapter 2 quantifies different configurations that exploit the on-chip 512-bit vector units within different task-based threading paradigms. Message Passing Interface (MPI) communication-reducing and NUMA-aware approaches for the FMM’s global tree data exchange are examined with different cluster modes of KNL. By applying several algorithm- and architecture-aware optimizations for FMM, we show that the N-body kernel on 256 threads of KNL achieves on average 2.8x speedup compared to the non-vectorized version, whereas on 56 threads of Broadwell, it achieves on average 2.9x speedup. In addition, the tree traversal kernel on KNL scales monotonically up to 256 threads with task-based programming models. The MPI-based communication-reducing algorithms show expected improvements of the data locality across the KNL on-chip network.
Also, due to the increased cost of moving data to different levels of system memory relative to the cost of a floating point operation \[13\], enhancing data locality is inevitable. Chapter 3 shows that the conventional wisdom of using space-filling curve partitioning may not work well for boundary integral problems, which comprise a significant portion of FMMs application user base. This chapter proposes an alternative method that modifies orthogonal recursive bisection to relieve the cell-partition misalignment that has kept it from scaling previously.

To reduce synchrony in distributed and shared environments, the effect of two well-known runtime systems, namely STAPL and Charm++, on the scalability of FMM is explored in Chapter 4. A skeleton representation (high-level description) on top of STAPL has been benchmarked using FMM, Fast Fourier Transform (FFT) and others to implicitly exploit non-blocking execution paths of the underlying code.

Last but not least, in the network of a supercomputer, a cluster or a many-core chip, reduction of communication is a key ingredient to achieving scalability in hierarchical N-body algorithms like FMM. Hence, several strategies to deal with communication volume, complexity and locality are proposed in Chapter 5. Firstly, control over granularity of communication is supported at user-level to find the optimal balance between a bulk-synchronous collective communication of the local essential tree and a Remote Direct Memory Access (RDMA) per task per cell depending on source distribution. Finally, the dynamic sparse data exchange proposed by Hoefler et al. \[15\] is extended it to a hierarchical sparse data exchange, which is demonstrated at scale to be faster than the commonly used MPI library’s MPI_Alltoallv.

1.1 FMM in Computational Science

FMM can be used to accelerate particular matrix vector products arising from discretization of partial differential equations. The general matrix vector product form is \[ \sum_{j=1}^{N} m_{ij}x_j. \] Hence an algorithm that speeds up matrix-vector product is a fast
The goal is to solve the linear system $Ax = b$ where $A$ is $N \times N$ matrix, $x$ is an $N$-vector and $b$ is an $N$ vector. Direct solutions can be reached using iterative methods requiring $O(N^3)$ (Gauss Elimination, LU Decomposition, SVD, etc.). They should converge in $k \ll N$ steps, where each step needs a matrix vector multiply, typically $O(N^2)$. FMM cuts down memory and compute complexity to $O(N)$ for specific matrices.

Fast algorithms are used for structured dense matrices, whose entries depend on only $O(N)$ parameters. For example, many matrices that behave like the Fourier matrix can use [FFT and Inverse Fast Fourier Transform (IFFT)] These matrices appear with special structure such as Circulant, Toeplitz, Hankel, Vandermonde Matrices. Such matrices are diagonalized using the FFT in $O(N \log N)$ steps then a diagonal matrix-vector multiply can be done in linear time.

FMM speeds up matrix-vector products in the form

$$s(y_j) = \sum_{j=1}^{N} a_i \Phi(x_i, y_j)$$  \hspace{1cm} (1.1)$$

This summation typically takes $O(MN)$ operations. FMM takes this complexity down to $O(M + N)$ operations for a given precision $\epsilon$. FMM works on loosely structured matrices depending on $x_i$, $y_j$, and $\phi$. One approach is to use analytical manipulation of series for faster summation. FMM introduces approximate evaluation, and bounds the error analytically, by expressing functions in an appropriate functional space using a given basis, and manipulating series to achieve approximate evaluation.

FMM can be used if the integral equation in

$$\int k(x, y)u(x)dx = \sum_{j=1}^{N} k(x_j, y)u(x_j)w_j$$  \hspace{1cm} (1.2)$$
where \(k(x, y)\) is the kernel function. In matrix form, \(1.2\) can be rewritten as

\[
v = \Phi u
\]

where

\[
\Phi = \begin{pmatrix}
\Phi(y_1, x_1) & \Phi(y_1, x_2) & \ldots & \Phi(y_1, x_N) \\
\Phi(y_2, x_1) & \Phi(y_2, x_2) & \ldots & \Phi(y_2, x_N) \\
\vdots & \vdots & \ddots & \vdots \\
\Phi(y_M, x_1) & \Phi(y_M, x_2) & \ldots & \Phi(y_M, x_N)
\end{pmatrix}
\]

\[
Y = \{y_1, y_2, \ldots, y_M\}, y_j \in \mathbb{R}^d, j = 1, \ldots, M
\]

\[
X = \{x_1, x_2, \ldots, x_N\}, x_i \in \mathbb{R}^d, i = 1, \ldots, N
\]

\[
v_j = \sum_{i=1}^{N} u_i \Phi(y_j, x_i), j = 1, \ldots, M.
\]

We call \(X\) the source points and \(Y\) the target points.

### 1.1.1 Sample FMM Matrices (Green’s Functions)

A Green’s function is the matrix part of integrand of the Boundary Integral Equation (BIE) representing the solution of certain elliptic Partial Differential Equations (PDEs). For fundamental details about Green’s functions, refer to [16, 17]. Examples include, but are not restricted to, the following:

- Green’s functions for the Laplace and Helmholtz equations

\[
\Phi(y, x) = \frac{1}{4\pi|y - x|}
\]

\[
\Phi(y, x) = \frac{e^{ik|y - x|}}{4\pi|y - x|}
\]

where \(k\) is the wave number.
- Vorticity field of an element locate at $x_i$ when we consider the spatial properties of the vorticity field at a fixed time $t$. For more details, refer to [18].

$$\Phi(y, x_i) = \nabla_y \times V(y, x_i) \quad (1.6)$$

Applications of Green’s function arise from constant coefficient Wave, Heat, Laplace, Helmholtz, and Ordinary Differential Equations resulting from Boundary Value Problem (BVP) [19].

1.1.2 Factorization and Complexity

1.1.2.1 Sample One-Dimensional Factorization

The degenerate kernel replaces the kernel $\Phi(y, x)$ of the linear one-dimensional Fredholm equation [20] by creating an expansion around a local or global point denoted by the monopole $x_*$ in Eq. 1.7.

$$\Phi(y_j, x_i) = \sum_{m=0}^{\infty} a_m(x_i - x_*) f_m(y_j - x_*) = \sum_{m=0}^{p-1} a_m(x_i - x_*) f_m(y_j - x_*) + \epsilon(p, y_j, x_i),$$

$$\forall x_i, y_j \in \mathbb{R}^d \quad (1.7)$$

Eq. 1.8 breaks down the dependency between sources and targets by plugging in Eq. 1.7 into the kernel Eq. 1.1 yielding the complexity reduction as highlighted by
Sec. 1.1.2.2

\[ v_j = \sum_{i=1}^{N} \phi(y_j, x_i) u_i \]

\[
\sum_{i=1}^{N} \left[ \sum_{m=0}^{p-1} a_m(x_i - x_*) f_m(y_j - x_*) + \epsilon(p, y_j, x_i) \right] u_i
\]

\[
\sum_{i=1}^{N} \sum_{i=1}^{N} \left[ a_m(x_i - x_*) u_i + \epsilon(p, y_j, x_i) u_i \right]
\]

(1.8)

where \( c_m = \sum_{i=1}^{N} a_m(x_i - x_*) u_i \)

1.1.2.2 Complexity Reduced from \( N^2 \) to \( N \)

Algorithms 1 and 2 abstractly explain how complexity is reduced from quadratic to linear or logarithmic linear by introducing a constant \( P \ll N \) expansion terms for each cell. Typically, “SOURCE” loop in Alg 2 is applied hierarchically on tree cells using the Particle-to-Multipole (P2M) followed by the Multipole-to-Multipole (M2M) kernels in a bottom-up manner. These kernels are depicted in the “Upward Pass” stage explained in the famous paper by Greengard et al. [3]. For well-separated cells, target values, which can be any physical quantity governed by kernels from Sec 1.1.1 are calculated with respect to “TARGET” loop in Alg 2. The number of expansion terms (a.k.a. poles) and the number of bodies per leaf cell are tightly coupled to the overall accuracy of FMM. For more rigid numerical implementation and error bound analysis of the method, consult [21].

1.2 Applications of the \( N \)-body Method

\( N \)-body methods can naturally be applied to problems where the physics itself is described by a collection of discrete points, even before numerical discretization.
Algorithm 1: $N$-body with Quadratic Complexity.

1. for $j = 1, \ldots, M$ do 
2. $v_j = 0;$ 
3. for $i = 1, \ldots, N$ do 
4. $v_j = v_j + \Phi(y_j, x_i)u_i$ 
5. end 
6. end 

Algorithm 2: FMM optimized kernel.

1. SOURCE; 
2. for $m = 0, \ldots, p-1$ do 
3. $c_m = 0;$ 
4. for $i = 1, \ldots, N$ do 
5. $c_m = c_m + a_m(x_i - x_\ast)u_i$ 
6. end 
7. end 
8. TARGET; 
9. for $j = 1, \ldots, M$ do 
10. $v_j = 0;$ 
11. for $m = 0, \ldots, p-1$ do 
12. $v_j = v_j + c_m f_m(y_j - x_\ast)$ 
13. end 
14. end
A many-body problem under gravitational or electrostatic forces is a typical example, where stars and atoms can be represented as point sources of mass and electrostatic charge, respectively. \(N\)-body methods can be extended from discrete fields to continuum fields through discretization. This makes it possible to use these methods for solving problems in structural mechanics \[22\], fluid mechanics \[23\], electromagnetics\[24\], acoustics \[25\], and even quantum mechanics \[26\]. However, just because it is applicable does not mean that it is the optimal method to solve that particular problem.

Fast \(N\)-body methods have been favored in applications where the geometry information changes dynamically. If the geometry is stationary, it is efficient to store this information in the form of a matrix, and to perform sparse/dense linear algebra operations on this same matrix across iterations. \(N\)-body methods can be thought of as matrix-free methods, where a matrix is formed on-the-fly before being multiplied to a vector of source points. It is obvious that such methods become advantageous only when the matrix/geometry changes frequently, since storing them would not save any computation in such cases. This is precisely the case for particle-based methods where each particle advances its location every time step. Adaptive mesh refinement may also result in a similar amount of geometry updates if the system is very dynamic.

1.3 Literature Review

1.3.1 FMM and \(N\)-body Algorithms

There are many factors that influence the comparative advantage of the \(N\)-body approach over other elliptic PDE solvers like FFT and multigrid. The asymptotic constant plays a critical role in determining the relative performance of these different \(O(N)/O(N \log N)\) algorithms. The FFT is known to have as few as \(2N \log N\) \[27\] operations and multigrid could have as few as \(5N\) operations. The FMM on the other hand typically has a much larger asymptotic constant. However, the use of transla-
tional and rotational symmetry in prescribing the position of points can reduce this constant significantly. Comparing FFT against FMM is difficult because FFT has a higher spatial resolution per unknown, so comparing for the same $N$ is not reasonable. However, for fields that have local features or discontinuities the homogenous spatial resolution of FFT is certainly a disadvantage [28]. Furthermore, when translational and rotational symmetry is utilized in the FMM by prescribing the position of points, it can use BLAS-3 operations much more efficiently than multigrid. Therefore, FMM becomes faster than multigrid to solve the same problem up to the same accuracy in such cases [28]. The FMM in the traditional sense requires a Green’s function solution, so the type of scientific applications that it can handle is limited to those that have a Green’s function. Generalization of the FMM to hierarchical low-rank approximations of matrices [29] enables the same framework to be applied to a much wider range of applications. These methods use low-rank approximation methods such as rank-revealing QR [30], truncated SVD [31], or adaptive cross approximation [32] instead of multipole expansions. This frees the FMM from its dependence on the existence of Green’s functions, so that it can be applied to problems like variable coefficient Poisson equations or covariance matrices, which are not directly solvable with FMM. When predicting the performance of FFT, FMM and multigrid on future architectures, a useful indicator is communication complexity since the bottleneck of any algorithm becomes the communication as it approaches its limit of parallel scalability. FFT has a communication complexity of $O(P^{1/D})$ for a D-dimensional decomposition. Multigrid has a communication complexity of $O(\log P)$. We have recently been able to prove that FMM also has a communication complexity of $O(\log P)$ [33]. Therefore, we see that both FMM and multigrid are communication optimal. This proof for $O(\log P)$ of FMM can be extended to its algebraic variants as well [33].
1.3.2 Runtime Systems and Skeletons

Since the first appearance of skeleton-based programming in [34], several skeleton libraries have been introduced. Some of these libraries support skeleton composition either at the specification level or at the implementation level. Examples are SkeTo [35], Müesli [36], JaSkel [37], SKiPPER [38], and Eden [39].

SkeTo [35] is a C++ skeleton framework implemented on top of MPI that provides parallel skeletons for parallel data structures. SkeTo allows successive invocation of elementary skeletons, but requires global synchronization between skeleton invocations due to the Bulk Synchronization Parallel model it implements. Our framework defines composition in terms of point-to-point dependencies, avoiding global synchronization costs.

Muesli [36] is a C++ skeleton library that is built for both shared and distributed memory, sitting on top of OpenMP and MPI. It provides a variety of task parallel and data parallel skeletons, but the composition of skeletons is restricted. The creation of a new composition requires reimplementation, as it cannot be defined directly from existing skeletons.

JaSkel [37] is a Java skeleton framework that provides farm, pipe and heartbeat skeletons. JaSkel provides separate implementations of each skeleton for shared and distributed memory systems, making it difficult to extend. Moreover, composition of skeletons in this framework requires reimplementation, limiting its generality.

SKiPPER [38] is a skeleton library developed in Caml. SKiPPER allows composition of its limited set of skeletons, e.g., map, reduce, etc. These skeletons are tailored towards vision applications and can work only for shared-memory systems. The compositions in this framework are tailored for specific purposes and cannot be generalized and extended.

Eden [39] is another skeleton framework which is intended to be an extension of Haskell. Similar to other functional languages, Eden allows non-strict functions,
demand driven evaluation and monads. Eden provides methods for composing and nesting existing skeletons, and allows new skeletons to be added. The main drawback of Eden is that it cannot compete with high performance computing skeleton frameworks that are developed in languages like C and C++. In our framework we present the same expressibility level as Eden, while preserving performance.

There have already been a few attempts to use data-driven execution models with parallel hierarchical $N$-body methods. Jetley et al. use the Charm++ execution model for their cosmological $N$-body code ChaNGa [40]. They compare several different cosmological datasets on several different architectures, and show significant improvement in the scalability over another cosmological $N$-body code PKDGRAV. They show that a naïve load-balancing scheme based on work-stealing increases the amount of communication three-fold. ChaNGa has also been extended to run on GPUs [41]. The tree construction and tree traversal are done on the CPU and only the force calculation is performed on the GPU. They report 3.82 Tflops (single precision) on 896 CPU cores + 256 S1070 GPUs, which is less than 2% of the theoretical peak. They are able to calculate approximately 10 million particles per second on 448 CPU cores + 128 GPUs. However, state-of-the-art parallel $N$-body codes such as pfalcON and ExaFMM can calculate 10 million particles per second on a single CPU socket [42].

When assessing the usefulness of new data-driven runtime systems, it is unconvincing and problematic to use a code with orders of magnitude slower serial performance. As mentioned earlier, data-driven execution models add value not by providing load-balancing or data-caching features to parallel $N$-body methods, but rather by adding flexibility to the granularity at which these mechanisms can be executed. However, slow serial performance of the code will skew the discussion on the optimal granularity. For example, techniques on the finer end of the spectrum in Figure 5.1 will seem acceptable if the serial performance was slow enough, while in reality the communi-
cation latency could actually be too large for codes like pfal\textsc{con} and ExaFMM. The same can be said to the case of Dekate et al. \cite{43}, where they use the Parallel\textsc{x} execution model for the Barnes-Hut treecode \cite{5} and report a performance of 100K particles per second on a single CPU socket. This is almost 100 times slower than the state-of-the-art $N$-body codes, which can do 10 million particles per second.

### 1.3.3 Communication Reduction

Due to their increased importance in large-scale simulations, there is now a considerable literature on implementing parallel hierarchical $N$-body solvers, e.g., FMM. Also, since they are among Berkeley’s seven dwarfs, the numerical methods that are believed to be the most impactful in science and engineering according to \cite{13}, it is important to address issues arising at exascale especially the increasing cost of data movement (through memory hierarchy or network) as opposed to floating point operations. Even though many of the current FMM implementations are scalable to the full machine they run on, a communication reducing approach that works on at least an order of magnitude more nodes tends to be rarely the emphasis of these implementations. This tendency is justified in accordance to the trend in enhancing a node with multi/many-core capabilities. However, even within a many-core node, more sophisticated methods should be used to place and exchange data to get the maximum performance reported by the vendor. This is already implied in equipping the second generation of Intel\textsuperscript{\textregistered} Xeon Phi\textsuperscript{TM} processors code-named Knights Landing (KNL) with memory ‘clustering modes’. An example work that achieves full machine scalability using GPUs is that of Bédorf et al. \cite{10}, where a parallel algorithm for sparse tree construction and traversal that works completely on the GPU is introduced. At the construction phase, they map the 3D coordinates to Hilbert’s linear ($N$-bit) addresses, then particles are sorted to achieve locality in memory. To avoid the typical sequential insertions to build Hilbert trees \cite{44}, one particle is assigned per
GPU thread. A level-wise mask is applied successively on each particle to discover its predecessors such that cells with less than $N_{leaf}$ are considered leaves. Grouping of particles is done using parallel compact algorithm. To exploit the massively parallel GPU threads, a breadth-first traversal is used to carry out the computation. They report a processing rate of 2.8 million particles per second. This work was extended to an MPI parallel version where 24.77 PFlop/s (mixed precision) on the full Titan system [45] was achieved.

Speck et al. [16] report scalability on up to 262,144 cores by introducing temporal parallelism (parallel-in-time algorithm) on top of MPI/Pthreads spatial decomposition to overcome the strong scaling limits when the number of particles per node becomes too small. The scalability is shown for up to 4M particles; then when they take advantage of shared and distributed memory parallelism, and exploit the overlap of data-exchange and computation, they calculate 2 billion particles on 262,144 cores of JUGENE, according to [47]. Lashuk et al. [48] propose an FMM implementation that scales on up to 196,608 cores by providing a novel domain-specific bulk synchronous all-reduce algorithm for remote tree communication. They report communication complexity of $O(\sqrt{P} \cdot (N/P)^{2/3})$, which comes from their hypercube alltoall communication scheme. Hoefler et al. [15] discuss the time and memory complexity of the common protocols used for the dynamic sparse data exchange problem and develop the non-blocking exchange protocol ($NBA$) with constant memory overhead. Their novel algorithm improves the runtime of sparse data-exchange up to 8,192 processors of Bluegene/P by a factor of 5.6. They prove and model a generic time complexity of $O(\log P)$ using the LogGP model.

Zandifar et al. [49] provide a parallel FMM implementation as a benchmark for their high-level skeletons (abstract parallel patterns) framework which executes on top of the STAPL runtime system that dynamically schedules task on highly heterogeneous architectures. They reuse several parallel patterns like the bucket-sort
and alltoall to perform geometric bisection and to aggregate the local essential tree (LET) respectively. They achieve comparable performance to the corresponding base MPI implementation by taking advantage of the underlying data-driven execution and asynchronous task scheduling guaranteed by the runtime system. Many features of Charm++ like task migration and Structured Control Flow are augmented in \[50\] to overlap computation with the communication of the local essential tree (LET).

1.4 Objectives and Contributions

The contributions of this thesis address many known limitations of the FMM through the following advances:

- Vectorized and shared-memory many-core optimizations of the direct $N$-body kernels which comprise the major compute-intensive part of FMM.
- Asynchronous adaptations of the global tree communication to the Charm++ runtime system.
- High-level expression of the FMM execution path using STAPL Skeleton Framework.
- A novel demonstration that shows a weakness in Hilbert’s space-filling interval partitioning for boundary element distributions.
- A communication scheme with adjustable granularity, which enables the overlap of local essential tree communication with computation that otherwise cannot be overlapped.
- Introduction of the adaptive $HSDX$ a neighborhood collective communication algorithm for exchanging the global tree in a few steps by direct near-field communication only.
Chapter 2

Direct N-body Kernel on Many-core Architecture

2.1 Direct N-body Code

We begin with an example of a direct N-body kernel, first in plain C language and
then using Single Instruction Multiple Data (SIMD) intrinsics. The plain N-body
kernel is shown in Listing 2.1. We define eight arrays that describe the properties
of the body. The arrays $x, y, z$ are the coordinates, $m$ is the mass/charge, $p$ is the
potential, and $ax, ay, az$ are the acceleration in each direction. The equation we
calculate is the smoothed Laplace potential

\[
\phi_i = \sum_{j=1}^{N} \frac{m_j}{r_{ij}} \quad (2.1)
\]

and acceleration

\[
a_i = \nabla \phi_i = -\sum_{j=1}^{N} \frac{m_j r_{ij}}{r_{ij}^3} \quad (2.2)
\]

where

\[
r_{ij} = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2 + (z_i - z_j)^2 + \epsilon^2} \quad (2.3)
\]

is the distance between the bodies at $i$ and $j$, and $\epsilon$ is the smoothing factor.
2.2 SIMD and Shared-Memory Approaches

The arrays \(x, y, z, m\) are the given prescribed values in the present example, a random number uniformly distributed between 0 and 1. These values are used to calculate the potential and acceleration on all \(N\) bodies, which is induced by all \(N\) bodies. This results in a double loop from 0 to \(N - 1\), as shown in Listing 2.1. The loop for \(i\) goes over the target bodies, while the loop for \(j\) goes over the source bodies.

```
#pragma simd
#pragma omp for
for (i=0; i<N; i++) {
    float pi = 0;
    float axi = 0;
    float ayi = 0;
    float azi = 0;
    float xi = x[i];
    float yi = y[i];
    float zi = z[i];
    for (j=0; j<N; j++) {
        float dx = x[j] - xi;
        float dy = y[j] - yi;
        float dz = z[j] - zi;
        float R2 = dx * dx + dy * dy + dz * dz + EPS2;
        float invR = 1.0f / sqrtf(R2);
        float invR3 = m[j] * invR * invR * invR;
        pi += m[j] * invR;
        axi += dx * invR3;
        ayi += dy * invR3;
        azi += dz * invR3;
    }
    p[i] = pi;
    ax[i] = axi;
    ay[i] = ayi;
    az[i] = azi;
}
```

Listing 2.1: Direct \(N\)-body kernel. Targets (outer loop) are vectorized.

We will use the code in Listing 2.1 as a base case to see how much performance we can achieve on the coprocessor without changing the code. The combination of the two pragmas for SIMD and OpenMP makes it possible to parallelize the outer loop both over threads and SIMD vectors. The performance results when using different compiler options are shown in the following section. This same code runs on both
the processor and the coprocessor.

The coprocessor has 512-bit wide SIMD intrinsics, which are similar to Streaming SIMD Extensions (SSE) and Advanced Vector Extensions (AVX). Since these intrinsic instructions directly map to assembly instructions, it leaves less ambiguity in what the compiler is doing to the code. To be more specific, it tells the compiler to explicitly perform load, store, fmadd, and rsqrt operations, and also the loop to be vectorized is also explicitly specified.

The direct \( N \)-body kernel has an outer loop (Sources (\( I \))) and an inner loop (Targets (\( J \))). With the use of SIMD intrinsics, it is possible to specify that the outer loop should be vectorized by putting 16 array elements into the SIMD registers and using a stride of 16 for the loop, as shown in Listing 2.2. The general structure of the direct \( N \)-body code has not changed, but all the operations are now written in \_mm512 intrinsics, and all intermediate values are declared as \_m512 registers. When possible the \textit{fmadd} instruction is explicitly specified.

An alternative form of vectorizing the inner loop is shown in Listing 2.3. The code is almost identical to the one in Listing 2.2 except the stride of 16 is now in the \( j \) loop, and a \textit{reduce-add} operation must be performed at the end instead of a simple \textit{store}.

The code for AVX intrinsics is very similar to the ones shown in Listings 2.2 and 2.3 except the \textit{fmadd} are changed to separate \textit{mul} and \textit{add} operations and the \textit{reduce:add} operation in Listing 2.3 becomes a combination of \textit{permute2f128}, \textit{add} and \textit{hadd} operations.

2.3 Task-based Traversal of ExaFMM

The traversal stage essentially calculates near-field or self interactions (P2P); whereas far-field potentials are aggregated to well-separated cells through M2L kernel calls.
#pragma omp for
for (i=0; i<N; i+=16) {
    __m512 pi = _mm512_setzero_ps();
    __m512 axi = _mm512_setzero_ps();
    __m512 ayi = _mm512_setzero_ps();
    __m512 azi = _mm512_setzero_ps();
    __m512 xi = _mm512_load_ps(x+i);
    __m512 yi = _mm512_load_ps(y+i);
    __m512 zi = _mm512_load_ps(z+i);
    for (j=0; j<N; j++) {
        __m512 xj = _mm512_set1_ps(x[j]);
        __m512 yj = _mm512_set1_ps(y[j]);
        __m512 zj = _mm512_set1_ps(z[j]);
        __m512 R2 = _mm512_set1_ps(EPS2);
        R2 = _mm512_fmadd_ps(xj, xj, R2);
        R2 = _mm512_fmadd_ps(yj, yj, R2);
        R2 = _mm512_fmadd_ps(zj, zj, R2);
        __m512 mj = _mm512_set1_ps(m[j]);
        __m512 invR = _mm512_rsqrt23_ps(R2);
        mj = _mm512_mul_ps(mj, invR);
        pi = _mm512_add_ps(pi, mj);
        invR = _mm512_mul_ps(invR, invR);
        invR = _mm512_mul_ps(invR, mj);
        axi = _mm512_fmadd_ps(xj, invR, axi);
        ayi = _mm512_fmadd_ps(yj, invR, ayi);
        azi = _mm512_fmadd_ps(zj, invR, azi);
    }
    _mm512_store_ps(p+i, pi);
    _mm512_store_ps(ax+i, axi);
    _mm512_store_ps(ay+i, ayi);
    _mm512_store_ps(az+i, azi);
}

Listing 2.2: Direct N-body kernel with intrinsics. Targets (outer loop) are vectorized.
#pragma omp for
for (i=0; i<N; i++) {
  __m512 pi = _mm512_setzero_ps();
  __m512 axi = _mm512_setzero_ps();
  __m512 ayi = _mm512_setzero_ps();
  __m512 azi = _mm512_setzero_ps();
  __m512 xi = _mm512_set1_ps(x[i]);
  __m512 yi = _mm512_set1_ps(y[i]);
  __m512 zi = _mm512_set1_ps(z[i]);
  for (j=0; j<N; j+=16) {
    __m512 xj = _mm512_load_ps(x+j);
    __m512 yj = _mm512_load_ps(y+j);
    __m512 zj = _mm512_load_ps(z+j);
    __m512 R2 = _mm512_set1_ps(EPS2);
    R2 = _mm512_fmadd_ps(xj, xj, R2);
    R2 = _mm512_fmadd_ps(yj, yj, R2);
    R2 = _mm512_fmadd_ps(zj, zj, R2);
    __m512 mj = _mm512_load_ps(m+j);
    __m512 invR = _mm512_rsqrt23_ps(R2);
    mj = _mm512_mul_ps(mj, invR);
    pi = _mm512_add_ps(pi, mj);
    invR = _mm512_mul_ps(invR, invR);
    invR = _mm512_mul_ps(invR, mj);
    axi = _mm512_fmadd_ps(xj, invR, axi);
    ayi = _mm512_fmadd_ps(yj, invR, ayi);
    azi = _mm512_fmadd_ps(zj, invR, azi);
  }
  p[i] = _mm512_reduce_add_ps(pi);
  ax[i] = _mm512_reduce_add_ps(axi);
  ay[i] = _mm512_reduce_add_ps(ayi);
  az[i] = _mm512_reduce_add_ps(azi);
}

Listing 2.3: Direct N-body kernel with intrinsics. Sources (inner loop) are vectorized.
The essential difference between pure tree codes and FMM is that the former usually constructs the tree using a linked-list data structure; the tree is traversed in a recursive top-down manner, and well-separated cells are identified by applying the MAC. In contrast, adaptive FMM does not traverse the tree, nor does it construct a linked-list between parent and child cells. It, however, constructs a Hilbert/Morton key by interleaving bits of x-y-z cell coordinates. Typically the parent’s neighbor’s child cell is considered well-separated. Nonetheless, there are several downsides of this technique, which are highlighted in [51].

Due to potential task-based parallelism, we configure ExaFMM to use Dual-Tree-Traversal (DTT), which traverses the source and target cells at the same time. Classical OpenMP threading is not applicable for the lack of an outer-loop over target cells. DTT takes a source and a target cell, and expands the larger until either MAC is satisfied or both are leaf cells. Alg. 3 and Alg. 4 demonstrate the general structure of DTT code. Nested task-parallelism can be effectively incorporated by passing an integer nspawn that indicates the size of cells that can spawn a task as shown in Line 8 of Alg. 4.

**Algorithm 3:** DualTreeTraversal($C_i$, $C_j$).

1. if $C_i > C_j$ then
2.     foreach $c_i$ in $C_j$.Children do
3.         Interact ($c_i$, $C_j$)
4.     end
5. else
6.     foreach $c_j$ in $C_i$.Children do
7.         Interact ($C_i$, $c_j$)
8.     end
9. end
Algorithm 4: Interact($C_i$, $C_j$).

1. if $C_i$ and $C_j$ are leafs then
2. P2P($C_i$, $C_j$)
3. else
4. if $C_i$ and $C_j$ satisfy MAC then
5. M2L($C_i$, $C_j$)
6. else
7. if SizeOf ($C_i$, $C_j$) > nspawn then
8. Spawn (DualTreeTraversal ($C_i$, $C_j$))
9. else
10. DualTreeTraversal ($C_i$, $C_j$)
11. end
12. end
13. end

2.4 NUMA-aware Communication Reducing Algorithms

The local essential tree (LET) is the union of trees representing the entire domain as perceived by the local process. LET communication is known to be the major factor that hinders FMM’s perfect scaling. [52], [15] and [53] describe specific communication protocols named HSDX and NBX respectively. They provide optimizations that are specific to distributed sparse data exchange, which generally suits the communication structure of FMM’s global tree. We explore the effect of different communication strategies within the KNL chip. Note that we implemented all of these strategies on an ExaFMM branch [54]. Table 2.1 briefly highlights various techniques that we benchmark. Note that “hierarchical” protocol means that the data is aggregated along a structured hierarchy such as graphs and trees, whereas “sparse-aware” protocol avoids direct communication with partitions without or with very little data to exchange (almost negligible). In the context of NUMA systems, hierarchical protocols tend to maximize locality of the data within each local caches, and in the case of data exchange, each process requires the data only from its neighboring MPI ranks. Hence, the communication is mostly localized inside the NUMA socket. However, if
the required data happens to be in different NUMA socket, then MPI would communicate the cache line from the socket’s memory, which is very negligible in proportion to locality maximizing communication protocols.

Table 2.1: MPI-Based Communication Paradigms.

<table>
<thead>
<tr>
<th>Name</th>
<th>MPI Calls</th>
<th>Complexity</th>
<th>Hierarchical</th>
<th>Sparse-Aware</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alltoallv</td>
<td>MPI_Alltoallv</td>
<td>MPI specific</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>Hierarchical Alltoallv</td>
<td>MPI_Comm_Split</td>
<td>MPI specific</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>Point-to-Point</td>
<td>MPI_Isend, MPI_Irecv, MPI_Wait</td>
<td>O(P)</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>Hypercube [55]</td>
<td>MPI_Comm_Split</td>
<td>O(log P)</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>NBX</td>
<td>MPI_Isend, MPI_Irecv, MPI_Ibarrier</td>
<td>O(log P)</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>HSDX</td>
<td>MPI_Distgraph_create, MPI_Neighbor_alltoallv</td>
<td>Ω(log P), O(log² P)</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>One-Sided [56]</td>
<td>MPI_Win_create, MPI_Get</td>
<td>O(P)</td>
<td>no</td>
<td>no</td>
</tr>
</tbody>
</table>

2.5 Results and Discussions

For KNL experiments, we used two Linux servers that run CentOS Linux 7.3.1611 Operating System. Both servers are powered by Intel Xeon Phi CPU 7210, which is equipped with 64 hardware cores that execute at 1.30GHz clock frequency, and both have access to 116 GB of DRAM. The typical specifications of the KNL chip that we used here, can be found in [57]. For Broadwell experiments, we used a Linux server that runs Ubuntu 14.04.5 LTS Operating System. The server is powered by dual sockets of Intel Xeon CPU E5-2680 v4, each of which is equipped with 14 hardware cores that executes at 2.40GHz clock frequency. Each socket has access to a single address space of size 64 GB of DRAM. So, the server has a NUMA node of in total
28 hardware cores and 128 GB of DRAM. For KNC experiments, we used a Linux server that runs Scientific Linux release 6.4 (Carbon) Operating System. The server is powered by two Intel Xeon Phi 7120P coprocessors, each of which is equipped with 61 hardware cores that runs at 1.238GHz clock frequency, and each has access to 16 GB of DRAM. The typical specifications of the KNC chip that we used here, can be found in [58]. The two KNC chips are hosted by a dual socket Intel Sandy Bridge E5-2670 CPU. Each socket consists of 8 hardware cores (in total 16 cores). The CPU clock speed is 2.6GHz. Both sockets share a 64 GB DRAM (32 GB per socket). All of the experiments here were run with Intel Parallel Studio XE 2017 as the main software stack that comes with Intel ICC, MPI, TBB, OpenMP, and Cilk. The data sets are based on a single precision Laplace kernel with Cartesian coordinates, and the FMM order of expansion is set to 4. For the KNL results, all of the experiments are ran with -xMIC-AVX512, and for KNC, we use -mmic compiler option. For Broadwell, on the other hand, we use -xHost compiler flag. All of the experiments are compiled with -O3 compiler optimization flag. All of the experiments here are summarized using the arithmetic mean of the CPU wall clock time across 10 independent runs, which forms the sample space, and an error bar is drawn to show the +/- standard deviation of the mean for each experimental sample.

2.5.1 SIMD Optimizations of the $N$-body Kernel

The $N$-body kernel is constructed with two nested for loops. The outer loop is the target loop and the inner loop is the source loop. We explore loop tiling on each loop, with 16 stride size for KNL. So, in a single CPU cycle, each OpenMP thread fetches 16 bytes of data into the vector unit. In the case of two threads per core, each thread processes 16 bytes simultaneously utilizing the two vector units per core of KNL. However, if one thread per core is running, the next 16 bytes are pipelined in the second vector unit, and the thread scheduler alternates between them in a se-
Figure 2.1: Direct N-body kernel running on two Intel architectures, KNL (quadrant cluster and flat memory modes) and Broadwell [Problem Size: 1 million particles].

rialized manner, which keeps the core busy as much as possible. Furthermore, when
the full number of threads per core are running, the threads are pipelined to process
the data of both vector units. So, with four threads per core, KNL utilizes both
vector units and the pipelining potentials available in the out-of-core execution of the
core’s instruction pipeline. We observe that tiling targets as opposed to sources wins
constantly in KNL; in each outer loop iteration, cache lines pertaining to elements in
the target vector are loaded only once to AVX512 register using _mm512_load_ps
intrinsic. This in turn does not require calling _mm512_reduce_add_ps after iterat-
ing over sources, which must be done otherwise because vectorizing effects of source
fields must eventually be reduced to one value at target. Fig. 2.1 presents the perfor-
mance of the N-body kernel running on KNL comparing five different optimization
techniques: 1) Target-Intrin: N-body outer-loop tiling. 2) Source-Intrin: N-body
inner-loop tiling. 3) Target-Auto: outer-loop wrapping with #pragma simd. 4) Souce-Vec: inner-loop wrapping with #pragma simd. 5) No-vec: scalar code.

We note that the handwritten vectorization does not improve much over auto-vectorization
in KNL. It even appears that the ICC compiler was able to detect the event of recip-
rocal square root known as _mm512_rsqrt28_ps. Overall, vectorization benefits the
kernel and shows significant improvements compared to the non-vectorized version of the code. This is not entirely the case in Broadwell; the variations are not proportional to their rivals in KNL, which suggests that outer-loop manual tiling cannot be avoided in Broadwell. We infer that KNL’s AVX512 has a more sophisticated mechanism of matching correct vectorization than AVX2.

Figure 2.2: FLOP/s performance across the 2 Intel many-core generations, KNL (quadrant cluster and flat memory modes) and KNC.

Fig. 2.2 draws a comparison between floating point applicabilities of the 2 many-core generations by Intel, namely KNC and KNL, in terms of the aforementioned vectorization techniques. Error bars suggest reasonable stability in clocking frequency
in both generations. Auto-vectorization in KNL reaches maximum flop/s rate in an at least four times smaller problem, which strongly suggest that it utilizes local caches in a much more efficient manner. The drop in performance for slow versions happens exactly at the time when the performance of manual target vectorization saturates. This also suggests that the drop happens when prefetching and cache reuse could no more hide the overhead caused by source vectorization [58], which is $2^{15}$ in KNL (Fig 2.2(a)) and $2^{14}$ in KNC (Fig 2.2(b)).

![Figure 2.3: Dual tree traversal using different task threading models. (Problem Size: 100 million particles).](image)

![Figure 2.4: Strong scaling of FMM tree traversal on KNL vs. Haswell using different number of MPI processes (Problem Size: 100 million particles).](image)
2.5.2 Dual Tree Traversal with Task-Based Threading

Fig. 2.3 shows traversal scalability using several threading libraries. The purpose of this test is to assess the DTT (Alg. 3) performance using task-based/lightweight threading libraries on many-core architectures. Error bars are hardly observable, because frequency scaling has been disabled on KNL to stabilize performance. As expected, there is a general loss of scalability aspect when hyperthreading is enabled. Intel TBB perfectly scales up to 64 threads (1 thread/core). Scaling to the full chip, i.e., 256 threads, its relative speedup is 14.94, with an efficiency of 0.469, compared to 0.4249 in Intel Cilk and 0.1912 in OpenMP tasks. It is observed that there is a weak separation between user-level and OS-level threads in OpenMP tasks. This is due to the very marginal performance gain from enabling hyperthreading in OpenMP tasks [58] (1.1x speedup between 64 and 128 threads). Intel TBB, therefore, has the lowest task creation overhead, due to its efficient performance for heavily recursive tasks. However, Cilk does not seem to pose significant degradation in performance although it has minimal development time since it is integrated as a C++ language extension in modern Intel compilers.

2.5.3 Communication Reduction On KNL

As compute nodes are packaged with more low frequency cores, it is essential that MPI communication scales within main memory or across the NUMA sockets. Therefore, we apply various MPI communication reducing algorithms from Table 2.1 to FMM’s tree communication. Results that are shown in Fig 2.5 are executed with 64 MPI ranks, and a single thread per each rank, so that the effect of locality-maximizing behavior can be clearly observed.

HSDX (Distgraph) performs better than the others, and this due to restricting exchanges to neighbors only, which makes it potentially NUMA-aware and yields acceptable on-chip performance. In other words, in HSDX algorithm, we tend to
maintain a load balance between the KNL tiles, so that each tile acts like a sender and receiver of the cache lines. Thus, this model of communication prevents any long distance cache line transfer inside the chip, and maintains load balance of the cache line distributions across the tiles. To further demonstrate this, we investigate this phenomena when we change the cluster mode of KNL. As shown in Fig 2.6, the HSDX the cluster modes of KNL do not have significant performance impact on the algorithm, and the performance differences between different modes are very negligible. Note that SNC-2 and SNC-4 modes are still experimental modes [59].

One-sided communication has a huge overhead for shared window creation, which requires soft locking prior to data access. This latency cannot be hidden when fetching sparse data either from the memory or from the other L2 caches. Even though, KNL has a great support for AVX512 prefetching instructions, locking the window before accessing the data imposes an implicit synchronization barrier on every data read. This creates a significant overhead on a cache-coherent systems.
2.6 Conclusions

With many-core processors with high degree of fine-grained thread parallelism within a single shared-memory compute node, practitioners are now compelled to investigate strong thread scaling. We present optimizations and thorough analysis of an FMM code on a modern high performance Intel many-core architecture, KNL. We described how to optimize a direct $N$-body kernel on the coprocessor. We were able to achieve about 1.5 TFlops single precision performance in KNC as opposed to 2.5 TFlops in KNL by simply using the compiler option `icc -mmic openmp fimf-domain-exclusion=15` and adding a `#pragma simd` to the original C code for CPUs. The strong scalability of multi-core execution was close to ideal, while the scalability of the intra-core threading was less efficient. The choice of "KMP_AFFINITY" did not have any effect when all 256 KNL threads were utilized. By using the _mm512 intrinsics, we were able to increase the performance to about 2.5 TFlops even at smaller problem sizes. We found a strong dependence of the performance on the problem size and the relation was not monotonic. Up to a certain problem size vectorizing the inner loop gave better performance, but after $N > 16,384$
vectorizing the outer loop gave better performance. The vectorization of the outer loop was made possible by use of a `#pragma simd` directive or `_mm512` intrinsics.

We extract the potential SIMD and thread-level parallelisms of three different computationally intensive kernels, namely P2P, tree traversal, and LET communication kernels. We demonstrate several shared-memory optimizations on these kernels, including different task-based threading paradigms, vectorization, loop tiling, and NUMA-aware communication-reducing. Our shared-memory optimizations present significant improvements that are reflected on the N-body kernel compare to the out-of-the-box compilation of the non-optimized version. These feature in excess of 2.8x speedup on two Intel multi and many architectures, KNL and Broadwell. Furthermore, the task-based parallelism of the tree traversal kernel shows almost linear scaling, within a massively parallel single compute node, up to 64 thread contexts of KNL. With hyperthreading the performance gain becomes slightly monotonic. The NUMA-aware communication algorithm based on optimizing MPI_alltoall communication protocol to maintain load balancing and shorter cache line transfers within a chip are explored. It is found that HSDX perform considerably faster than any other communication models; even across different cluster modes of KNL, it still maintains marginally the same performance.
Chapter 3

Partitioning and Communication of the Local Essential Tree (LET)

3.1 Dataflow of FMM

The bottom picture in Figure 3.1 shows a geometrical partitioning of a two-dimensional domain and the corresponding data-dependency between the location of the cells. The P2M kernel takes the information of the particles (coordinates and charges) and calculates multipole expansion coefficients from this information (shown in red). For details of the mathematical formulation of FMM see [60, 1]. The M2M kernel takes the information of the multipoles from its child cells and aggregates this information into a new multipole expansion at the center of a larger cell (shown in orange). The M2L kernel takes all the information of the multipole expansions in the tree and translates them to local expansions (shown in yellow). There is a special rule for M2L kernels that it can only interact with cells that are sufficiently far compared to its cell size. The bigger the cell the further the other cell must be. The gray zones in Figure 3.1 show the region where the M2L kernel is valid. The union of the gray zones on the three levels of M2L, M2L, and P2P shown in the center column add up to the entire domain. In a much deeper tree in 3D this would be like peeling layers of M2L interaction lists until it reaches a ball of P2P neighbor lists, but they add up to the whole domain. The Local-to-Local (L2L) kernel takes the information given by the M2L kernel and cascades it down the tree (shown in light-green), until it reaches the bottom at which point the Local-to-Particle (L2P) kernel is called to translate
that information to each particle (shown in dark-green).

Figure 3.1: Illustration of the flow of FMM calculation, and the interaction between source and target particles.

The data-flow of FMM is analogous to a mail delivery system, where the information is aggregated from local post office to a larger hub, the delivery between remote locations is done with cargo aircraft, and then distributed back to the local post
offices before delivery to the individual. This is a very efficient delivery system for computational problems that require information to travel from everywhere to every other place. Mathematically speaking, elliptical equations belong to this class of problems where some form of information must travel from one end of the domain to the other for the system to achieve a state of equilibrium. This could either be done by successive iteration with local halo communication over stencils, or it can be done more directly and asynchronously by packing/compressing everything and sending it over the network at once.

An important fact that can be overlooked in Figure 3.1 is that the M2L and P2P phases are much more expensive than the other phases. These two phases make up more than 90% of the total runtime, so naively integrating the data-flow with the remaining 10% will never result in a performance increase of more than 10%. Furthermore, every M2L cell depends on hundreds of M2M cells. It is not a clear data-path where each node in the Direct Acyclic Graph (DAG) has one arrow pointing to the next node. It is a DAG with hundreds of arrows pointing to a single M2L node. This is another reason why data-flow programming at this level of granularity is not favorable.

FMM has two major communication phases: the partitioning of particles (load-balancing), and the LET communication (prefetching). We describe in the next two subsections the details of each of these two phases and how Charm++ is used to add asynchronicity and granular flexibility.

### 3.2 Partitioning Schemes for FMM

There are two traditional objectives associated with good partitioning of the N-body problem: evenly splitting data among partitions to achieve work balance, and providing efficient access to non-local data. There is no optimal approach that can simultaneously handle these two objectives, because of strict considerations on locality
of data for high arithmetic intensity, granularity, and the size of communication which can vary based on space-time proximity of partitions.

(a) HOT (Morton).
(b) HOT (Hilbert).
(c) ORB.
(d) Present method.

Figure 3.2: Schematic of different partitioning schemes. (a) shows the hashed octree with Morton keys. (b) shows the hashed octree with Hilbert keys. (c) shows the orthogonal recursive bisection with an underlying global tree. (d) is the present method using an orthogonal recursive bisection with independent local trees and tight bounding boxes.

3.2.1 Preliminaries

Partitioning schemes for fast N-body methods can be categorized into ORB [61] or HOT [12].

3.2.1.1 Orthogonal Recursive Bisection (ORB)

The ORB [61] forms a balanced binary tree by finding a geometric bisector that splits the number of particles equally at every bisection of the tree. The direction of the geometric bisector alternates orthogonally (x, y, z, x, ...) to form a cascade of rectangular subdomains that contain equal number of particles similar to Fig 3.2(c).
For nonuniform distributions the aspect ratio of the subdomain could become large, which leads to suboptimal interaction list size and communication load. This problem can be solved by choosing the direction of the geometric bisector to always split in the longest dimension. The original method is limited to cases where the number of processes is a power of two, but the method can be extended to non-powers-of-two by using multi sections instead of bisections [62].

3.2.1.2 Hashed Oct-Tree (HOT)

In HOT, initially proposed by [12], domain is partitioned by splitting Morton/Hilbert ordered space filling curves into equal segments as shown in figures 3.2(a) and 3.2(b). Morton/Hilbert ordering maps the geometrical location of each particle to a single key. The value of the key depends on the depth of the tree at which the space filling curve is drawn. Three bits of the key are used to indicate which octant the particle belongs to at every level of the octree. Therefore, a 32-bit unsigned integer can represent a tree with 10 levels, and a 64-bit unsigned integer can represent a tree with 21 levels. Directly mapping this key to the memory address is inefficient for nonuniform distributions since most of the keys will not be used. Therefore, a hashing function is used to map the Morton/Hilbert key to the memory address of particles/cells.
3.2.2 Adopted Partitioning Strategies

3.2.2.1 Parallel Sampling-Based Techniques for Finding Splitters/Bisectors

Parallel sampling-based techniques have proven to be useful for both finding the bisectors in ORB [62] and finding the splitting keys in HOT [63]. Both ORB and HOT are constructing parallel tree structures, but in different ways. There is an analogy between parallel tree construction and parallel sorting. The idea behind ORB is analogous to merge sort, where a divide and conquer approach is taken. HOT is analogous to radix sort, where each bit of the key is examined at each step. Therefore, sampling-based techniques that are known to be effective for parallel sorting are also effective for parallel tree partitioning. The partitioning can be separated into two
steps. The first step is to find the bisectors/key-splitters by using a sampling-based parallel sorting algorithm. An example of such sampling-based partitioning is shown in Fig. 3.3. Sorting is only performed among the buckets (not within them) and this is done only locally. The only global information that is communicated is the histogram counts, which is only a few integers and can be done efficiently with an MPI_allreduce operation. The bins can be iteratively refined to narrow the search for the splitter of the HOT key or ORB bisector. This will determine the destination process for each particle. The second step is to perform an all-to-all communication of the particles. Since the ORB bisector is one floating point number and the HOT key is one integer, it is much less data than sending around particle data at each step of the parallel sort.

3.2.2.2 Weakness in Space-Filling Partitioning for Boundary Distributions

It is well-known that the main advantage of Hilbert curve as opposed to Morton is its locality preserving properties in 2D. It is not clear, however, to what extent we can generalize this property in higher dimensions [64]. As a counterexample to the locality property, we observe that it is not entirely preserved in case of 3D boundary element distributions, which increases the distributed interaction list size. The reason for that comes from the intuitive notion of space-filling curves, that is, when the space is not filled, e.g., in boundary spherical distribution, interpolation of spatial points to Hilbert curve does not necessarily map to keys that are continuous in space. This is attributable to the fact that keys are not interpolated in their natural order, since point are spread out on surface units. Fig. 3.4 shows particles laid out in their respective Hilbert order. Due to the geometry of the space-filling curve, movement across dimensions happens orthogonally, hence, if hollow space is encountered in the orthogonal direction, it will introduce discontinuity in the partition as in Fig. 3.4(a).
Clearly, this does not apply to uniform dense distributions, which comprise many classical applications of FMM making HOT partitioning an optimal choice in such cases.

Figure 3.4: A Hilbert partition from a boundary spherical distribution viewed from different perspectives. A space discontinuity exists even though partitions are in correct Hilbert order due to the existence of hollow space in orthogonal dimensions.

3.2.2.3 Hybrid Partitioning

In our implementation, we choose modified version of ORB over HOT for a few other reasons. One of the main reasons is that we were able to improve a major defect of ORB – partition-cell alignment issue. Since geometrically closer points interact more densely with each other, it is crucial to keep the particles in the same cell on the same process in order to minimize communication. However, if a global Morton/Hilbert key is used to construct the local trees, the ORB may place a bisector in the middle of a cell as shown in Figure 3.2(c). This results in an increase in the interaction list
size. We avoid this problem by using local Morton/Hilbert keys that use the bounds of the local partition. This may at first seem to increase the interaction list near the partition boundaries since two misaligned tree structures are formed. However, when one considers the fact that the present method squeezes the bounding box of each cell to tightly fit the particles as shown in Figure 3.2(d), it can be seen that the cells are not aligned at all in the first place. Furthermore, our flexible definition of the multipole acceptance criteria optimizes the interaction list length for a given accuracy regardless of the misalignment.

3.2.2.4 Weighting by Workload

One area where the existing load-balancing and prefetching scheme can be improved is the granularity at which they are performed. Figure 3.5 shows the spectrum of granularity for the partitioning phase. Currently, the partitioning phase is constrained to the granularity of a single time step. One could coarsen the granularity by delaying the update of the partition for a few time steps, thereby adding more room for asynchronous execution. It is also possible that a repartitioning could take place within a time step in case of a node failure. Adding such flexibility to the partitioning granularity is a partial requirement for making the algorithm fault tolerant.

![Figure 3.5: Shows the different granularity at which the partitioning phase can take place.](image)

There is one more important ingredient for an efficient partitioning scheme – weighting by the workload. Particles have varying interaction list sizes, so equally splitting the bisection/key results in suboptimal load-balance. Weighting the particles
with the workload from the previous time step is a simple and effective load-balancing strategy. This technique was mentioned in the original HOT paper [12] and little has been done to improve it to this day. It would be naïve to propose a work-stealing mechanism for fast \( N \)-body methods without understanding the significance of this practical solution that has stood the test of time. Data-driven execution models should be able to augment this tailored feature rather than to reinvent it. Although this weighting scheme was originally proposed for HOT, it can obviously be used to determine weights for particles during the bisection in ORB.

One limitation of the weighting scheme is that it only balances the workload and not the communication. There have been efforts to use graph partitioning tools with the workload as node-weights and communication as edge-weights, in order to create partitions that have an optimal balance of both the workload and communication [65]. This method has only been compared with Morton key splitting without weights, so the advantage over Morton key splitting with weights is unclear. In the present work we attempt to balance the workload and communication simultaneously by calculating the weight for the \( i^{th} \) particle \( w_i \) according to

\[
w_i = l_i + \alpha \times r_i \quad (3.1)
\]

where \( l_i \) is the local interaction size, \( r_i \) is the remote interaction list size, and \( \alpha \) is a constant that is optimized over the time steps to minimize the total runtime. \( l_i + r_i \) is the total interaction list size and represents the workload, while \( r_i \) reflects the amount of communication. By adjusting the coefficient \( \alpha \), one can amplify/damp the importance of communication balance. Making this an optimization problem to minimize the total runtime is what we prefer over minimizing the load-imbalance since the latter is not our final objective. Moreover, the variables \( l_i, r_i \), and the total runtime are already measured in the present code so the information is available at
3.3 Communication of the Local Essential Tree

Once particles are partitioned, those in the local domain are used to construct a local tree. We use a completely local construction of the octree using the local bounding box, instead of using a global Morton/Hilbert key that is derived from the global bounding box. This allows us to reuse all parts of the serial code and only add a few routines for the partitioning, grafting of trees, and communication. Therefore, any modification in the serial code is immediately reflected in the parallel code.

After the local tree structure is constructed, a post-order traversal is performed on the tree structure and Particle-to-Multipole (P2M) and Multipole-to-Multipole (M2M) kernels are executed bottom up. The P2M kernel is executed only at the leaf cells. It loops over all particles in the leaf cell to form the multipole expansion at the center of the leaf cell. The M2M kernel is executed only for the non-leaf cells. It loops over all child cells and translates the multipole expansions from its children’s centers to its center.

Once the multipole expansions for all local cells have been determined, the multipole expansions are sent to the necessary processes in a sender-initiated fashion [66]. This reduces the latency by communicating only once, rather than sending a request to remote processes and then receiving the data. Such sender-initiated communication schemes were common in cosmological N-body codes since they tend to use only monopoles, and in this case the integer to store the requests is as large as the data itself if they were to use a request-based scheme. This data is used to construct the LET, that is, the union of all trees representing the entire domain as seen by the local process [67]. It gets coarser depending on the distance of the remote cell. In the present method, it is formed by simply grafting the root nodes of the remote trees. In conventional parallel FMM codes, a global octree is formed and partitioned using...
Figure 3.6: Schematic of the local essential tree (LET) for conventional HOT/ORB partitioning and present method.

either HOT or ORB. Therefore, the tree structure was severed in many places, which caused the merging of the LET to become quite complicated. Typically, code for merging the LET would take a large portion of a parallel FMM code, and this made it difficult to implement new features such as periodic boundary conditions, mutual interaction, more efficient translation stencils, and dual tree traversals. exaFMM is able to incorporate all these extended features and still maintain a fast pace of development because of this simplification in how the global tree structure is geometrically separated from the local tree structure.

While the remote information for the LET is being transferred, the local tree can be traversed. Conventional fast $N$-body methods overlap the entire LET communication with the entire local tree traversal. The LET communication becomes a bulk-synchronous MPI_alltoallv type communication, where processes corresponding to geometrically far partitions send logarithmically less information, thus resulting in $O(\log P)$ communication complexity where $P$ is the number of processes. Nonetheless, in traditional fast $N$-body codes this part is performed in a bulk-synchronous manner.
By supplementing the tree with local Hilbert keys, we no longer require prior knowledge of the geometry of the remote partitions, and we are able to perform tree traversal (i.e., Remote M2L and P2P) asynchronously. Figure 3.6 can be thought of as three remote trees located in the same order in space. We note that traversal of rank$_0$ can start independently from myrank. Also, myrank has more cells to traverse due to its geometric location.
Chapter 4

FMM Communication Latency Hiding Using Runtime Systems

4.1 exaFMM with Charm++

In order to understand the amount of potential asynchrony in FMM, one must understand the data-dependencies between the different phases of the FMM. FMM has six different mathematical operations that it performs, each with different dependencies and workloads. Figure 3.1 shows the data dependency of FMM in two separate schematics. The picture on the top shows a birds-eye view of the interaction between the red source particles and the blue target particles. Each connection or arrow shown in this figure represents a data-dependency. The interactions are grouped hierarchically so that far particles interact more sparsely. This is quite different from FFT, where even remote data points require equal amount of communication between them.

Charm++ [68] has been successfully used in a well-known scalable molecular dynamics library (NAMD) [69]. It provides an opportunity to augment this bulk-synchronous approach to load-balancing by offering control over the granularity. We are interested in the data-flow programming model of Charm++, which allows us to asynchronously execute the FMM kernels while the communication for partitioning is happening. To our knowledge, there have not been any attempts to overlap computation with the communication in the partitioning phase. Charm++ also provides task migration capabilities, but we decided not to use this feature for the current study. This is because we believe that it is much more efficient to “strategically update” the
partitions so that they are well balanced, than to try to “steal work” after they are partitioned poorly.

exaFMM-Charm++ replaces the global synchronization points, which consist of aggregation of LET data through all to all communication, with asynchronous sender-initiated entry functions that represent a coherent work entity called Chare. The local partitions are mapped to Chares that are accessible through entry functions. Such entry constructs propagate cell/body data across nodes to traverse the corresponding LET, rather than traverse all LETs in one shot. The advantages are reflected at both CPU and Network utilization levels. Global communication is typically synchronous in nature and are dependent on two factors: network bandwidth and initialization cost. It is clear that for large messages the first factor dominates whereas latency hiding could overcome this problem in rare cases; however, in general, such events act like global barriers. By supporting the traversal as an entry card to the LocalTree structure, the post-order traversal is triggered once the LET is received. The function in Figure 4.1 shows a direct mapping of the synchronized global communication to Charm++, that undergoes the Structured Control Flow abstraction. The “when” construct controls the sequence at which messages are received, and the code inside it will be executed at the receiver side. It is clear from the abstraction that the next

```c
entry void allToAllCells() {
    atomic {
        transportCellsToAll();
    }
    for (count=0; count<numChares; count++) {
        when transportCells(int cellCount,
            Cell b[cellCount], int sender) atomic {
            processCells(cellCount, b, sender);
        }
    }
    atomic {
        finishAllToAllCells();
    }
};
```

Figure 4.1: A generic bulk synchronous Charm++-mapped version of global communication.
workload is triggered once all of messages are received. The function in Figure 4.1 is utilized by exaFMM-Charm++ to replace the global blocking receive of cells with a remote asynchronous call that will process the message and proceed based on the rank of the sender.

Once the LET is formed the $M2L$ and $P2P$ kernels can be calculated using this information from the remote processes. The calculation of these two kernels takes a large portion of the execution time of FMM. The $P2P$ kernel only requires information from its neighbors, while the $M2L$ kernel requires information from an intermediate range. Besides these read-dependencies these two kernels do not have any level-wise dependency within the tree structure and can be processed in parallel on a per cell basis. In exaFMM the $M2L$ and $P2P$ kernels are processes without forming an explicit interaction list by using the dual tree traversal [70]. After the dual tree traversal is finished, a post-order traversal is performed and $L2L$ and $L2P$ kernels are executed to cascade the information down the tree to the particles.

4.2 Algorithmic Skeletons with STAPL

The increase in availability of high-performance computing systems has made the need for expressive and efficient methods of parallel program specification more evident. Many parallel libraries and frameworks currently provide easier methods for expressing parallel applications [71, 72, 73, 74]. However, most of these studies are tied to specific programming models, and programs expressed using such methods are not reusable in other libraries and frameworks.

Algorithmic skeletons [34], on the other hand, tackle this issue by separating program specification from parallel implementation. These higher-level representations, which are usually defined using polymorphic higher-order functions, represent common interaction patterns prevalent in parallel programs [75]. map, zip, and reduce are examples of these patterns. The functional representation of skeletons provides
<table>
<thead>
<tr>
<th>Operators</th>
<th>elem, repeat, compose</th>
</tr>
</thead>
<tbody>
<tr>
<td>Skeletons</td>
<td>allgather, allreduce, alltoall, bitreversal, broadcast, butterfly, copy, fft, gather, inner-product, map, pointer-jumping, reduce, reverse-butterfly, reverse-tree&lt;(k)&gt;, scan, scatter, transpose-2d, transpose-3d, wavefront-2d, wavefront-3d, tree&lt;(k)&gt;, zip&lt;(k)&gt;, zip-reduce&lt;(k)&gt;</td>
</tr>
</tbody>
</table>

Table 4.1: Provides skeletons and compositional operators.

opportunities for formal analysis and transformations to be applied on parallel programs, regardless of their parallel implementations \[75\]. Providing efficient parallel implementations for these fundamental skeletons in a skeleton framework can simplify parallel programming. Algorithm developers focus on the computation of an application and leave the parallelism details to be handled by the skeleton framework developers. Moreover, the functional representation of skeletons allows large and complex applications to be composed from these building blocks.

Several skeleton frameworks provide efficient implementations of fundamental skeletons for both shared-memory systems and small clusters \[76\]. However, in these frameworks, skeleton composition is either not considered at all, requiring reimplementation of composed skeletons, or is a **Bulk Synchronous Parallel (BSP)** execution of each skeleton in a composition followed by a global synchronization at each step \[35, 77\]. Even if these synchronizations do not play a large role in the performance of applications on lower core counts, they can degrade performance at a larger scale.

The **STAPL** Skeleton Framework resolves these outstanding issues. In this framework, we use parametric data flow graphs as our internal representation of skeletons.
This representation allows programs to run efficiently on parallel systems, regardless of their size. We provide efficient implementations for various fundamental skeletons, as listed in Table 4.1, from which larger programs can be composed. We translate composition of skeletons by point-to-point dependencies between their corresponding data flow graphs. Using point-to-point dependencies, instead of global synchronizations, allows programs written in terms of skeletons to scale better. To show expressivity we present skeleton-based implementation of the NAS EP, IS, and FT benchmarks. An \( N \)-body application using the FMM hierarchical algorithm shows the reusability of skeletons in a real-world application. Our framework can be easily extended either through composition of skeletons or by adding data flow graph representations for new skeletons. Furthermore, our framework can be ported to other libraries with data flow engines.

Please note that many details of The stapl Skeleton Framework implementation will be eliminated for conciseness. Further literature review and specifications are available in [49].

4.3 Skeleton Composition

Since skeletons are defined in terms of higher-order functions, they are by definition composable. Many applications can be expressed as a composition of fundamental skeletons. For example, map-reduce, a frequently used skeleton, can easily be expressed as a composition of the map and the reduce skeletons. Although specifying skeletons composition is relatively simple at the specification level, translating them to efficient parallel implementations is not as straightforward. This issue has limited the usability of many skeleton frameworks [76] to very small applications.

In the stapl Skeleton Framework we use data flow graphs as our internal representation of skeletons. This representation allows composition to be defined in terms of point-to-point dependencies between the data flow graph representation of
skeletons. Therefore, we eliminate the need for global synchronizations in algorithms that are defined as compositions of other skeletons, and allow communication and computation to be overlapped.

In this section, we describe the process of generating such data flow graphs from their fine-grain dependence relations using our composition operators: elem, repeat, and compose. The interfaces for these Operators are given Table 4.2.

We use inner-product as our example. Formally, this skeleton is defined as a composition of a zip and a reduce operators (reduce(+) \circ zip(\times)). First, we explain how these two fundamental skeletons are created using the elem and repeat operators from their parametric dependence relations which we refer to as parametric dependencies for brevity. We then show how these two skeletons are composed to represent the data flow graph representation of the inner-product skeleton (Fig. 4.2).

Table 4.2: The grammar for the compositional operators.
4.3.1 compose - Arbitrary Composition

Functional composition is very straightforward for defining simple compositions. However, this form of expressing composition becomes less expressive in arbitrary and complex compositions. In functional programming languages this problem is resolved using *let* expressions (a.k.a. *letrec*). A *let* expression simplifies compositions by potentially avoiding repetition and, at the same time, improving readability [78]. In a *let* expression, subexpressions are given names and can be reused; we use a mini-language to express the same type of compositions. The grammar for this mini-language is listed in Table 4.3.

4.3.1.1 Ports

We use our mini-language descriptions to define the mappings of the input and output of the skeletons in this type of composition. In Listing 4.1 we show how the mini-language can be used to express the functional composition used for the inner-product skeleton.

The first line after the name in Listing 4.1 indicates that this skeleton receives two inputs. In the second line, these inputs are passed to the zip skeleton. The result of this computation is then passed to the reduce skeleton. Finally, the output of the reduce skeleton is used as the output of the inner-product skeleton. This simple example defines the simple functional composition using our mini-languages and is given here for easy of understandability. In the STAPL Skeleton Framework, we provide generic port-mappings for simple functional compositions of arbitrary size and do not require users to provide port-mappings for such cases.

We use the mini-language for complex compositions to express the port-mappings. We provide a Python-based tool which translates these descriptions to strongly-typed port-mappings in C++. The generated files can be used in compositions by simply
1 simple-port-mapping:
2 input = [in1 in2]
3 v1 = zip<2>(* [in1 in2]
4 v2 = reduce(+) [v1]
5 output = [v2]
6 // C++ code for this composition
7 auto inner_product = compose<portmaps::simple-port-mapping>(
8 zip<2>(*), reduce(+))

Listing 4.1: Examples of functional composition using the compose operator.

| Composition | input = [(Var)+] |
| Statement | output = [(Var)+] |
| Var = Var[(Var)+] |
| Var ::= | [a - Z0 - 9]+ |

Table 4.3: The grammar for the mini-language used by the compose operator.

adding the port-mapping name to the compose operator (portmaps::simple-port-mapping in Listing 4.1). Our tool also generates a GraphViz [79] dot file for the given composition description. As an example, the graphical representation of the bucket-sort skeleton is shown in Fig. 4.3.

![Figure 4.3: The graphical representation of the bucket-sort skeleton’s port-mappings.](image)

We use this method of composition to show the compositions used in our case studies in the results section. The C++ code for these compositions is similar to the one line listed in Listing 4.1 Therefore, we omit it in the rest of the examples for ease of readability.
4.3.2 Case Studies

In this section, we present the skeleton-based implementation of the 3D FFT computation used in the NAS FT (Fourier Transform). In this benchmark we use the transpose-3D skeleton, which is also composed from other skeletons. Finally, we express an N-body application using FMM, which uses the bucket-sort skeleton, as an example of reusability, in addition to several other skeletons. These examples demonstrate the reusability and expressivity of skeletons.

4.3.2.1 Skeleton-based FT

In the NAS FT benchmark, a partial differential equation is solved by using a 3D FFT (Fast Fourier Transform) on a 3D matrix [80]. A 3D FFT algorithm can be implemented using three steps of applying 1D FFTs across each dimension. In the literature, two methods are generally used for this computation [81]: distributed and transpose-based. In the first method, an efficient distributed 1D FFT is used. Although this method is very expressive, the long-range accesses in the algorithm prevent this method from scaling beyond very small clusters [81]. In the second method, which relies on efficient implementation of the sequential 1D FFT, 1D FFTs are applied across the dimensions for which data is locally available. The matrix is transposed whenever necessary to make the values across a dimension available locally. We use the transpose-based method in our framework due to the lack of scaling in the distributed method (Listing 4.2).

There are also two methods for matrix partitioning which are used for 3D FFT in the literature [81, 82]: the slab decomposition, in which each processor gets a 2D slab of the input, and the pencil decomposition, in which each processor gets a pencil (column) of the input. In this chapter, we show the expressive skeleton-based representation for the slab decomposition, although the pencil decomposition can be expressed in a similar way. As can be seen in Listing 4.2, in the forward FFT, we
compute 1D FFTs across the $x$ and $y$ axes first, and then transpose the matrix from $(x,y,z)$ to $(z,x,y)$. We use $\text{transpose-3D}$ for this step. This skeleton is defined in term of the $\text{zip}$ and $\text{alltoall}$ skeletons, which is omitted for the sake of brevity. After the matrix is transposed, 1D FFTs are computed across the $z$ dimension. The reverse 3D FFT is defined in a similar way.

### 4.3.2.2 Skeleton-based FMM

Using the set of skeletons provided in our framework, we have implemented FMM in terms of skeletons (Listing 4.3). As we will see in the evaluation section, our implementation not only increases the expressivity of this algorithm, but also improves its overall performance in comparison to the reference implementations. In addition, by providing algorithm selection at user's level (switching between various implementations of $\text{alltoall}$, $\text{allreduce}$, and $\text{allgather}$ skeletons in our framework), we allow algorithm developers to adapt their algorithm based on their heuristics. This is an advantage over the MPI implementation of such algorithms.

Moreover, our implementation is reusable for other PDE problems that can be solved using Green’s function of the form listed in Eq. 4.1 ($\delta$ represents the Dirac delta)
function). The Green’s function, in these problems, provides a visual interpretation of an impulse response, which is caused by a force or a charge at the center of mass.

\[ \mathcal{L}G(x, s) = \delta(x - s) \] (4.1)

To achieve acceptable performance in an N-body problem, an efficient partitioning scheme should be used. This partitioning scheme has to consider three factors: number of particles per partition, workload, and input distribution. For a uniform distribution, the global geometric ORB is sufficient for achieving a suitable load-balancing. For a nonuniform distributions the Plummer distribution can be used. In both cases, it is important to keep geometrically close points in the same partition, to reduce the intra-node interaction list sizes. Additionally, the previous workload and the amount of communication have to be balanced simultaneously to achieve load-balance across time-steps.

To fine-tune the partitioning algorithm based on the uniformity of input, we use the bucket-sort skeleton. Using this skeleton, we sort the splitters for our partitioning selection criteria and determine the physical bounds of each partition by applying an allreduce skeleton on the results. We then use the build-tree to build an octree at each partition, based on the geometric positions of particles. In the upward_pass, we recursively calls the [P2M] kernel for terminal cells or [M2M] (if a cell is within the multipole acceptance criteria). When octrees are created, we recalculate the bounds and populate them with the help of an allreduce skeleton followed by an allgather skeleton. Afterwards, to reduce communication in the next steps, we create the Local Essential Trees (LETs) in each partition and make this information available to other partitions using the alltoall skeleton. LETs contain a subset of the local tree cells that are required by other partition for their compu-
72

Listing 4.3: FMM Skeleton Composition.

tations. Since the number of LETs required by a partition is inversely proportional to distance, the complexity of this step is \( O(\log P) \). Once LETs are received, we combine them to form the global octree. In the \texttt{upward\_pass} kernel, we invoke the \texttt{M2L}, \texttt{L2P} and \texttt{P2P} kernels on the trees, to calculate potential and acceleration of target particles. Finally, we use the \texttt{repeat} operator to calculate the subsequent snapshots. Since particles at boundaries might move across nearby partitions, we do not require a global repartitioning.

4.4 Performance Analysis

4.4.1 Communication Latency Hiding Using Charm++

<table>
<thead>
<tr>
<th>Case</th>
<th>( N_{body} )</th>
<th>( P )</th>
<th>( \theta )</th>
<th>( N_{crit} )</th>
<th>( N_{spawn} )</th>
<th>Distribution</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>( 10^8 )</td>
<td>10</td>
<td>0.4</td>
<td>256</td>
<td>1,000</td>
<td>Cube</td>
</tr>
<tr>
<td>2</td>
<td>( 10^8 )</td>
<td>10</td>
<td>0.4</td>
<td>512</td>
<td>1,000</td>
<td>Sphere</td>
</tr>
<tr>
<td>3</td>
<td>( 10^8 )</td>
<td>10</td>
<td>0.4</td>
<td>64</td>
<td>1,000</td>
<td>Plummer</td>
</tr>
</tbody>
</table>

Table 4.4: FMM parameters.
The following tests were performed on the TACC Stampede system without using the coprocessors. Stampede has 6400 nodes, each with two Xeon E5-2680 processors with eight physical cores and 32GB of memory. We used the Intel compiler (module intel/13.0.2.146, impi/4.1.0.030) and used the Intel Thread Building Blocks library for the threading model. The exaFMM code that was used for the current study is publicly available on bitbucket. \(^1\)

![Cube, Sphere, Plummer](image)

Figure 4.4: Different distributions of particles.

### 4.4.1.1 Strong Scalability

We compare the scalability of exaFMM with and without the use of Charm++. exaFMM has many tunable parameters as shown in Table 4.4. \(N_{body}\) is the number of bodies, \(P\) is the order of multipole/local expansion, and \(\theta\) is the multipole acceptance.

\(^1\)https://github.com/exafmm/exafmm-alpha
criteria. The FMM has a theoretical error bound of $O(\theta P)$, while the computational complexity varies between $O(N\theta^{-2}P^3)$ and $O(N\theta^{-2}P^6)$ depending on the type of basis [51]. Unlike, previous treecodes that can only control $\theta$ or FMM codes that can only control $P$, exaFMM can achieve the optimal speed by controlling both $P$ and $\theta$ simultaneously.

In Table 4.4, $N_{\text{crit}}$ represents the maximum number of particles per leaf cell, while $N_{\text{spawn}}$ is the minimum number of particles per spawned thread. Using a large $N_{\text{crit}}$ will create a shallower tree and decrease the number of M2L interactions, but will increase the number of particles per cell and therefore increase the number of P2P interactions. Using an optimal $N_{\text{crit}}$ value is essential to balance the workload between the M2L and P2P kernel, which are the two most expensive parts of the FMM. Increasing $N_{\text{spawn}}$ will allow fewer threads to be created and will decrease the overhead of the task spawning. Decreasing $N_{\text{spawn}}$ will cause more threads to be created and will make it easier to load-balance, but may increase the runtime due to the overhead caused by spawning many tasks. These values were carefully chosen to maximize the performance on Stampede.

The final entry in Table 4.4 is the distribution of particles. We have selected three different types of distributions, which are representative of the actual distributions in scientific application codes. An illustration of the three distributions is shown in Figure 4.4. The “Cube” distribution can be found in molecular dynamics simulations where water molecules are evenly distributed throughout a cubic domain. The “Sphere” distribution has points only on the surface of the sphere. This is representative of boundary integral problems, where a surface mesh is used to discretize the problem. The “Plummer” distribution is typical for cosmological $N$-body simulations, where the mass is distributed unevenly with very high concentration in certain areas.

We perform a strong scalability test of the FMM by keeping the number of particles
to the value shown in Table 4.4 and increasing the number of cores. All the values in Table 4.4 are kept constant throughout the strong scalability tests. We first run up to 16 cores per node and then increase the node count once the number of cores per node is saturated. The total number of cores used in the largest run was 4,096.

The results of the strong scalability test using exaFMM with and without Charm++ are shown in Figure 4.5. The divergence from ideal scaling is mainly caused by the increase in the interaction list size when splitting the constant-sized tree into smaller and smaller segments. By looking back at Figure 3.2, one can see that all partitioning schemes will suffer from this problem because it is difficult to maintain a small surface to volume ratio when partitioning a constant domain into thousands of subdomains. For any partitioning scheme, the shapes of the partitions tend to be neater at a macroscopic level, but the unevenness in the particle distribution at the microscopic scale tends to create oddly-shaped partitions as you go finer.
We take a closer look at the strong scalability runs by plotting the breakdown of the runtime in Figure 4.6. The breakdown in Figure 4.6 corresponds to the plot for exaFMM in Figure 4.5. The main difference between the two plots is that Figure 4.5 is showing the speedup, whereas Figure 4.6 is showing the runtime multiplied by the number of cores. This is done so that the bar plot for larger core counts is clearly visible. Therefore, in Figure 4.6 a constant bar height will mean perfect strong scalability.

It can be seen that the “Comm partition” phase is consuming a large time on 4,096 cores. This is the communication of the partitioning stage, which is very large for the initial step. Note that the two LET communication phases “Comm LET bodies” and “Comm LET cells” are completely overlapped with the “Traverse” and cannot be seen in Figure 4.6. The original exaFMM code overlaps the LET communication with local tree traversal so adding Charm++ does not improve the performance any further for this part. However, the communication for the initial partitioning phase is not overlapped with any computation in exaFMM (or any other fast N-body code as far as the authors are aware), so the asynchronous execution model of Charm++ provides some benefit for this part.

### 4.4.1.2 Load Balancing

The increase in runtime of the “Traverse” phase shown in Figure 4.6 is mostly attributed to the increase in the interaction list length as mentioned earlier, but it is also partially caused by load-imbalance. We see this in Figure 4.7, where the runtime across all cores is shown with the same legend as Figure 4.6 but this time without multiplying the runtime by the number of cores. As can be seen from Figure 4.4, the Plummer distribution is highly nonuniform and is difficult to partition to thousand of subdomains. Furthermore, the main difficulty of partitioning N-body codes is that the work-load is not directly proportional to the partition size. For mesh-based
methods, partitioning into equal size subdomains would result in somewhat equal workload. However, since each particle has a different interaction list size, partitioning the domain so that the number of particles are equal will result in suboptimal load-balance.

It is difficult to assess the quality of our load-balancing scheme by just looking at a single case, especially if perfect balance is not a reasonable goal to aim for. Therefore, we will compare the results for the different distributions shown in Table 4.4. If the most benign distribution is showing the same amount of load-imbalance as the most difficult case. Then we should be able to conclude that there is little room for improvement. Figure 4.8 shows the distribution of the runtime on each core for six different cases. The top three cases are for the “Cube”, “Sphere”, and “Plummer” distribution with the standard weighting scheme based on the interaction list length. The bottom three cases are for the same distributions but with our new weighting scheme that tries to optimize for both the work and communication load by using Eq. (6.15). The number of cores is set to 1,025 to create an environment where the

---

Figure 4.6: Breakdown of strong scaling for $N = 10^8$ particles.
number of processes is not a power of two. We used 16 threads per node except for
the last node which used only 1 thread. Therefore, we had to use one extra node,
which makes the partitioning of MPI ranks not so straightforward.

We see from Figure 4.8 that all distributions have somewhat similar load imbalance
despite these difficult conditions. Therefore, we conclude that our partitioning scheme
can handle difficult distributions and difficult number of processes to the same degree
that it can handle the easy ones.

4.4.2 Communication Latency Hiding Using the STAPL Skeleton Framework

We have evaluated the Skeleton Framework on three massively parallel systems: a
24,576 node BG/Q system (VULCAN), each node containing a 16-core IBM PowerPC
A2 for a total of 393,216 cores; a 5,272 node Cray XC30 supercomputer (Piz Daint),
each node containing a hyperthreaded 8-core Intel SandyBridge with 32GB per node,
for a total of 42,176 cores (84,352 hyperthreads); and a 1,266 node Cray XC40 super-
computer (a Piz Daint extension), each node containing two 12-core Intel Haswell CPUs, for a total of 30,144 cores.

### 4.4.2.1 NAS Fourier Transform

As shown in Fig. 4.9, our implementation of the **Fourier Transform (FT)** benchmark shows comparable performance to the hand-optimized reference implementation (Fig. 4.9). The FT benchmark stresses the system with interprocessor communication in its 3D matrix transpose, requiring efficient alltoall communication. Our overhead is due to the copy-semantics of the **stapl** runtime system. The **stapl** runtime system requires one copy between the user-level to the MPI level on both sender and receiver side. These extra copies result in only 30-40% overhead in our implementation, as some of the overhead is compensated for by not requiring global synchronizations.
4.4.2.2 Skeleton-based FMM

As can be seen in Fig. 4.10 and 4.11, our skeleton-based implementation of the FMM algorithm can scale better than the reference implementation. In both versions, we are exploiting distributed memory parallelism; no shared-memory or GPU optimizations are involved. This is attributable to the fact that, for this benchmark, we are making a direct comparison between a hand-optimized MPI implementation, given by ExaFMM reference code, and a Skeleton-based distributed implementation that internally uses the same serial FMM kernels given by the reference implementation.

In Fig. 4.10, we show the weak-scaling results for 50K particles per process (100M particles on 2K processes). As can be seen in this figure, the execution time for the reference implementation diverges rapidly from the skeleton-based version, after 32 processes and runs out of memory after 128 processes. This is due to the overhead caused by the global communication of LETs in the reference implementation, using MPI_Alltoall. Our implementation continues to scale up to 2K processes. By replac-
ing global synchronization with point-to-point dependency and reusing fine-tuned skeletons in our algorithm, we are able to scale by 2 more orders of magnitude than the related work and solve a problem of $10^8$ particles.

In Fig. 4.11 we show the strong scaling results for a problem of 25M particles. As you can see in this figure, the scalability of our implementation persists until 256 cores; at this point the theoretical amount of serial work assuming no communication is approximately 0.16s. This is considerably smaller than the actual amount of communication. The reference implementation crashes due of the same memory allocation issue, mentioned earlier, after 8 processes.

![Weak Scaling on CrayXC40 (Piz Daint Extension)](image)

Figure 4.10: Weak scaling of FMM using MPI vs Skeleton-based Implementations.

4.5 Conclusions

Distributed memory parallelization models for FMM have traditionally been bulk-synchronous, but dynamic load-balancing and data prefetching mechanisms have existed for over two decades. For load-balancing, the hashed-octree and orthogonal
recursive bisection are both effective techniques for maximizing data locality while balancing the workload among the partitions by using the workload from the previous step as weights when partitioning. For data-prefetching, the local essential tree is formed by communicating all necessary parts of the remote tree upfront.

Unlike previous work on asynchronous fast $N$-body methods such as ChaNGa and PEPC, the present work performs a direct comparison against the traditional bulk-synchronous approach and the asynchronous approach using Charm++. Furthermore, the serial performance of our FMM code is over an order of magnitude better than these previous codes, so it is much more challenging to hide the overhead of Charm++.

We also propose a novel partitioning scheme, which allows us to geometrically separate the local tree from the global tree. This was only possible because our FMM uses the dual tree traversal, which does not require a global key nor cubic cells. By taking advantage of this feature of the dual tree traversal, we were able to simplify the grafting of the local essential tree greatly. This simplification of our code made it
possible to readily integrate with frameworks such as Charm++ with relative ease.

In order to demonstrate the effectiveness of the current combination of state-of-the-art load-balancing, data-prefetching, and data-flow execution models, we performed a strong scalability test that spans over three orders of magnitude without offsetting the problem size. As expected, the communication for the initial partitioning phase became a bottleneck at 4096 cores, but we were able to improve this by using asynchronous execution model of Charm++. This allows us to achieve over a 1000 times speedup for an highly nonuniform Plummer distribution with \( N = 10^8 \) particles.

We confirmed that our weighting scheme for the partitioning works evenly well for various particle distributions. Random distribution in a cube, points on a spherical shell, and the highly nonuniform Plummer distribution all had a similar load-imbalance for \( N = 10^8 \) particles on 1025 (not 1024) cores. This also demonstrates that our partitioning scheme works equally well for non-powers of two. The exaFMM code that was used for the current study is publicly available on bitbucket.\(^2\)

We also presented a novel method for representing skeletons and their composition in the STAPL Skeleton Framework. We then composed the basic skeletons to build more FMM and FFT as benchmarks. With the abstraction that ports provided, we were able to define data dependencies between the skeletons as point-to-point dependencies of their corresponding data flow graphs, allowing programs to scale efficiently beyond 100,000 cores.

\(^2\)https://github.com/exafmm/exafmm-alpha
Chapter 5

Communication Reduction Using the Hierarchical Sparse Data Exchange

5.1 Communication Reduction for the Adaptive Tree

In the following sections, we present different novel techniques that can be used to do the \( \text{HSDX} \) of the adaptive FMM tree, which are generally applicable to a variety of algorithms constituting definition 5.1. The optimization of global tree communication is essential to achieve strong scaling especially at a large scale. Such class of communication becomes very challenging due to the fact that exaFMM has a highly optimized serial code that utilizes many-core parallelism, making the complete overlap with computation infeasible. The natural solution to this problem is to strong scale communication, but to our knowledge, it is not straightforward to achieve that for practical reasons such as network congestion, growing interaction lists, and the different implementations of some MPI collectives that do not scale by definition e.g., MPI\texttt{Alltoallv}. Therefore, it is important to look at these caveats while implementing a domain-specific communication scheme of the global FMM tree.

Definition 5.1.1. Let \( \mathcal{T} \) be a global adaptive tree with \( \mathcal{L} \) levels numbered from \( l_0 \) to \( l_k \) (coarse to fine) and partitioned to \( \mathcal{P} \) processes. \( s \) is the essential subtree size such that \( 0 < s < S \). \( P_i, P_j \subset l_k \), if the finest level \( P_i, P_j \) share is \( k \). We have a hierarchically sparse data exchange \( \text{HSDX} \) if for \( P_i, P_j \subset l_1 \) and \( P_i, P_v \subset l_2 \), \( s_1 < s_2 \) and \( s_1! = 0 \).
5.1.1 Overlapping Computation Depending on Communication Granularity

FMM has two major communication phases – the partitioning of particles, and the communication of the local essential tree (LET). The former performs dynamic load-balancing and the latter can be thought of as a prefetching or data caching mechanism. Data-flow execution models add value not by providing these features, but by adding flexibility to the granularity at which these phases can be executed asynchronously. Figure 5.1(a) shows the different granularity at which the partitioning phase can take place, while Figure 5.1(a) shows the different granularity at which the LET communication can take place. The bulk-synchronous model can be viewed as an extreme case of communication aggregation. The importance of asynchronous communication arises from the fact that the latter is a limiting factor to performance at exascale especially when done collectively. This appears to be the case for hierarchical algorithms such as FMM and Multigrid method (MG). Hence, communication needs to be balanced and efficiently overlapped with local work. In FMM, it is known that a substantial amount of time is spent in doing local Multipole-to-Local (M2L) and Particle-to-Particle (P2P) computations, but the question is how often we need to communicate to reduce blocking for data given the problem size, distribution and scale. To answer this question, we have parametrized our FMM to accept different granularities of communication represented by the size of the LET’s subset. The subsets may contain non-leaf cells requiring $O(p)$ steps where $p$ is the order of multipole expansion (higher $p$ increases arithmetic intensity for low-level kernels) or leaf cells requiring $O(N/P)^2$ steps for $P =$ number of processes. The typical case would be to call a blocking MPI_Recv on the expected tag because there is no useful work to do in the current context; however, since MPI does not provide guarantees on the order of messages when used in mixed mode, our code consumes the available subtree and marks it as traversed. This mechanism will maximize concurrency and minimize the
Figure 5.1: The different granularity at which the FMM communication can take place. The bulk-synchronous model can be viewed as an extreme case of communication aggregation.

message queuing time. The calling task will keep traversing until requested cell is received or traversed by another task.

Figure 5.1(a) shows the spectrum of granularity for the LET communication (prefetching) phase. Conventional parallel $N$-body methods use a bulk-synchronous \texttt{MPI\_alltoallv} to communicate the whole LET at once, and overlap this communication with the local tree traversal to hide latency. One could over-decompose the LET down to a per cell request, and then aggregate the communication to the optimal granularity. The bulk-synchronous communication model can be thought of as an extreme case of aggregation, while something like an RDMA per task per cell would be at the other end of the granularity spectrum. There is a caveat: We still require further tuning to reduce global communication by indirectly relaying multipoles through neighbor processes as we will show in Sec. 5.1.2 using Algorithm 5.

5.1.2 Hierarchical Sparse Data Exchange Protocol (HSDX).

Lashuk et al. \cite{84} define a set of parameters that denote the interaction lists, i.e., $U$, $V$, $W$ and $X$-lists of the FMM tree. The same analogy can be used for describing the relationship between adjacent processes such that exchanging the entire LET can happen in a few steps. The $U$- and $V$- lists constitute the adjacent nodes/processes through which global cells that contribute to the local tree are relayed. For the majority of the spatial $N$-body partitioning methods, we can use the subdomain’s bounding box to depict partitions that share a face, an edge or a vertex in $O(1)$ steps using Lemma 5.1.1 This enables us to create a breadth-first data exchange graph
that starts from the local tree and covers all the cells from the essential tree. Each node in the graph contains the corresponding partition id and the adjacent partition id, which is needed since communication strictly happens between adjacent nodes.

Figure 5.3 shows the exchanges needed to receive the entire LET by target process (3,3), with overlapping direct clusters enclosed in dashed squares. The corresponding data exchange graph of node (3,3) contains a node with id (1,5) and an adjacent id of (2,4), meaning that cell data of (1,5) can be acquired through (2,4) in the second stage of exchanges. To inherently achieve algorithmic balance, we hardwire edges in such a way that messages are evenly distributed over direct neighbors. If we start
Figure 5.3: The underlying data exchange graph of central process (3,3) within a uniform 2D grid of processes.

with direct neighbor (2,4), a naive approach would exhaust all its direct neighbors, namely \{(1,3),(1,4),(1,5),(2,5),(3,5)\}, thus overloads its buffers and causes imbalance. The next neighbor (3,4) will only have (4,5) data to relay. Therefore, we design our communication graph such that for internal processes in a uniform domain, the average number of messages received from direct neighbors in each step is \(\lceil \frac{5D - 3D}{3D - 1} \rceil\).

Using notations from Table 5.1 we can generalize this formula to nonuniform domains if we turn it into

\[
NB = \left\lceil \frac{\tau(P,1) - \zeta(\Omega(P))}{\zeta(\Omega(P)) - 1} \right\rceil
\]  

(5.1)

We finally reach a stage where each process has access to the near and far-field interactions, thus accomplishing global communication using multiple calls to MPI_Neighbor_alltoallv. Algorithm 5 and Table 5.1 summarize our method.
Table 5.1: HSDX Algorithm communication symbols.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Indication</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P$ and $P'$</td>
<td>local and global partitions</td>
</tr>
<tr>
<td>$\Omega(P)$</td>
<td>subdomain boundary</td>
</tr>
<tr>
<td>$\zeta(\Omega(P))$</td>
<td>direct neighbors of $P$</td>
</tr>
<tr>
<td>$T$</td>
<td>level-by-level communication adjacency graph</td>
</tr>
</tbody>
</table>

5.1.2.1 Time Complexity of the adaptive HSDX

A good lower bound complexity for HSDX is [Non-blocking Exchange (NBX)] i.e. $\Omega(\log P)$ from [15], when non-neighbor data exchange is extremely sparse or non-existent. The hierarchical sparsity in Definition 5.1 increases as we move away from target processes. The data exchange graph can be mapped to a tree since there is exactly one path from $P_i$ to $P_j$, with an order bounded by Eq 5.1. An upper bound is analogous to a fully dense communication, such that $O(\log P)$ exchanges happen $O(\log P)$ times, which is equivalent to $O(\log^2 P)$. Table 5.2 shows FMM communication complexity for uniform domains.

**Lemma 5.1.1.** A partition $P'$ is added to the adjacency list of $P$ iff for any dimension $D$ $\text{maxBound}(P'_x) - \text{maxBound}(P_x) > \epsilon$ and $\text{minBound}(P_x) - \text{minBound}(P'_x) > \epsilon$

5.1.3 Pairwise Exchange for Reducing Contention

It is observed at large scale that direct communication between sources and targets results in network contention which can be amortized by relaying multipoles through neighbor processes while utilizing the well-known pattern of $N$-body interactions. Therefore, to mimic $O(\log P)$ complexity for boundary distributions, we implement a modified version of the well-known hypercube (butterfly) global communication scheme which starts out by the fine neighbor interactions depicted by $(P \oplus 2^i)$ and gets coarser as we move towards the log $P$ step. This is clearly visualized in Fig 5.2.
Algorithm 5: HSDX - Hierarchical Sparse Data Exchange.

input: A list \( l_{in} \) of cells and destinations
output: A list \( l_{out} \) of cells and sources

1. foreach \( P' \) in \( \Omega(P, \beta) \) do
2. \quad add\((P', \zeta(\Omega(P)))\)
3. end
4. \( \mathcal{T} \leftarrow \text{BuildCommTree}(\zeta(\Omega(P))) \);
5. create distributed MPI graph topology;
6. foreach \( l \) in \( \mathcal{T}.\text{Levels} \) do
7. \quad foreach \( P' \) in \( \zeta(\Omega(P)) \) do
8. \quad \quad reduce tree based on the bounding box and forward to \( P' \);
9. \quad end
10. exchange meta data;
11. \quad call MPI\_Neighbor\_alltoallv;
12. end

using Extrae, a tool that uses different interposition mechanisms to inject probes into the target application so as to gather information regarding the application performance. During this work, the tool is used to better understand the performance of the application pertaining to the used communication techniques. In Fig 5.2, the horizontal axis represents the visualized timeline and the vertical axis represents the MPI processes. The yellow colors denote communication links, whereas the blue colors represent computation and red color symbolizes MPI\_Wait calls. One of the main advantages of carrying out communication in \( \log P \) steps, as in is Fig 5.2(a) is that subtrees received at intermediate stages can be asynchronously traversed, which otherwise cannot be done if communication is done with blocking collectives.

### Table 5.2: Communication complexity of FMM.

<table>
<thead>
<tr>
<th>Reference</th>
<th>Communication complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Teng (1998) [85]</td>
<td>( O\left(P(N/P)^{2/3}(\log N + \mu)^{1/3}\right) )</td>
</tr>
<tr>
<td>Lashuk et al. (2009) [84]</td>
<td>( O\left(\sqrt{P}(N/P)^{2/3}\right) )</td>
</tr>
<tr>
<td>Lashuk et al. (2012) [48]</td>
<td>( O\left(\log P + (N/P)\right) )</td>
</tr>
<tr>
<td>Yokota et al. (2014) [33]</td>
<td>( O\left(\log P + (N/P)^{2/3}\right) )</td>
</tr>
</tbody>
</table>
Table 5.3: $\mathcal{HSDX}$ strong scalability analysis with FMM.

<table>
<thead>
<tr>
<th>$P$</th>
<th>4,096</th>
<th>8,192</th>
<th>16,384</th>
<th>32,768</th>
<th>65,536</th>
</tr>
</thead>
<tbody>
<tr>
<td>HSDX</td>
<td>32.72</td>
<td>17.02</td>
<td>9.27</td>
<td>5.008</td>
<td>3.05</td>
</tr>
<tr>
<td>Rel. Speedup</td>
<td>1</td>
<td>1.92</td>
<td>3.53</td>
<td>6.53</td>
<td>10.70</td>
</tr>
<tr>
<td>Efficiency</td>
<td>-</td>
<td>0.96</td>
<td>0.88</td>
<td>0.81</td>
<td>0.66</td>
</tr>
<tr>
<td>Enhancement</td>
<td>3.87%</td>
<td>11.41%</td>
<td>10.55%</td>
<td>16.27%</td>
<td>23.44%</td>
</tr>
</tbody>
</table>

5.2 Performance Analysis

5.2.1 Experimental Setup

We perform experiments on Shaheen XC40, the rank 32 supercomputer according to the November 2018 Top500 list, located at King Abdullah University of Science and Technology. It has 196,608 physical cores and HPL performance of 5.537 PFlop/s. Each node is equipped with dual socket Intel Xeon E5-2698v3 16C 2.3GHz and Cray Aries interconnect with dragonfly topology.

Throughout the following experiments, the underlying FMM code is compiled with the Laplace kernel, Cartesian coordinates, $P = 4$ (order of expansion) and spherical boundary distribution unless otherwise stated. Problems have been partitioned using the hybrid partitioning from section 3.2.2.3. To demonstrate the effectiveness of the presented methodologies, we start by showing how optimal grain size for a specific problem is chosen, then assessing the scalability with the tuned granularity of communication. Then, results from using $\mathcal{HSDX}$ vs. existing communication reducing approaches are presented. Good scalability shows that an inordinate cost is not paid for intra-node communication, as opposed to the conventional bulk-synchronous approach, for which performance depends on the underlying network topology, the implementation of collectives like alltoall or allgather, the available memory size and bandwidth, and the frequency at which synchronization is triggered.
5.2.2 Communication Time for Different Granularities

In order to show the direct effect of asynchronous traversal on performance, we gradually vary the grain-size and measure the communication time, which is the most dominant factor at a large scale. Optimal granularity is a tuning parameter that varies with problem size, distribution and other factors as depicted by the average communication time in Fig. 5.4 where subtree size is gradually increased. The theoretical maximum size is the entire LET. We stop at a certain threshold (32 in this case) because when it is increased further, a huge jump in time occurs. This is attributable to the change in communication protocol as per the Cray® MPICH specification from Eager Message to Rendezvous Message Protocol. When the message size exceeds a specific threshold (8 KB in this case), MPICH2 GNI NetMod alters the pathways towards a more relaxed algorithm for point-to-point inter-node messaging. A similar approach is developed in other MPI implementations like Open MPI and Intel® MPI. Hence, the remote tree traversal enables us to tune the performance by reducing the communication time enough to increase the impact of latency hiding.
5.2.3 Scalability of Spherical Boundary Distribution with $\mathcal{HSDX}$

In Fig. 5.5, we test the strong scalability at optimal grain size using $\mathcal{HSDX}$ for a large problem of $10^{10}$ particles. It follows that we have an efficient asynchronous communication when remote calls are non-blocking, have tunable granularity and when control is handed over to useful work rather than waiting immediately. To show this, we have integrated and compared several communication protocols within exaFMM in Fig. 5.5. We note that $\mathcal{HSDX}$ is the closest to ideal scaling and has the advantage of fastest time-to-solution since it limits the inter-rack communication penalty on the dragonfly network by solely exchanging data through neighbors. By just looking at Fig. 5.5, it is hard to see that $\mathcal{HSDX}$ is at potential advantage for the exascale era. So we find it crucial to present Table 5.3 that shows a more detailed analysis of the strong scalability. We notice a 6-fold increase in performance gain (from 3.87% to 23.44%) over the corresponding MPI_Alltoallv implementation as more cores are added. The parallel efficiency decreases, however, as the problem gets
smaller while communication overhead prevails. Conventional $O(P)$ communication schemes stop scaling after 2048 nodes (65,536 cores) of Shaheen XC40. According to our largest setup that has an input of $N = 10^{10}$, we have an update rate of approximately $10^9$ particles/second.

PVFMM is a large-scale FMM library that uses a kernel independent implementation, thus widens its target range of applications that require calculation of potential for elliptic kernels [86]. In this experiment, we attempt to compare the strong-scaling performance of PVFMM to our exaFMM branch. It is worth noting that citing independent work is not meant to deem one superior to the other, but on the contrary, it is to give rise to our promising performance boosting strategies that tackle problematic communication and partitioning issues that are likely to arise in the near exascale era. In their most recent reports on PVFMM, Malhotra et al. [86] report perfect scalability up to 256 cores when running the Laplace kernel to compute potentials for $10^8$ distributed on the surface of an ellipsoid. From that point onwards, communication cost starts to grow. They achieve 95% speedup corresponding too about 37% parallel efficiency. We switch to neighborhood collective communication presented in Section 5.1.2 for this comparison, since it vastly reduces network contention by propagating cells through direct neighbors only. Fig. 5.6(a) shows consistent weak-scalability of communication over the conventional MPI_Alltoallv implementation.

The presented approach shows a faster time-to-solution in exaFMM vs. PVFMM when computing 2 billion unknowns as in Fig. 5.7. We cannot claim that scaling will persist indefinitely beyond the depicted number of cores, but when we have an exascale application that requires orders of magnitude larger problems that can fit in the machine’s memory, we have a strong evidence of strong scalability.
5.2.4 Evaluation of Neighborhood Collective Communication using $\mathcal{HSDX}$

Fig. 5.6(a) compares $\mathcal{HSDX}$ using neighborhood collectives to $\mathcal{NBX}$ and MPI_Alltoallv. For the class of problems that constitute a hierarchically sparse data exchange defined in 5.1, $\mathcal{HSDX}$ is asymptotically bounded by the $c_1 \log P$ and $c_2 \log^2 P$. This behavior is shown for the boundary distribution solving Laplace Cartesian FMM kernels with $P = 4$ (order of expansion). However, the figure does not suggest that $\mathcal{HSDX}$ can generally replace its rivals; we still believe that $\mathcal{NBX}$ would outperform our algorithm in the general sparse data exchange, because it has the advantage of both $O(\log P)$
upper bound in addition to the use of a non-blocking barrier and synchronized sends [15] [87].

Fig. 5.6(b) weak scales a small example in order to reduce the effect of non-neighbor communication. The fact that \texttt{HSDX} and Pairwise exchange exhibit similar performance is anticipated since they almost have identical log $P$ behavior in such cases. They seem to lose therein against MPI\texttt{Alltoallv} because of the initialization overhead included in communication time.

5.3 Conclusions

This chapter proposes algorithms that improve data locality, remote data access, and loadbalance of the $N$-body problem. These algorithms contribute to an FMM solver that exploits communication redundancy and computation overlap. We show that Hilbert space-filling curves may not be the most optimal choice to partition boundary domain distributions. \texttt{HSDX} shows good strong and weak scalability for large adaptive hierarchically sparse problems, and falls within proven asymptotic time complexities. Shared memory parallelism is important to utilize resources within a node and to alleviate the problems with MPI resource management; thus we need to consider it in future implementations. We are working on improving \texttt{HSDX} so that it exploits the advantages of \texttt{NBX} to widen its range of use cases. As for application, we are intending to make the presented solver a part of an FMM preconditioner for the Poisson equation, which has variety of applications in diffusive and equilibrium processes in fluid dynamics and many other applications.
Figure 5.7: Strong scaling $2^{31}$ unknowns for sphere distribution and $P = 4$ and comparing exaFMM while using HSDX communication and PVFMM.
Chapter 6

FMM-Accelerated Wave Scattering Solver on Emerging HPC Architectures

We present a boundary integral equation solver for wave scattering suited for many-core processors, which are expected to be the building blocks of energy-austere exa-scale systems, and on which algorithmic and architecture-oriented optimizations are essential for achieving worthy performance. We use the GMRES iterative method and the Fast Multipole Method (FMM) to implement the MatVec kernel. We implement highly optimized kernels for both shared- and distributed-memory architectures and provide various performance models for tuning the task-based tree traversal implementation of FMM. We develop optimal architecture-specific and algorithm-aware partitioning, load balancing, and communication reducing mechanisms to scale up to 6,144 compute nodes of a Cray XC40 with 196,608 hardware cores. Task-based traversal achieves roughly 77% and 60% of the peak single precision floating point performance of a 56-core Skylake Scalable processor and a 72-core Knights Landing processor, respectively. These represent approximately seven-fold speedups compared to the baseline scalar code. We report weak scalability in accordance with the best possible $O(\log P)$ communication complexity and the theoretical scaling complexity of FMM. This results in up to 85% efficiency in strong scaling while computing in excess of 2 billion DoF on the full-scale Cray XC40. The numerical results are converged to within 1.0e-4 relative 2-norm residual accuracy of the analytical solution for the sphere and a multiple flower-shaped scatterer displays comparable algebraic convergence in a more general setting.
6.1 Introduction

The Boundary Element Method (BEM) [88] is one of the most efficient means to compute the numerical solution of the wave scattering problem through relying upon dimensionality reduction. Following the surface discretization, the resulting linear system of equations ($Ax = b$) features a large dense matrix. In order to efficiently solve such a linear system of equations, one can use an iterative method, e.g., GMRES, inside which a fast hierarchical approximation of the Matrix-Vector product (MatVec) can be computed through utilizing the Fast Multipole Method (FMM) [89]. For nonuniform grids, FMM uses an octree-based hierarchical domain decomposition, by which it exhibits near-optimal scalability [28]. In addition, the Multilevel Fast Multipole Algorithm (MLFMA) [90] employs FMM as a low frequency accelerator for the MatVec kernel, which computes integral equations of electromagnetic or acoustic wave scattering [91]. Furthermore, Takahashi et al. [92] utilizes the Inverse FMM (IFMM) [93, 94] to precondition the 3D Helmholtz equation and adopts the low-frequency FMM (LFFMM) [95].

In this chapter, we present an extreme scale implementation of an FMM-accelerated linear solver for wave scattering for the complex 3D Helmholtz Boundary Integral Equation (BIE). FMM is a highly compute intensive algorithm [96] that is portable and adaptable to different levels of parallelism [97], and exhibits a scalable communication [52, 49]. It is thus natural to rely upon such algorithm to accelerate the MatVec kernel to scale the application performance to a large number of tightly-coupled compute nodes. In addition, a broad-spectrum of literature has explored optimizing, tuning, and scaling the non-oscillatory FMM on a wide-range of computing architectures, considering both shared- and distributed-memory settings [98, 99, 100, 101, 102, 103, 104]. On the other hand, the underlying 3D Helmholtz kernel that is described in [105] does not possess a self-contained near or self-singularity treatments, through which the accuracy of the solution, irrespective of the conver-
gence rate, is improved [106]. Therefore, the presented solver transparently handles adjacent and self singularity, while efficiently contending with a wide range of extreme scale performance challenges. A direct application of this solver is the acoustic wave scattering that is driven by arbitrary scattering objects embedded in a 3D medium [107, 108, 109].

Following summarizes our key contributions of the paper:

- Develop an extreme scale FMM-accelerated BIE linear solver for oscillatory kernels that tackles a similar problem to Bruno et al. [110], but exploits the $O(N \log N)$ complexity when high-frequency computations dominate.
- The 3D complex Helmholtz kernels are revisited in light of shared and distributed memory performance to reach the hardware modeled performance. These enhancements result in a fast time-to-solution that compensates for the ill-conditioning due to sub-wavelength breakdown as described by Darve et al. [95].
- Our solver can compute in excess of 2 billion Degrees-of-Freedom (DoF), and can run to the full scale of the Cray XC40 supercomputer with 85% strong scaling’s parallel efficiency.
- The weak scalability is achieved in accordance with the $O(\log P)$ communication complexity and the theoretical scaling complexity of FMM.

The rest of the chapter is organized as follows. In Section 6.2 we provide background on the underlying mathematics of the application code, with further details on the singularity treatment in Appendix A.3. Section 6.3 describes the kernel implementation of the solver and its workflow. Section 6.4 presents the shared-memory optimization means implemented to speedup the single node performance, with more details in Appendix A.4. In Section 6.5 we describe the extreme scale implementation aspects, which include partitioning, load balancing, and communication reducing. Section 6.6 details, for experimental reproducibility, the workload characterizations,
underlying hardware and software stack, and methodologies of designing the performance analysis. Section 6.7 presents our performance evaluation results, with additional details on the convergence characteristics with respect to the mesh refinement in Appendix A.5. Finally, we conclude in Section 6.8 with brief outline of our ongoing work.

6.2 Background

This section outlines the mathematical aspects of our FMM Helmholtz solver, including the formulation of the incident wave scattering, and a concise description of the underlying FMM kernels.

6.2.1 BIE Formulation

Let $S$ represent the surface of a closed scatterer residing in a homogeneous medium. In general, fields of propagation waves in such medium are governed by Equation 6.1, where $U(r,t)$ is the unknown acoustic pressure, $c$ is the speed of sound, and $\nabla^2$ is the Laplacian operator [111].

\[
\nabla^2 U(r,t) - \frac{1}{c^2} \frac{\partial}{\partial t^2} U(r,t) = 0, \quad (6.1)
\]

Given the time-harmonic wave (i.e., $e^{-jwt}$), $U(r,t)$ has the form of Equation 6.2.

\[
U(r,t) = \text{Re}[U_0(r)e^{-jwt}] \quad (6.2)
\]

Inserting Equation 6.2 into Equation 6.1 obtains the Helmholtz Equation 6.3, where $k$ is the wave number.

\[
\nabla^2 U_0(r) + k^2 U_0(r) = 0 \quad (6.3)
\]

Equation 6.4 is the surface integral result of plugging the second form of the
Greens theorem \cite{112} into Equation \ref{6.3}, where \( p(r') \) is the pressure field at the source point \( r' \), \( q(r') = \frac{\partial p(r')}{\partial n} \) is the velocity, \( p^{\text{inc}}(r) \) represents the incident plane wave at the observation point \( r \), and \( G(r, r') \) is the scalar Greens function.

\[
p^{\text{inc}}(r) + \int_{\mathcal{S}} \left[ \frac{\partial G(r, r')}{\partial n} p(r') - G(r, r') q(r') \right] dS' = \frac{1}{2} p(r), \quad r \in \mathcal{S}
\] (6.4)

The scalar Green’s function is represented in Equation \ref{6.5}, where \( R = |\mathbf{R}| = |r - r'| \) is the distance between source and observation points.

\[
G(r, r') = \frac{e^{jkR}}{4\pi R}
\] (6.5)

Equation \ref{6.6} considers the soft boundary condition \cite{113} with \( p = 0 \) in Equation \ref{6.4}.

\[
\int_{\mathcal{S}} G(r, r') q(r') \, dS' = p^{\text{inc}}(r)
\] (6.6)

[Note: The integral equation formulation used herein subjects to internal resonance problem, which is a well-known issue that has been addressed using different approaches, e.g., Combined Helmholtz Integral Equation Formulation (CHIEF) \cite{114} or Burton-Miller formulation \cite{115, 116}. However, a reasonably accurate numerical convergence result can be achieved, even with internal resonance problem, if the excitation frequency does not coincide exactly with the cavity resonance frequencies \cite{117}. In Section \ref{6.7} and Appendix \ref{A.5} this important phenomena of interest is explored and verified.]

\subsection*{6.2.2 Discretization via Nyström Method}

To discretize the scatterer’s surface, we first divide it into curvilinear triangular patches for higher-order geometry modeling. Then, a high-order Nyström method \cite{118, 119} is used to expand the unknown surface velocity. Each curvilinear patch
has \( N_i \) interpolation points defined on the patch. The unknown velocity is expanded as an interpolation based on its values at those points given by Equation 6.7 where \( \vartheta^{-1}(r') \) is the Jacobian at \( r' \), \( L(i,n)(\zeta, \eta) \) is the Lagrange interpolater at \( r' \) calculated in a right triangle system \( u, v \) [118], and \( \{I\}_{(i,n)} \) is the set of unknown expansion coefficients at the \( i^{th} \) interpolation point on the \( n^{th} \) patch.

\[
q(r') = \sum_{n=1}^{N_p} \sum_{i=1}^{N_i} \vartheta^{-1}(r')L(i,n)(\zeta, \eta)\{I\}_{(i,n)}
\] (6.7)

Substituting Equation 6.7 into Equation 6.6 and applying the point-matching testing [118] at the interpolation points leads to a final discretized matrix Equation 6.8, where \([V^{inc}]_{j,m} = p^{inc}(r(j,m)).\]

\[
ZI = V^{inc}
\] (6.8)

The entries of \( Z \) are given by Equation 6.9. \( r(j,m) \) is the \( j^{th} \) interpolation point on the \( m^{th} \) patch: \( j = 1, ..., N_p \) and \( m = 1, ..., N_i \).

\[
[Z]_{(j,m)(i,n)} = \int_{\Delta n} G(r(j,m), r')\vartheta^{-1}(r')L(i,n)(\zeta, \eta)dr'
\] (6.9)

Numerical solution of the BIE with the Nyström method requires an evaluation of the singular integrals either numerically or analytically. Several singularity treatment techniques have been proposed (e.g., polar coordinate transformation [120], singularity subtraction technique [121], and Duffy transformation [106]). In this work, we use the Duffy transformation, since it works well with the weak \( 1/R \) singularity from Equation 6.5 [122]. This method is briefly summarized in Appendix A.3.

### 6.2.3 FMM as an Accelerator for the BIE

From an algebraic perspective, FMM works as a matrix-free accelerator for the matrix-vector multiplication of certain matrices arising from the elliptic PDEs [3] that satisfy
Green’s formula (see Equation 6.5). The solution of the linear system involving \( G(r, r') \) can produce the target vector field (i.e., \( q(r') \) of Equation 6.4) iteratively. As mentioned, the BEM discretization matrix is dense, and thus, the BIE evolves into a summation (Equation 6.7) that is very costly to calculate. Equation 6.10 is an illustrative example that uses FMM to approximate the scattered field or the impulse response due to a monopole source placed inside a closed sphere. \( p(r) \) is the field due to all sources, which represent the reflection of the source within closed domain. \( G(r, r') \) is already defined in Equation 6.5, whereas \( q(r') \) is the strength of the \( j^{th} \) source:

\[
p(r) = \sum_{j}^{N_s} G(r, r')q(r')
\]

(6.10)

Calculating the effect at many target points (e.g., \( N_t \)) results in quadratic complexity, which can be reduced to \( O(N_s + N_t) \) by expanding Equation 6.10 into a series of spherical harmonics [25], as in Equation 6.11. The non-singular part of the solution can be formulated as in Equation 6.12:

\[
p(r) = ik \sum_{n=0}^{\infty} \sum_{m=-n}^{n} S_{-m}^{m}(r_j)R_{n}^{m}(r), r \leq r_q
\]

(6.11)

\[
p(r) = ik \sum_{n=0}^{\infty} \sum_{m=-n}^{n} C_{n}^{m} R_{n}^{m}(r)
\]

(6.12)

\[
C_{n}^{m} = \sum_{r_q < R_{\text{max}}} Q_{Q}S_{-m}^{m}(r_q)
\]

(6.13)

In FMM, the coefficients (also known as the multipoles) \( C_{n}^{m} \) are computed for each point, and the corresponding series is truncated at \( P \ll N \), which is the order of expansion, and thus, the complexity is cut down to \( P^2 \) for a single target.

The FMM implementation used herein is a synergistic combination of ExaFMM
and FMMLIB3D. The 3D Helmholtz Fortran kernels from FMMLIB3D have been reformulated and recoded in C++ to augment hardware-specific performance enhancements. We have incorporated several state-of-the-practice software performance engineering methodologies to squeeze the performance to the hardware limit, by focusing on consolidating various architecture-specific and algorithm-aware optimization means. Figure 6.1 summarizes the underlying FMM’s tree passes integrated herein.

In brief, the wideband spectrum of interactions is divided into low and high frequency regimes where the underlying expansion (i.e., $H$, $J$, and Exponential) is invoked accordingly. After hierarchically decomposing the domain into a quad/oct tree as shown in Figure 6.2, an upward sweep applies P2M and M2M operators to propagate leaf sources into multipoles relying upon the H-expansion. Next, a traverse stage (horizontal pass) is performed using the M2L operator that uses either the high-frequency diagonal translation or the exponential expansion, such that the well-separated cells interact as multipoles or, otherwise, as particles. Finally, a downward pass stage is
employed to translate local expansions to the corresponding target cells, known as the L2P operator. The adjacent neighbors, on the other hand, interact in a sub-quadratic manner using Equation 6.5 inside the P2P kernel.

\[ \text{(a) Original domain before partitioning.} \] \[ \text{(b) Quad-tree partitioning on 4 processes.} \]

Figure 6.2: Particles in a spherical shell 2D domain.

6.3 BEMFMM System Architecture

Having explained the fundamental formulation of the underlying FMM, the acoustics problem, and the 3D complex Helmholtz equation, this section describes the implementation attributes of the presented solver, namely BEMFMM. A high-level representation of BEMFMM is depicted in Figure 6.3 and following describes its key routines.
Mesh Normalizer: One of the severe obstacles that we encounter at the large-scale experiments is reading large input meshes from persistent file systems, e.g., LUSTRE. We, thereby, implement a parallel, system-dependent mesh file handler routine that parses a variety of formatted mesh ASCII files (e.g., GMSH and I-DEAS), along which an FMM-specific partitioning scheme is developed. Our in-house domain decomposition is designed to interface a third-party partitioning tool, e.g., ParMETIS, to carry out the fine- and coarse-grained domain decomposition locally. The partitioning scheme implemented herein possesses a dual-stage routine, as follows:

- A hierarchical, multilevel division approach that partitions the mesh file on the master node (i.e., MPI rank 0), and then splits and broadcasts the files across the available MPI ranks. This approach, however, is limited by a small number of compute nodes, in which it perfectly works up to 64 nodes, each accesses at most 128 GB of memory and about 2 TB of disk storage.

- To overcome the scalability hurdles imposed by the network and memory constraints of the first stage, we develop a preprocessing phase to handle the file distribution beyond 64 MPI ranks. It is an external C++ “call-back” routine that constructs an intermediate binary big-endian mesh file of the original file. The
generated intermediate file is supplemented with extra headers outlining the underlying characteristics of the mesh file distribution. In other words, we abstract out the format-specific data, and translate it into coordinates corresponding to the elements. Each MPI process, thereafter, calculates its offset and reads a block of the data in a collision-free manner prior to the FMM-specific partitioning. Consequently, we conserve a great deal of computational time that is consumed to parse TeraBytes of ASCII numbers. Furthermore, this approach does not require synchronization of file reads or even using MPI I/O. This, in turn, simultaneously load balances the network reads in terms of bandwidth and throughput.

**BEMFMM:** It is the main module that contains the source field discretizer and the interfaces to the underlying numerical iterative solver. [Note: At the current implementation, we link to either our in-house implementation of GMRES, or an interface to PETSc framework [125, 126, 127], and we plan to include various interfaces to different existing numerical libraries targeting different hardware architectures, e.g., ViennaCL [128] or Trilinos [129].] Table 6.1 shows the parameters that control the entire cycle of the solver.
Table 6.1: BEMFMM control parameters.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Values</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>θ and φ</td>
<td>0° − 180°</td>
<td>Spherical coordinates of source/observation points</td>
</tr>
<tr>
<td>µ</td>
<td>2 - 6</td>
<td>Order of underlying triangular mesh</td>
</tr>
<tr>
<td>β</td>
<td>2 - 4</td>
<td>Order of interpolation basis function</td>
</tr>
<tr>
<td>freq</td>
<td>Hertz</td>
<td>Resolution frequency</td>
</tr>
</tbody>
</table>

**Application Specific**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Values</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>ϵ</td>
<td>1.0e²</td>
<td>Iterative residual norm tolerance</td>
</tr>
<tr>
<td>max iter</td>
<td>&gt; 0</td>
<td>Max number of iterations</td>
</tr>
<tr>
<td>ngprcs</td>
<td>≤ 42</td>
<td>Gauss quadrature points for RCS field</td>
</tr>
<tr>
<td>th iter</td>
<td>&gt; 0</td>
<td>Threshold to move from direct to iterative solver</td>
</tr>
</tbody>
</table>

**Solver Specific**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Values</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>threads</td>
<td>&lt; Thread_{max}</td>
<td>Max number of OMP and TBB threads</td>
</tr>
<tr>
<td>ncrit</td>
<td>&gt; 8</td>
<td>Max number cells per leaf</td>
</tr>
<tr>
<td>nspawn</td>
<td>&gt; 8</td>
<td>Min number of cells to spawn a task</td>
</tr>
<tr>
<td>P</td>
<td>&gt; 5</td>
<td>Max number of expansion terms per cell</td>
</tr>
<tr>
<td>accuracy</td>
<td>e²</td>
<td>Desired FMM accuracy</td>
</tr>
<tr>
<td>partition</td>
<td>[0-2]</td>
<td>Geometric,ORB+Hilbert,ParMETIS</td>
</tr>
</tbody>
</table>

**FMM Specific**

**FMM Wrapper:** It is an abstraction on top of the ExaFMM library [54]. The enhancements described in Section 6.4 and Section 6.5 are both integrated into ExaFMM. In addition, ExaFMM implementation utilized herein is built on top of our C++ version of FMMLIB3D.

6.4 Thread-level Parallelism

The well-known linear complexity of FMM is accomplished by mapping positions of tree cells into Morton/Hilbert keys (see Figure 6.4), and probing each cell for its interaction list by interpolation of bits while traversal [130]. Figure 6.4(b) depicts the geometric tree partitioning of cells, which maps positions to binary keys. The line segments represent the original level-wise Hilbert orders. However, maximizing thread-level parallelism has been proven more successful using the Dual Tree
Traversal (DTT) approach [51, 42], which is known for its adaptability to multi- and many-core emerging architectures. For example, to find the scattered field at target positions encoded by 0000 Hilbert order in Figure 6.4(b), DTT simultaneously traverses source and target trees, and recursively uncovers the cell-cell interaction list (see Figure 6.4(a)).

One of the disadvantages of DTT is its sensitivity to granularity (i.e., the minimum number of cells required to spawn a task (s)), and size of tasks (i.e., the maximum number of bodies per leaf cell (c)). Hence, we consider them as tuning parameters in our implementation, and we use them to develop a performance model to comprehend the behaviors of the execution time. [Note: The performance model is visualized by a heatmap in Figure 6.10 and Figure 6.11 of Section 6.7.2]. We believe that it is crucial to consider maximizing locality of references, by which we can obtain many advantages of high concurrency and throughput in latency-bound compute kernels. Equation 6.14 models the optimal parameters (i.e., (s, c)) to fill the L2/L3 caches, and to maximize concurrency, throughput, and cache locality. The multiplier “2” is inclusive of source and target, c is the number of bodies per leaf cell (i.e., size of tasks), k is the task size in bytes (i.e., sizeof(c)), T is the number of hardware threads per core, CSZ/BSZ is the cell/body structure size in bytes (i.e., sizeof(struct cell_t) and sizeof(struct particle_t), respectively),
\( s \) is the task spawning parameter (i.e., granularity), \( \log_2 c \) is the depth of recursive branch, and L2/L3 Last Level Cache (LLC) size in bytes.

\[
\min_{s,c} f(s, c) = (M2L_{size} + P2P_{size}) - \text{L2/L3 Cache} \\
= (2 \times c \times k \times T \times [(CSZ \times \log_2 \frac{s}{c}) + BSZ]) - \text{L2/L3 Cache}
\]  

(6.14)

### 6.5 Distributed-memory Optimizations

Having explained the key shared-memory optimization techniques to improve the application’s single node performance, we elaborate on the main distributed-memory and MPI-based implementation aspects yielding robust mechanisms for extreme scale partitioning, load balancing, and communication reducing.

#### 6.5.1 Partitioning and Load Balancing

Despite the fact that FMM is asymptotically linear in terms of the theoretical time complexity, a naive distributed-memory workload partitioning can take away the advantages of using an optimal algorithm. In general, For N-body codes, the domain has to be decomposed in order to maximize local computations (near-field), while minimizing the volume of global communications. Additionally, the computation and communication ratios are balanced in such a way that large amounts of computation can be carried out between communication events.

**Partitioning Stage:** The core building block of our partitioning implementation for large-scale is the modified Orthogonal Recursive Bisection (ORB) by \([52]\). Since the growth of interaction lists is governed by distribution randomness, workload size, and communication volume cost, obtaining an optimal partitioning is extremely challenging. For example, cells have different sizes of interaction lists, and therefore, equidistributing the results of the ORB might lead to a suboptimal balance of the
workload. Hence, we further improve the efficiency of our partitioning strategy through weighting by workload. In other words, we use the workload size of the previous time step to weight the particles, so that we promote a cost-effective and adaptive load balance across the MPI ranks. Originally, this technique is implemented in the original Hashed Oct-Tree (HOT) [130], but we reshape it and fine-tune it to be applicable and coherent to our application code. One drawback of the weighting scheme is that it only balances the workload, but not the communication volume. Therefore, there have been significant efforts to employ a graph partitioning tool to use the workload as node-weights and communication as edge-weights. This aims to create partitions that maintain both an optimal balance of the workload as well as the communication [65]. Furthermore, this method has only been compared with Morton key splitting without weights. Hence, we balance the workload and communication simultaneously by calculating the weight for the \( i^{th} \) particle \( w_i \) according Equation \ref{eq:w_i}, where \( l_i \) is the local interaction size, \( r_i \) is the remote interaction list size, and \( \alpha \) is a constant that is optimized over the time steps to minimize the total runtime. \( l_i + r_i \) is the total interaction list size and represents the workload, while \( r_i \) reflects the amount of communication. By adjusting the coefficient \( \alpha \), one can amplify/damp the importance of communication balance. Making this an optimization problem to minimize the total runtime is what we prefer over minimizing the load imbalance since the latter is not our final objective. In addition, the variables \( l_i \) and \( r_i \), and the total runtime are already measured in the present code, so the information is available with negligible cost.

\[
\begin{align*}
    w_i &= l_i + \alpha \cdot r_i
\end{align*}
\] (6.15)

Figure \ref{fig:granularity} manifests the granularity spectrum for the partitioning phase. We restrict the partitioning phase to the granularity of a configurable steps (GMRES restart), because it is a too costly process for a stationary system. Also, we have a
limited need for partitioning at fine granularity. We instantiate an artificial FMM partitioning and traverse call before invoking the GMRES solver, in order to apply weighting based on both workload as well as communication (see Equation 6.15).

![Diagram of partitioning](image)

**Figure 6.5:** Granularity of partitioning.

### 6.5.2 Communication Reduction

Once we determine the multipole expansions for every local cell, we pass the multipoles to the necessary processes in a “sender-initiated” fashion [66]. This reduces the latency, since we communicate only once rather than sending a request to remote processes and then receiving the data. Such “sender-initiated” communication schemes are common in cosmological N-body codes, since they tend to use only monopoles.

Figure 6.6 presents the Local Essential Tree (LET) that is formed from the information sent from the remote processes by simply grafting the root nodes of the remote trees. In a conventional parallel FMM code, a global octree is formed and partitioned using either HOT or ORB. Therefore, the tree structure is severed in many places, which complicates the merging of the LET. The merging LET code consumes a large bulk of the total execution time of FMM, and thus, incorporating additional features (e.g., periodic boundary conditions, mutual interaction, more efficient translation stencils, and dual tree traversals) that would affect the runtime. However, we geometrically separate the global tree structure from the local tree structure to merge the tree in a a single time step as shown by Figure 6.6 and are able to incorporate extended features.
While the remote information for the LET is being transferred, the local tree can be traversed. Conventional fast $N$-body methods overlap the entire LET communication with the entire local tree traversal. The LET communication becomes a bulk-synchronous $\text{MPI} \text{.alltoallv}$ type communication. To achieve scalability to all of Shaheen, we embrace the neighborhood-based communication protocol of [52], namely Hierarchical Sparse Data eXchange ($\mathcal{HSDX}$) (see Figure 6.7). The algorithm unfolds into 3 stages, as follows:

1. The MPI communication graph maps the ranks to adjacent nodes by calling $\text{MPI} \text{.Create} \text{.dist} \text{.graph} \text{.adjacent}$. This mapping uses their logical near counterparts from the global tree of Figure 6.6, and each color indicates a single communication route for the LET (e.g., rank 13 in Figure 6.7).
2. Perform communication of the neighboring trees/ranks.
3. Communicate the neighbors of neighbors’ data multiple times, since they are already available at adjacent nodes from previous communication steps.

### 6.6 Experimental Setup and Workload Characterization

This section describes our experimental platforms, the datasets, and the scientific performance engineering methodologies that are used to analyze and present the performance evaluation results.
6.6.1 Software Stack and Hardware Configuration

The source code is written in C++. We use PETSc release version 3.10 built on top of Intel Parallel Studio version 2018 Update 1, which includes Intel C/C++ compiler, Threading Building Blocks (TBB), OpenMP, Cilk Plus\(^1\), MPI, and Math Kernel Library (MKL). We use ParMETIS version 5.0. PETSc scalar type is set to complex. PETSc is complied with the C++ compiler, and the FORTRAN kernels are set to generic for faster complex number performance. All experiments are performed with the -O3 compiler optimization flag, and OpenMP affinity is set to scatter via KMP_AFFINITY=scatter. The pinning and binding of the thread contexts and the MPI ranks are set to target a specific quadrant/tile/core on KNL, and a specific socket/core on CPU. Furthermore, we use numactl Linux command to control binding and interleaving of memory channels. Table 6.2 summarizes the specifications of the Intel x86 architectures considered herein.

For the large-scale experiments, we use KAUST’s Shaheen XC40, the rank 29 supercomputer according to the TOP500’s June 2018 announcement. The system consists of 6,174 compute nodes, each of which is equipped with a dual socket Intel

\(^1\)Intel Cilk Plus is being deprecated in the 2018 release of Intel Software Development Tools.
Table 6.2: Hardware specifications.

<table>
<thead>
<tr>
<th>Family</th>
<th>KNL</th>
<th>Haswell (HSX)</th>
<th>Skylake (SKL)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Model</td>
<td>x200</td>
<td>E5V3</td>
<td>Scalable</td>
</tr>
<tr>
<td>Socket(s)</td>
<td>1</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>Cores</td>
<td>72</td>
<td>32</td>
<td>56</td>
</tr>
<tr>
<td>GHZ</td>
<td>1.50</td>
<td>2.60</td>
<td>2.10</td>
</tr>
<tr>
<td>Watts/socket</td>
<td>245</td>
<td>120</td>
<td>165</td>
</tr>
<tr>
<td>DDR4 (GB)</td>
<td>192</td>
<td>128</td>
<td>264</td>
</tr>
<tr>
<td>Frequency Driver</td>
<td>acpi-cpufreq</td>
<td>acpi-cpufreq</td>
<td>acpi-cpufreq</td>
</tr>
<tr>
<td>Max GHZ</td>
<td>1.50</td>
<td>2.60</td>
<td>2.10</td>
</tr>
<tr>
<td>Governor</td>
<td>conservative</td>
<td>performance</td>
<td>ondemand</td>
</tr>
<tr>
<td>Turbo Boost</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

Haswell CPU (see Table 6.2). The entire system has 197,568 hardware cores, and 786 TB of main memory. (Note that in our experimentations we consider the full scale of Shaheen is 6,144 compute nodes with 196,608 hardware cores, by which we purposely leave 30 compute nodes (i.e., 960 hardware cores) untouched for logistical configurations.) The compute nodes are connected by the Cray Aries interconnect with dragonfly topology, which provides for a maximum of 3 hops for a message between any pair of nodes. Theoretically, Shaheen has a peak double precision floating point performance of 7.2 PFlop/s.

On Shaheen, we use PETSc release version 3.10 built on top of GNU GCC/G++ Compiler version 7.0, GNU OpenMP, and Intel TBB for the GCC Compiler. We do not configure Cilk Plus on Shaheen. We use BLAS and LAPACK open source implementation, and we use Cray MPICH implementation that is built on top of GNU compiler. We use ParMETIS version 5.0. PETSc scalar type is set to complex. PETSc is compiled with the C++ compiler, and the FORTRAN kernels are set to generic for faster complex number performance.

The settings of the experiments are similar to the aforementioned single node settings with negligible differences in the command line variables and threading tunning instructions, which are are adjusted according to the underlying software stack of...
6.6.2 Dataset Description

Throughout the following experiments and unless otherwise clearly stated, we consider a spherical object to scatter an incident uniform plane wave. The sphere’s radius is set to $a = 1m$, and the medium speed of sound is set to $343m/s$. We use a second order curvilinear (curved-triangle) meshes, in which every mesh element (triangle) has 6 quadrature points (unknowns/DoF): 3 points associated with the triangle nodes and 3 points associated with the triangle edges (see Figure 6.8). The meshes are generated via GMSH and I-DEAS. For detailed mesh properties, Table 6.3 describes the specifications of our experimental datasets.

6.6.3 Experimental Setup

To report the most accurate performance measurements irrespective of the hardware states and conditions, we apply several state-of-the-practice scientific performance engineering methodologies, which overcome any possible hardware-oriented performance variations.
Table 6.3: Spherical mesh dataset descriptions.

<table>
<thead>
<tr>
<th>Mesh</th>
<th>Elements</th>
<th>Nodes</th>
<th>Edges</th>
<th>Number of unknowns (N)</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>156</td>
<td>312</td>
<td>468</td>
<td>936</td>
</tr>
<tr>
<td>B</td>
<td>3,156</td>
<td>6,312</td>
<td>9,468</td>
<td>18,936</td>
</tr>
<tr>
<td>C</td>
<td>7,274</td>
<td>14,548</td>
<td>21,822</td>
<td>43,644</td>
</tr>
<tr>
<td>D</td>
<td>14,338</td>
<td>28,676</td>
<td>43,014</td>
<td>86,028</td>
</tr>
<tr>
<td>E</td>
<td>22,370</td>
<td>44,740</td>
<td>67,110</td>
<td>134,220</td>
</tr>
<tr>
<td>F</td>
<td>41,258</td>
<td>82,516</td>
<td>123,774</td>
<td>247,548</td>
</tr>
<tr>
<td>G</td>
<td>60,204</td>
<td>120,408</td>
<td>180,612</td>
<td>361,224</td>
</tr>
<tr>
<td>H</td>
<td>93,590</td>
<td>187,180</td>
<td>280,770</td>
<td>561,540</td>
</tr>
<tr>
<td>I</td>
<td>115,454</td>
<td>230,908</td>
<td>346,362</td>
<td>692,724</td>
</tr>
<tr>
<td>J</td>
<td>159,288</td>
<td>318,576</td>
<td>477,864</td>
<td>955,728</td>
</tr>
<tr>
<td>K</td>
<td>250,514</td>
<td>501,028</td>
<td>751,542</td>
<td>1,503,084</td>
</tr>
<tr>
<td>L</td>
<td>314,212</td>
<td>628,424</td>
<td>942,636</td>
<td>1,885,272</td>
</tr>
<tr>
<td>M</td>
<td>374,360</td>
<td>748,720</td>
<td>1,123,080</td>
<td>2,246,160</td>
</tr>
<tr>
<td>N</td>
<td>1,497,440</td>
<td>2,994,880</td>
<td>4,492,320</td>
<td>8,984,640</td>
</tr>
<tr>
<td>O</td>
<td>5,989,760</td>
<td>11,979,520</td>
<td>17,969,280</td>
<td>35,938,560</td>
</tr>
<tr>
<td>P</td>
<td>23,959,040</td>
<td>47,918,080</td>
<td>71,877,120</td>
<td>143,754,240</td>
</tr>
<tr>
<td>Q</td>
<td>95,836,160</td>
<td>191,672,320</td>
<td>287,508,480</td>
<td>575,016,960</td>
</tr>
<tr>
<td>R</td>
<td>383,344,640</td>
<td>766,689,280</td>
<td>1,150,033,920</td>
<td>2,300,067,840</td>
</tr>
</tbody>
</table>

The reported runtime results are summarized using the arithmetic mean across multiple independent runs, which form the sample space. The reported floating point rates, on the other hand, are summarized using the harmonic mean \([131, 132]\). Unless otherwise reported, we average approximately 50 runs for every experiment, except for the large-scale results, in which we are constrained by the available core hours. Thus, we reduce the size of the sample space for every run based on the available core hours (i.e., we roughly average between 5 to 10 runs for every experiments, based upon the problem size and the wall-clock time of a specific run). In addition, an error bar is drawn to show the +/- standard deviation of the mean for each experimental sample.
6.7 Performance Evaluation Results

We describe in this section the single- as well as the multi-node performance evaluation results for the acoustics application.

6.7.1 Data-level Parallelism Results

The most floating point intensive portion of our application code is the MatVec kernel, which is accelerated by FMM. To this end, speeding up such kernel increases the entire application’s performance, thus maximizing the performance at extreme scale. Figure 6.9 shows the single precision floating point performance of the Helmholtz P2P kernel on both Intel KNL as well as Skylake across different meshes. Relying on the Intel compiler’s auto-vectorization, the best performance we achieve on a single node of Skylake is roughly 5.7 TFLOP/s. Since the peak single precision floating point performance of our used edition of Skylake is roughly 7.5 TFLOP/s, our optimized P2P kernel obtains 77% out of the peak. Handwritten vectorization via AVX-512 intrinsics, on the other hand, achieves at most 27% (i.e., approximately 2.0 TFLOP/s). In contrast, the non-vectorized version gains a maximum of 14% (i.e., roughly 1.0 TFLOP/s), which means that our optimized version maintains an average 5.4x speedup relative to the scalar code. We observe a similar behavior on KNL, where the auto-vectorization achieves at most 60% (i.e., approximately 4.5 TFLOP/s) of KNL’s single precision peak floating point performance (i.e., roughly 6.9 TFLOP/s). This is 10x speedup, on average, compared to the scalar code (roughly 0.43 TFLOP/s).

In the handwritten AVX-512 intrinsics code, we explicitly reference the auxiliary fields data structure (cell_t) that includes the coordinates, source, and target values using _mm512_setr_ps intrinsic. The explicit setting of the vector register values is carried out since the (cell_t) data structure is allocated as an Array-of-Structs (AoS), which is a well-known harmful approach to the code vectorization. Hence, the
explicit vectorization does not improve much over the scalar code. In addition, the compiler initially fails to auto-vectorize the code as a consequence of the assumed data dependency resulting from the AoS allocation. One solution to this is to change the struct allocation of the code \[58\]. However, this can be a daunting proposition, in which the whole ExaFMM code needs to be adjusted to be compatible. This would also result in a significant performance reduction due to the loss of cache locality of references, an advantage granted by using AoS in conjunction with Morton orders. The other approach followed here, as explained earlier in the chapter, is to simplify the P2P kernel to allow the compiler to generate an efficient vector code. In this approach, we rely on the compiler to find an efficient way to deal with the hurdles of the AoS allocation. As a result of such a kernel rewrite, the compiler manages to generate an efficient code that achieves roughly 77% out Skylake’s peak, and nearly 85% out of the Intel MKL SGEMM\(^2\) performance on Skylake (i.e., 6.7 TFLOP/s). To verify how AoS data is loaded, we scrutinize the generated assembly vector code, and we find out that the compiler performs multiple vector strided loads with a stride size equal to the size of the SIMD lane. This approach efficiently deals with the AoS allocation, but it requires very complex assembly and intrinsics coding \[58, 131\]. Thus, the fact that the compiler manages to generate such code, indeed with the help of our subtle kernel simplifications, saves time and effort. Consequently, it protects

\(^2\)SGEMM is the Single Precision real valued GEneral Matrix-Matrix Multiplication kernel.
against writing error-prone and non-portable kernel codes.

### 6.7.2 Thread-level Parallelism Results

Figure 6.10 shows the runtime performance of the traversal Helmholtz kernel on KNL and Skylake with varying the grain ($y$-axis) and cell sizes ($x$-axis). If the grain size ($x$) is less than or equal to the summation of bodies enclosed within the source and target, then a task will be spawned by the work scheduler. Essentially, this means having smaller grain sizes would fork more threads, which implies creating a fine-grained thread pool. However, as manifested from Figure 6.10 having a coarse-grained thread pool with a smaller number of tasks, the performance improves. Similarly, having a smaller size of a task ($y$), which is the number of bodies within a leaf cell, exhibits better performance. Thus, it is important to think of processing not so much in terms of long functional threads but smaller sized tasks that can be handed to a thread pool. In addition, a careful number of generated tasks must be considered to avoid overfilling the scheduler thread pool.

![Figure 6.10](image.png)

**Figure 6.10:** Tuning the threading parameters performance of the traversal Helmholtz kernel. Performance is normalized by the arithmetic mean of the runtime.

To further illustrate the findings of Figure 6.10, Figure 6.11 presents the percentage differences between the utilized and available LLC of a specific architecture (i.e., 36MB aggregated L2 cache on KNL, and 38MB L3 exclusive (non-inclusive) cache on Skylake). This is solving 3,156 mesh elements with $P = 10$, which is not com-
putationally challenging, but exposes the highest level of concurrency due to the low task wall time and overhead. We notice a highly accurate prediction of the optimal parameters \( s = 64, 128, 256, c = 128 \) according to the lowest color gradient, which is valid on both architectures.

![Table](image)

Figure 6.11: Tuning the threading parameters performance of the traversal Helmholtz kernel. Performance is normalized by the runtime and the size of the L1/L2/L3 caches on Skylake, and the size of the L1/L2 caches on KNL.

### 6.7.3 Distributed-memory Parallelism Results

There are two main goals for this section. The first is to illustrate that the low-level, architectural-specific shared-memory optimizations employed by this work continue to provide similar performance benefits as we scale to a large number of compute nodes. The second goal, on the other hand, is to demonstrate the performance of the distributed-memory optimizations that enhance workload partitioning and communication load balancing. Also, we depict the scaling properties of our application, establish the scaling limits, and further study the benefits of data- and thread-level parallelism within a single compute node in the context of MPI+Threads+SIMD hybrid programming paradigm [133].

We observe that for certain heavy MPI collectives (i.e., MPI_alltoallv and MPI_Broadcastv), Cray MPICH obscurely fails with segmentation faults whenever the amount of data exceeds the \( 2^{31} \) bytes limit. Such failure happens even though the “count” argument of MPI routines, indicates that any array of MPI_Data_Type
up to $2^{31}$ elements is allowed. Therefore, in our implementation, we preemptively breakdown the collectives routines to multiples of 2 GB of absolute size to eliminate the opportunity for such non-deterministic behaviors.

In order to wisely exploit the hierarchical interconnect mode of operation of drag-onfly network of Shaheen, we tune the network topology parameters of Slurm system [134]. We trigger hierarchical usage of the network to reduce contention, and we adjust the number of the network switches for every job request heuristically based on the underlying nature of every experiment.

### 6.7.3.1 Large-scale Sanity Check

Evaluating the performance of a parallel algorithm can be convoluted. There are many fundamental challenges pertaining to tightly-coupled complicated hardware, compatibility of the underlying software stacks, theoretical complexity and scaling limits of certain algorithms, and the inadequacy of available performance counters for different execution stages of certain algorithms. In the large-scale context, performance metrics such as speedup and parallel efficiency are natural means of evaluating the performance of certain parallel algorithms on specific hardware systems. However, these metrics are essentially related to a predefined base case, and thereafter, they do not reflect the best possible appraisal of the upper- and lower-bound of the algorithmic complexity.

FMM is well-known for reducing the theoretical complexity of MatVec from $O(N^2)$ to $O(N)$ or $O(N \log N)$, depending upon the frequency band. Therefore, we carry out a “sanity check” to verify the theoretical complexity of our FMM Helmholtz implementation at large-scale settings. Figure 6.12 shows the data scaling aspects in practice of FMM kernels with different problem sizes on 1,024 compute nodes of Shaheen. The error bars fall within less than 1 standard deviation away from the arithmetic mean of the sample space size. Furthermore, within a 95% confidence
interval, our utilized FMM Helmholtz implementation is bounded between linear and logarithmic linear time complexity.

![Figure 6.12: Scalability results on 1,024 compute nodes of Shaheen. Performance is normalized by the FMM time per linear iteration and the total number of GMRES iterations.](image)

### 6.7.3.2 Communication Reduction and Load Balancing Results

Communication reducing and balancing using both $HSDX$ and repartitioning techniques adopted from Section 6.5.1 have a vital effect on the cumulative and absolute time of the global tree data exchange. The experiments in Figure 6.13(a) and Figure 6.13(b) depict the communication time before and after optimizations for 1,024 nodes and Mesh Q from Table 6.3. The jittery lines in Figure 6.13(a) represent the imbalance that is smoothed out after repartitioning in Figure 6.13(b). We also observe a roughly 1.8x speedup in time due to localizing the communication within the Aries Network using $HSDX$ from Figure 6.7. The cumulative communication time worsens in the case of having smaller node count (e.g., 128 nodes), as we witness a
6-fold improvement when re-balancing is triggered in Figure 6.13(c).

![Graphs and Diagrams]

(a) Initial tree communication time.  
(b) Time after communication enhancements.  
(c) Effect of our optimization techniques on the cumulative communication time on 128 nodes.

Figure 6.13: Communication reduction and balancing.

6.7.3.3 Strong and Weak Scalability Studies

In Figure 6.16, we present weak and strong scalability results on the Shaheen Cray XC40 supercomputer from 1 compute node with 32 hardware cores up to 6,144 compute nodes with 196,608 hardware cores on different mesh refinements. The largest problem involves more than 2 billion DoF.

The cost of our large-scale wave scattering solver is overwhelmingly dominated by the cost of the underlying FMM Helmholtz kernels that are implicitly invoked by every GMRES iteration to perform a fast MatVec.

**Weak Scalability Study:** In this study depicted by Figure 6.16(a), we report the performance and convergence time across different mesh refinements. Table 6.4
shows the weak scalability experimental settings. At every mesh refinement step, we make sure that $\lambda = 5h$, i.e., 5 edges per wavelength. Thus, we circumvent over-decomposition when adding more compute resources.

Table 6.4: Weak Scalability results on Shaheen from 1 to 4,096 Intel Haswell compute nodes. $T_{mv}$ is the single MatVec time. FMM is configured with $e^{-03}$ accuracy. $\epsilon_{solution}$ is the solution error, whereas $\epsilon_{far}$ is the scattered field’s error at $\infty$ when sampling 361 points at ($\theta = [0^\circ : 0.5^\circ : 180^\circ]$). [Note: In the last two cases, we configure GMRES with max_iter=860 to avoid running out of core hours, since the solver takes great deal of time-to-solution in order to converge.]

<table>
<thead>
<tr>
<th>Cores</th>
<th>Mesh</th>
<th>$f$ (KHz)</th>
<th>Iter</th>
<th>$T_{mv}$ (s)</th>
<th>$\epsilon_{solution}$</th>
<th>$\epsilon_{far}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>32</td>
<td>H</td>
<td>3.720</td>
<td>851</td>
<td>43.0764</td>
<td>3e-02</td>
<td>4e-03</td>
</tr>
<tr>
<td>128</td>
<td>M</td>
<td>7.440</td>
<td>993</td>
<td>44.8185</td>
<td>1e-02</td>
<td>2e-03</td>
</tr>
<tr>
<td>512</td>
<td>N</td>
<td>14.880</td>
<td>1,123</td>
<td>46.7767</td>
<td>2e-02</td>
<td>3e-03</td>
</tr>
<tr>
<td>2,048</td>
<td>O</td>
<td>29.760</td>
<td>1,308</td>
<td>48.4761</td>
<td>2e-02</td>
<td>3e-03</td>
</tr>
<tr>
<td>8,192</td>
<td>P</td>
<td>59.521</td>
<td>1,523</td>
<td>47.8685</td>
<td>3e-02</td>
<td>6e-03</td>
</tr>
<tr>
<td>32,768</td>
<td>Q</td>
<td>119.043</td>
<td>853</td>
<td>50.6201</td>
<td>1e-01</td>
<td>1e-02</td>
</tr>
<tr>
<td>131,072</td>
<td>R</td>
<td>238.086</td>
<td>860</td>
<td>52.4421</td>
<td>2e-01</td>
<td>3e-02</td>
</tr>
</tbody>
</table>

Assessing an FMM-based accelerated solver requires a careful consideration to the fact that the most optimal communication scales as $O(\log P)$ \[52\]. Hence, given the communication complexity, we achieve near-optimal parallel efficiency of FMM per linear iteration as we refine the mesh, and respectively, increase the hardware resources and the frequency. In addition, to illustrate the effect of the global FMM tree communication on the scalability, Table 6.5 presents the critical points, in which we could experience performance degradation based on the Shaheen’s network. The intra-node communication is expected to slow down as we move up across different network units, with hops being the costliest. Despite the fact that the communication and the computations are overlapped in our FMM implementation, the communication effects cannot be completely neglected. Furthermore, the weak scalability results of Figure 6.16(a) manifests a logarithmic growth in the amount of the diagonal and off-diagonal data exchange. Nonetheless, since we employ our optimized communication protocol, namely $HSD\chi$, all tree data exchanges are restricted only by the neighboring compute nodes. Thus, the overhead of moving data across the dragonfly all-to-all network groups of Shaheen does not affect the parallel efficiency, which is
represented by the red numerical labels in the Figure 6.16.

Table 6.5: Characteristics of Shaheen’s XC40 dragonfly network.

<table>
<thead>
<tr>
<th>Level</th>
<th>Hardware/Network Unit</th>
<th>Nodes</th>
<th>Cores</th>
<th>Cores (Overhead)</th>
<th>Hops</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Socket</td>
<td>1</td>
<td>16</td>
<td>32</td>
<td>N/A</td>
</tr>
<tr>
<td>2</td>
<td>NUMA XC40 Node</td>
<td>1</td>
<td>32</td>
<td>64</td>
<td>N/A</td>
</tr>
<tr>
<td>3</td>
<td>Blade</td>
<td>4</td>
<td>128</td>
<td>256</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>Chassis</td>
<td>64</td>
<td>2,048</td>
<td>4,096</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>Cabinet</td>
<td>192</td>
<td>6,144</td>
<td>8,192</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>Local all-to-all Group</td>
<td>384</td>
<td>12,288</td>
<td>16,384</td>
<td>1</td>
</tr>
<tr>
<td>7</td>
<td>Global all-to-all Group 1</td>
<td>2,304</td>
<td>74,728</td>
<td>131,072</td>
<td>2</td>
</tr>
<tr>
<td>8</td>
<td>Global all-to-all Group 2</td>
<td>6,174</td>
<td>197,568</td>
<td>N/A</td>
<td>3</td>
</tr>
</tbody>
</table>

In conclusion, we weak scale up to 4,096 compute nodes as we manage to solve around 2 billion DoF system in about 52 seconds per GMRES iteration, which is on average 50 million DoF per second. Numerically, the problem becomes more ill-conditioned as the problem size is increased, even though a convergent behavior is still observed when comparing to the analytical solution at infinity ($\epsilon_{\text{far}}$) and on the spherical surface ($\epsilon_{\text{solution}}$). In fact, the underlying wideband FMM implementation incorporates specific expansions for low and high frequencies [1], and therefore, converging at 1e-04 ($\epsilon_{\text{far}}$) and 1e-02 ($\epsilon_{\text{solution}}$). In addition, our convergence results are reasonably accurate, which evidently means that the excitation frequency does not coincide exactly with the cavity resonance frequencies [117]. Nevertheless, internal resonance problem affects the number of iterations, and thus, it gets higher with the increasing frequency (or sphere radius in terms of wavelength). [Note: We construct our numerical experiments so that the radius of the sphere (in terms of the wavelength) is increased with the number of discretization elements (i.e., the sampling/resolution in space stays the same ($\lambda = 5h$, i.e. 5 edges per wavelength)). In other words, the frequency is increased with the number of discretization elements so that the sampling rate in space stays the same. Furthermore, the iterative solver, i.e., GMRES, stops when it reaches a residual error of 1e-04 (except for the largest 2 experiments where we only test scalability and fix the max iterations to 850 iterations).
We also maintain the level of FMM accuracy between simulations. As expected, the number of iterations increases significantly as frequency is increased. To clarify our findings, we also report on the accuracy of the solution as well as the accuracy of the far field (computed using the solution), with respect to analytical Mie-series solution, to demonstrate that our FMM implementation is error controllable. In addition, the complexity of FMM is proven to scale as $O(N\log N)$ for this numerical experiment. This is supported by our usage of FMMLIB3D that reports similar complexity [1]. To this end, our FMM-accelerated wave scattering solver is among the fastest and most scalable FMM solvers for oscillatory kernels (i.e., 3D Helmholtz kernels [135, 136]).

**Strong Scalability Study:** We perform the study on every mesh of the weak scalability individually (see Figures 6.16(b) through 6.16(h)). The two largest meshes are scaled up to the full number of the available compute nodes of Shaheen (i.e., 6,144). We carry the strong scaling far enough to show where the stagnation sets in at large scale, which is expected as the problem size per node gets smaller and the communication time becomes dominate. Additionally, since the data has to travel across the dragonfly network units or all-to-all hops (see Table 6.5), the downturn in the parallel efficiency is foreseen. In other word, as we travel past the first all-to-all group, a consistent performance instability is experienced, especially after level 6 (i.e., 16,384 cores) in Figures 6.16(b) 6.16(c) 6.16(d) and 6.16(e). Similarly, propagation within a local all-to-all group, up to three cabinets, may adversely affect the performance. For example, observe the performance beyond 256 hardware cores in Figure 6.16(h). Nonetheless, the performance subtleties in such cases are almost negligible, since the communication is mostly hidden within the local computations at the lower compute core counts.

Figure 6.14 studies the proportional contribution of each FMM kernel to the execution time of a single MatVec in strong scaling settings. Although, the bars should have equal heights (Threads $\times$ Time[s]), the observed overhead is a consequence of
Figure 6.14: Breakdown of computation and communication times across 2 different problem sizes on a single SKL node and 4,096 nodes of Shaheen XC40. $\lambda = 5h$, $P = 30$ and $\epsilon_{FMM} = 1.0e^{-03}$.

LET building and communication. In addition, it does not surpass the execution time of M2L and P2P together, and it can be successfully overlapped with the horizontal pass shown in Figure 6.1. Figure 6.14(a) is the result of scaling within a single Skylake compute node (MPI + 2 Threads). We observe a similar behavior across 4,096 nodes of XC40, when we reproduce the experiment in Figure 6.16(d) using Figure 6.14(b).

6.7.4 Convergence Aspects and Numerical Error

This experiment demonstrates the accuracy of the developed FMM kernels inside the GMRES iterative solver for acoustic wave scattering. It is performed for the single layer kernels associated with Helmholtz equation. Surfaces are discretized with second order curvilinear-isoparametric quadrilateral elements. The use of second order curvilinear elements is crucial for the accurate implementation of a high order solution. In addition, applying a second order scheme to large elements with high order basis functions is characteristically more accurate, in which the slope of the error curve becomes steeper as we increase the basis order. Nevertheless, accuracy may suffer due to the geometry approximation if surfaces are not accurately modeled. We demonstrate this phenomenon of interest in Appendix A.5.

To motivate the prospective numerical enhancements of this work highlighted
in Section 6.8, Table 6.6 presents the convergence behaviors for different mesh refinements and wavelengths on SKL. \( \text{iter}_\beta \) signifies the advantage of using \( p \)-mesh refinement for interactions within \([R < \text{near}]\) to reduce the number of iteration. We observe that FMM’s performance is agnostic to the distribution on a single node. To a certain degree, this is not anticipated at a large scale due to the nondeterministic time spent in global tree communication. Since we remark that the iteration count increases with larger \( N \), integrating a domain preconditioner, inside which FMM is invoked, and treating hyper-singularity is evidently the right path to robustify and fasten the convergence of the linear solver.

Table 6.6: Convergence results of various problem sizes and wavelengths on a single Intel Skylake node. \( \text{iter}_\alpha \) is the number of iterations without quadrature refinement, as opposed to \( \beta \).

<table>
<thead>
<tr>
<th>( N )</th>
<th>( \lambda )</th>
<th>( \text{iter}_\alpha )</th>
<th>( T_{mv} )</th>
<th>( \text{iter}_\beta )</th>
<th>( T_{mv}(s) )</th>
<th>( P_{Max} )</th>
<th>( \epsilon_{\text{solution}} )</th>
<th>( \epsilon_{\text{far}} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>18,936</td>
<td>15h</td>
<td>25</td>
<td>0.2</td>
<td>26</td>
<td>0.25</td>
<td>10</td>
<td>9.7e-03</td>
<td>3.1e-04</td>
</tr>
<tr>
<td></td>
<td>10h</td>
<td>45</td>
<td>0.28</td>
<td>46</td>
<td>0.3</td>
<td>10</td>
<td>1.5e-02</td>
<td>2.1e-03</td>
</tr>
<tr>
<td></td>
<td>5h</td>
<td>94</td>
<td>0.35</td>
<td>91</td>
<td>0.37</td>
<td>15</td>
<td>3.2e-02</td>
<td>4.4e-03</td>
</tr>
<tr>
<td>86,028</td>
<td>15h</td>
<td>54</td>
<td>0.6</td>
<td>53</td>
<td>2.6</td>
<td>10</td>
<td>1.03e-02</td>
<td>2.9e-03</td>
</tr>
<tr>
<td></td>
<td>10h</td>
<td>94</td>
<td>1.15</td>
<td>68</td>
<td>2.6</td>
<td>15</td>
<td>3.0e-02</td>
<td>1.0e-03</td>
</tr>
<tr>
<td></td>
<td>5h</td>
<td>342</td>
<td>5.01</td>
<td>294</td>
<td>5.9</td>
<td>30</td>
<td>2.2e-02</td>
<td>3.1e-03</td>
</tr>
<tr>
<td>134,220</td>
<td>15h</td>
<td>51</td>
<td>1.5</td>
<td>50</td>
<td>5.9</td>
<td>15</td>
<td>3.2e-03</td>
<td>4.05e-04</td>
</tr>
<tr>
<td></td>
<td>10h</td>
<td>114</td>
<td>2.5</td>
<td>113</td>
<td>6.5</td>
<td>20</td>
<td>8.6e-03</td>
<td>2.7e-03</td>
</tr>
<tr>
<td></td>
<td>5h</td>
<td>179</td>
<td>13.8</td>
<td>175</td>
<td>14.1</td>
<td>40</td>
<td>1.5e-02</td>
<td>5.9e-02</td>
</tr>
</tbody>
</table>

In addition, Figure 6.15(a) simulates the velocity solution vector (Equation 6.7) on a spherical plane.

6.7.4.1 Convergence of Scattering by Multiple Objects

Table 6.7 shows the convergence aspects of scattering by multiple flower-shaped objects, which entails a slightly more challenging distribution (Figure 6.8(b)) with respect to our original settings. The performed study uses self-convergence while increasing the mesh resolution to get a more accurate solution. The frequency is kept constant \((f = 150Hz)\) Figure 6.15(b) is the visualized scattered field on the surface
6.8 Concluding Remarks and Future Work

We summarize the progress of this contribution towards migrating frequency-domain scattering to contemporary extreme architectures on a path to exascale:

- Implementation of a scattering solver for complex 3D Helmholtz including:
  - A numerical iterative linear solver based upon GMRES that uses FMM as a fast and accurate MatVec kernel.
  - Treatment for the self- and near-singularity of the sound-soft acoustics solver based on the Duffy transformation.

<table>
<thead>
<tr>
<th>N</th>
<th>iter</th>
<th>$T_{mv}$ (s)</th>
<th>$P_{max}$</th>
<th>$\epsilon_{self}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>9504</td>
<td>51</td>
<td>0.18</td>
<td>10</td>
<td>9.0809e-04</td>
</tr>
<tr>
<td>58320</td>
<td>186</td>
<td>0.42</td>
<td>10</td>
<td>2.6958e-04</td>
</tr>
<tr>
<td>262728</td>
<td>340</td>
<td>1.02</td>
<td>10</td>
<td>8.1249e-05</td>
</tr>
<tr>
<td>430344</td>
<td>750</td>
<td>1.31</td>
<td>10</td>
<td>4.4383e-05</td>
</tr>
<tr>
<td>872784</td>
<td>894</td>
<td>3.5</td>
<td>15</td>
<td>–</td>
</tr>
</tbody>
</table>
• Low-level optimization and fine-tuning for the shared-memory performance that address different emerging HPC architectures, including:
  – Optimal data-level parallelism through efficient vectorization.
  – Specific data structure allocation and striding to enhance the compiler’s optimization.
  – Yardsticks for writing SIMD-friendly, low-level FMM Helmholtz kernel codes.
  – Fine-tuned thread-level parallelism for the traversal kernels of FMM.
  – Predictive performance models for selecting the task-based parameters to tune the threading performance.

• Efficient large-scale architecture-specific and algorithm-aware implementation for distributed-memory parallelisms that manifests:
  – Adaptive nonuniform partitioning and load balancing schemes.
  – Scalable communication reducing protocols.
  – A large-scale performance model based on Cray’s dragonfly network topology.

• Near-optimal parallel efficiency of strong and weak scalability up to 6,144 compute nodes of a Cray XC40 with 196,608 Intel Haswell cores.

• Data scalability study to demonstrate the theoretical complexity of FMM.

• Solution of a 2 billion DoF systems of second order curvilinear triangular patches of a mesh to achieve a relative 2-norm residual accuracy of 1.0e-4.

• Single precision floating point performance of Skylake and KNL of about 68.5% of the theoretical peak, which represents a speedup of roughly 7.0x over the out-of-the-box compilation.

**Ongoing Work:** For future considerations, we are currently extending this work to address the followings:
• Sound-hard Acoustics at Scale: Duffy transformation perfectly handles weak singularity imposed by the integral of the Green’s function of Equation 6.4. However, it does not target the strong singularity \((1/R^2)\), which arises from the integral of the gradient of the Greens function. Considering sound-hard acoustics requires a special singularity treatment. At the moment, we are extending our code to include the computation of the double gradient formulation to handle sound-hard cases at scale. Consequently, we will be implementing different highly nonuniform domains, and adapting to more complex geometries, e.g., wing-fuselage configuration emulated by two intersecting ellipsoids.

• Exploring the possibilities of using an efficient and scalable preconditioning method alongside FMM to effectively reduce the number of iterations, e.g., Block Jacobi preconditioner.

• Coupling Helmholtz FMM with different boundary conditions, e.g., Quadrature by Expansion (QBX) [137, 138].

• Addressing the performance challenges of more heterogeneous HPC architectures, e.g., GPU and ARM.

In summary, the main focus of this chapter lies in the development of a highly scalable FMM that can be efficiently applied to the computation of the Helmholtz integral equation kernel. Particularly, our work deals with addressing the parallel challenges of such application, especially at extreme scale settings, with emphasis on both shared- and distributed-memory performance optimization and tuning on emerging HPC infrastructures. We believe that increasing the convergence rate of the iterative solver by using a preconditioner or switching to a well-conditioned integral equation formulation might be considered beyond the scope of this chapter. However, it is definitely in the roadmap of enhancing the stability of the numerical solver as it is a bold item of the aforementioned ongoing work. The crux of the matter is that the current contributions of the paper are yardsticks en route to efficiently
supplement our solver with additional features, e.g., handling sound-hard cases at scale and preconditioning.
Figure 6.16: Weak and strong scalability results on Shaheen from 1 to 6,144 Intel Haswell compute nodes. Performance is normalized by the FMM time per linear iteration and the total number of GMRES iterations. Numbers along the graph lines (red) indicate efficiency with respect to the ideal speedup (efficiency baseline is the smallest compute node count).
Chapter 7

Concluding Remarks

7.1 Summary

This thesis presents approaches that adapt FMM to upcoming exascale supercomputers. In a many-core environment, it is noted that the use of SIMD intrinsics can be avoided by analyzing the location of inserted #pragma directives, and arranging strided loop accesses to maximize spatial and temporal locality. The choice of vectorizing inner or outer loops for $N^2/P^2$ kernels depends on various factors that include the data access pattern, the data layout and the tile size. Generally speaking, when FMM is configured with a large leaf cell size, L1/L2 cache can be sufficiently reused within the inner-most loop, and it is recommended to vectorize the outer loop since the cost of SIMD load/store operations can be amortized by doing more computations within an $I$ iteration.

A key ingredient to reducing communication is an efficient domain-decomposition scheme that maintains data locality and reduces the overhead of halo exchange. The conventional wisdom of using space-filling curves to indefinitely partition the $N$-body problem is discovered to have a weakness that will appear primarily in boundary element distributions, which constitute the majority of FMM’s use cases. Therefore, this thesis proposes a hybrid partitioning approach that takes advantage of the existing space-filling ordering to hash the local tree, and ORB to carefully choose bisectors in order to reduce partition-cell misalignment, and graft the global tree in constant time. Furthermore, the effect of asynchronous communication is explored by mak-
ing Charm++ handle the overlap with local tree computation. It is found that this only helps at large scale, but more work needs to be done when communication is not an issue, i.e., at a small core count. Abstractly expressing FMM’s communication as point-to-point (local-to-remote) tree dependencies using the STAPL Skeleton Framework has a positive impact on scalability compared to the low-level MPI implementation.

Last but not least, an algorithmic reconsideration of FMM’s tree communication is found to be the most effective and portable due to the current dependence on bulk-synchronous message passing. Tuning communication granularity enables reductions in the effect of bulk-synchronous exchanges. This consequently enables latency hiding irrespective of the problem size. Upper bounded by $O(\log^2 P)$ and for inputs in excess of 9 orders of magnitude, $\text{HSDX}$ outperformed $\text{NBX}$ and MPI-Alltoallv by inherently reducing network contention via MPI 3.0 neighborhood collectives within Shaheen’s dragonfly network. Hypercube pairwise exchange has been implemented for comparative purposes since it directly mimics an $O(\log P)$ communication style and works with nonuniform distributions. It performs just as good as $\text{HSDX}$ in small examples when $\text{HSDX}$ is around its log $P$ lower bound.

### 7.2 Future Work

In the future, we plan to adapt $\text{HSDX}$ to other hierarchical algorithms such as multigrid and $\mathcal{H}$-matrices. For example, at the coarsening stage of the multigrid method, data can be aggregated hierarchically \cite{139}; hence, it can be expressed using $\text{HSDX}$. We will work on finding a deeper insight, besides profiling, on the correct granularity of task spawning within FMM’s dual-tree traversal based on $P$ (order of expansion) and $N$. We will also extend on our MIC work by benchmarking and studying the trade-offs between MPI-only and MPI+X programming paradigms within a single KNL and SkyLake, and we plan to optimize and explore the scaling of FMM across
high-bandwidth Intel Omni-Path Fabric interconnect. One of the key application challenges is to scale up scattered-field arising from double-layer potentials. Hence, we are embedding the solver with parallel translation operators for monopoles and dipoles, in which $1/R^2$ hyper-singularity will be corrected using [140]. The completed code will be packaged into an open-source C++ library called BEMFMM.

**Acknowledgments**

Support in the form of computing resources was provided by KAUST Extreme Computing Research Center, KAUST Supercomputing Laboratory, KAUST Information Technology Research Division, Intel Parallel Computing Centers, and Cray Supercomputing Center of Excellence. In particular, the authors are very appreciative to Bilel Hadri of KAUST Supercomputer Laboratory for his great help and support throughout scalability experiments on the Shaheen supercomputer.
REFERENCES


Mathematical Foundation for Helmholtz Green’s Function

A.1 Helmholtz Equation

The scalar Helmholtz equation is

\[ \nabla^2 \psi(r) + k^2 \psi(r) = 0 \]

\[ \nabla^2 = \frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2} + \frac{\partial^2}{\partial z^2} \] (A.1)

where \( \psi(r) \) is a complex scalar function defined for \( r = (x,y,z) \in \mathbb{R}^3 \), and \( k \) is some real or complex constant. It takes its name from the German scientist Hermann von Helmholtz (1821 - 1894), with significant contributions to acoustics, electromagnetics, and hydrodynamics. The Helmholtz equation can be derived from the conservation laws of physics and is the time-harmonic form of the wave equation. From an algebraic notion, it can be seen as an eigenvalue problem for the Laplacian.

A.1.1 Simple Acoustics

Acoustic waves are, by assumption, small perturbations (denoted by primes) of the medium density \( \rho(r,t) \), pressure \( p(r,t) \) and velocity \( v(r,t) \). In this context, the laws of conservation of mass and momentum can be written as:

\[ \frac{\partial \rho'}{\partial t} + \nabla.(\rho_0 v') = 0 \] (A.2)

\[ \rho_0 \frac{\partial v'}{\partial t} + \nabla \rho' = 0 \] (A.3)

---

The material in Appendix A is essentially a summary of the corresponding sections in [141] and [19].
where

\[ \nabla = i_x \frac{\partial}{\partial x} + i_y \frac{\partial}{\partial y} + i_z \frac{\partial}{\partial z} \quad (A.4) \]

\( \nabla \) is the partial derivative in Cartesian space with respect to \((i_x, i_y, i_z)\) basis vectors.

Differentiating Eq. A.2 in time and eliminating \( \partial v' / \partial t \) given Eq. A.3 we obtain:

\[ \frac{\partial^2 p'}{\partial t^2} = \nabla^2 p' \quad (A.5) \]

To close the system denoted by Eq. A.2 and Eq. A.3, we add the simplest form that relates pressure to density, provided by barotropic fluids:

\[ p = p(\rho) \quad (A.6) \]

We use the Taylor series around the unperturbed state:

\[ p = p(\rho_0) + \frac{dp}{d\rho}|_{\rho=\rho_0}(\rho - \rho_0) + O((\rho - \rho_0)^2) \quad (A.7) \]

Given that \( p(\rho_0) = \rho_0 \), and ignoring the second order nonlinear term, we obtain:

\[ p' = c^2 p', \quad c^2 = \frac{dp}{d\rho}|_{\rho=\rho_0} \quad (A.8) \]

where \( c \) is the speed of sound.

Plugging Eq. A.8 into Eq. A.5 yields the wave equation for pressure perturbations

\[ \frac{1}{c^2} \frac{\partial^2 p'}{\partial t^2} = \nabla^2 p' \quad (A.9) \]

Likewise, the wave equation for the velocity vector’s perturbations is

\[ \frac{1}{c^2} \frac{\partial^2 \mathbf{v}'}{\partial t^2} = \nabla^2 \mathbf{v}' \quad (A.10) \]
Eq. A.10 shows that the velocity vector satisfies the scalar Eq. A.9. To find the velocity field, the momentum equation (Eq. A.3) states that there exists a scalar function \( \phi \) such that:

\[
\mathbf{v}' = \nabla \phi, \quad \frac{1}{c^2} \frac{\partial^2 \phi'}{\partial t^2} = \nabla^2 \phi \quad (\rho_0 \frac{\partial \phi}{\partial t} = -p')
\]

(A.11)

Hence, the potential is solved for, then the velocity field is found as the gradient of the scalar \( \phi \).

### A.1.1.1 Monochromatic Waves

If the time dependence of the previous wave equation is a harmonic function of \( \omega \), then we can write

\[
\phi(r, t) = \text{Re}(e^{-i\omega t} \psi(r)), \quad i = -1
\]

(A.12)

The real part of the complex-valued \( \psi(r) \) is taken because \( \phi(r, t) \) is real. Substituting Eq. A.12 into Eq. A.11, we observe that the wave equation is satisfied if \( \psi(r) \) is a solution of the Helmholtz equation:

\[
\nabla^2 \psi(r) + k^2 \psi(r) = 0, \quad k = \frac{\omega}{c}
\]

(A.13)

\( k \) (wavenumber) is constant and is real when \( \omega \) is real. For waves propagating in fluids, \( k = \lambda/2\pi \). In other words, \( k \) is the number of waves per \( 2\pi \) units.

For multiple frequencies, the Fast Fourier Transform can be used to move from time to frequency.

### A.1.2 Green’s Function of the Helmholtz Equation

George Green (1793 - 1841) examined the solution of Eq. A.1 within a volume \( V \) that satisfies certain boundary conditions along the boundary surface \( S \). If we consider the case of ordinary differential equations, we can express the linear, inhomogeneous
differential equation as \( Lu = f \), where \( L \) is the ordinary differential operator corresponding to the independent variable \( x \) such that \( f \) is a known function and \( u \) is the desired solution. The problem can be seen as an eigenvalue problem \( Lu = \lambda u \), where \( u \) is an arbitrary constant. Alternatively, Green’s method represents the problem as \( Lg = \delta(x - \xi) \) around an arbitrary point of excitation \( \xi \). Green shows that solution of \( u(x) \) is given by the integral involving Green’s function \( g(x, \xi) \).

A simple example of Green’s function for the wave equation is the 1D string problem. In this problem the Green’s function is given by the equation

\[
\frac{\partial^2 g}{\partial t^2} - c^2 \frac{\partial^2 g}{\partial x^2} = c^2 \delta(x - \xi) \delta(t - \tau) \tag{A.14}
\]

where \(-\infty < x, \xi < \infty\), and \( 0 < t, \tau \). The Laplace transform of Eq. (A.14) assuming zero initial conditions, is

\[
\frac{d^2 G}{dx^2} - \frac{s^2}{c^2} c^2 G = -\delta(x - \xi)e^{-st} \tag{A.15}
\]

where \( G(x, s|\xi, \tau) \) is the Laplace transform of \( g(x, t|\xi, \tau) \). To solve Eq. (A.15), the Fourier transform is taken to obtain the algebraic equation.

\[
\overline{G}(k, s|\xi, \tau) = \frac{e^{-ik\xi-st}}{k^2 + s^2/c^2} \tag{A.16}
\]

To go back to the Green’s function denoted by Eq. (A.14) we use the definition of the Fourier transform, which implies

\[
G(x, s|\xi, \tau) = \frac{e^{-st}}{2\pi} \int_{-\infty}^{\infty} \frac{e^{-ik\xi-st}}{k^2 + s^2/c^2} dk \tag{A.17}
\]

Hence the solution of the heat equation boils down to evaluating the Fourier type integral in Eq. (A.17)
An example function $G$ in free-space for the 3D scalar Helmholtz Equation is defined as follows [142]:

$$G(x, y) = \frac{e^{ik|x-y|}}{4\pi|x-y|}, x, y \in \mathbb{R}^3$$  \hspace{1cm} (A.18)

$G$ is a function only of the distance between $x$ and $y$ and therefore a symmetric function of the two arguments, which satisfies:

$$\nabla^2 G(x, y) + k^2 G(x, y) = -\delta(x - y) \hspace{1cm} (A.19)$$

$\delta(x - y)$ is the Dirac delta distribution defined for arbitrary $f(x)$ as

$$\int_{\mathbb{R}^3} f(x)\delta(x - y)dV(x) = \begin{cases} f(y), & \text{if } y \in \mathbb{R}^3 \\ 0, & \text{otherwise} \end{cases} \hspace{1cm} (A.20)$$

Hence, the Green’s function $G(x, y)$ is a solution to the Helmholtz equation for $x \in \mathbb{R}^3 \setminus y$ and $y \in \mathbb{R}^3 \setminus x$. The equation that the Green’s function satisfies is called the inhomogeneous Helmholtz equation defined as:

$$\nabla^2 \psi(r) + k^2 \psi(r) = -f(r) \hspace{1cm} (A.21)$$

$f(r)$ is the spatial distribution of sources.

Green’s function computes the solution at the boundary of the domain, because the derivation of the corresponding solution of the Helmholtz equation follows from the divergence theorem [143].
A.1.3 Divergence Theorem

The divergence theorem was initially introduced by Gauss (1777 - 1855) to relate volume to surface integral as follows:

\[ \int_{\Omega} (\nabla \cdot u) dV = \int_{S} (n \cdot u) dS \]  
(A.22)

such that \( n \) is an outward unit normal vector to the surface \( S \) of the domain \( \Omega \). Assuming that the integral converges, the theorem is true for both finite and infinite domains. For \( n \)-dimensions, we can write:

\[ \int_{\Omega^n} (\nabla \circ A) dV = \int_{\partial \Omega^n} (n \circ A) dS, \Omega^n \subset \mathbb{R}^n \]  
(A.23)

Where \( A \) is a scalar or vector where \( \circ \) is defined.

A.1.4 Green’s Integral Theorem and Formula

It follows from the divergence theorem and integration by parts that

\[ \int_{\Omega} u \nabla^2 v dV = \int_{\Omega} v \nabla^2 u dV + \int_{S} n.(u \nabla v - v \nabla u) dS \]  
(A.24)

By integrating over the domain boundary \( \Omega \), the solution of Eq. A.1 can be found using the Green’s Formula:

\[ \int_{\Omega} \psi(x) \delta(x - y) dV(x) = \psi(y), y \in \Omega \]  
(A.25)

Plugging Eq. A.19 in Eq. A.25, we can write

\[ \psi(y) = -\int_{\Omega} \psi(x)[\nabla^2 G(x,y) + k^2 G(x,y)] dV(x) \]  
(A.26)
Using Eq. [A.25] and setting $u = \psi$ and $v = G$, we can rewrite Eq. [A.26] as

$$
\psi(y) = - \int_{\Omega} k^2 \psi(x) G(x, y) dV(x) - \int_{\Omega} G(x, y) \nabla_x^2 \psi dV(x)
$$

$$
- \int_{S} n.[\psi(x) \nabla_x G(x, y) - G(x, y) \nabla_x \psi(x)]dS(x)
$$

$$
= - \int_{\Omega} [\nabla_x^2 \psi(x) + k^2 \psi(x)] G(x, y) dV(x)
$$

$$
- \int_{S} n.[\psi(x) \nabla_x G(x, y) - G(x, y) \nabla_x \psi(x)]dS(x)
$$

(A.27)

Assuming that $\psi(x)$ satisfies Eq. [A.21] (the inhomogeneous Helmholtz equation), then $\psi(y)$ can be written as:

$$
\psi(y) = \int_{\Omega} f(x) G(x, y) dV(x) - \int_{S} n.[\psi(x) \nabla_x G(x, y) - G(x, y) \nabla_x \psi(x)]dS(x) \tag{A.28}
$$

In the case of an infinite domain, the solution is the impulse response

$$
\psi(y) = \int_{\Omega} f(x) G(x, y) dV(x) \tag{A.29}
$$

If $\psi$ satisfies Eq. [A.21] and setting $f = 0$. Then we can get the solution for $\psi$ from the boundary values using Eq. [A.28] and $\psi$ becomes the Helmholtz Integral Equation

$$
\psi(y) = \int_{S} [G(x, y) \frac{\partial \psi(x)}{\partial n(x)} - \psi(x) \frac{\partial G(x, y)}{\partial n(x)}]dS(x), y \in \Omega, \tag{A.30}
$$

$\delta/\delta n(x) = n \cdot \nabla_x$ and $n$ is directed outside the domain.

If $n$ is defined as a normal vector outside the scattering body. The solution for the scattering problems becomes

$$
\psi(y) = \int_{S} [\psi(x) \frac{\partial G(x, y)}{\partial (n)} - G(x, y) \frac{\partial \psi(x)}{\partial n(x)}]dS(x) \tag{A.31}
$$

$n$ is directed inside the domain.
A.2 Fast Multipole for Oscillatory Kernels

This section will demonstrate several use cases of multipole expansion and translation to evaluate boundary integrals.

A.2.1 Room Acoustics for Monopole Source

Assume a single sound source that has an intensity of $Q_0$ is placed at the center of a room. In the absence of walls, the acoustics field generated is:

$$\psi_0 = \frac{Q_0}{4\pi |r - r_0|} e^{ik|r - r_0|}$$

This field satisfies the Sommerfeld radiation conditions. The presence of walls is modeled by an infinite number of sources reflecting signals of certain frequency and radiation. The duration of sound reflection is denoted by $t_{max}$. Only images with $|r_q - r| \leq ct_{max}$ are considered. For an arbitrary point, $r$ is the radius vector. For images sources located inside $\omega_b$, the field can be written as:

$$\psi(r) = \sum_{r_q < b} \psi_q(r), \quad \psi_q(r) = \frac{Q_q}{4\pi |r - r_q|} e^{ik|r - r_q|}, \quad r_q = |r_q|, r = |r|$$

$Q_q$ is the intensity of the $q$th source. Since the number of source points $N$ can be very large, the evaluation of the summation over $M$ target points can have a quadratic complexity of $O(NM)$. The multipole expansion can drastically break it down to a linear complexity of $O(N + M)$.

A.2.2 Solution

The summation in Eq. A.33 can be split into two terms:

$$\psi(r) = \psi_s(r) + \psi_r(r)$$
where
\[ \psi_s(r) = \sum_{r_q < a} \psi_q(r), \quad \psi_r(r) = \sum_{a < r_q < b} \psi_q(r) \] (A.35)

Such that \( \psi_s \) is the singular part, and \( \psi_r \) is the regular part. The singular part includes the original source and the sources located within a sphere \( \omega_a \). We will consider the regular part \( \psi_r \), which is bounded between \( \omega_a \) and \( \omega_b \). It can be expanded to a series of spherical harmonics:
\[
\psi_q(r) = i k \sum_{n=0}^{\infty} \sum_{m=-n}^{n} S_n^{-m}(r_j) R_n^m(r), r \leq r_q
\] (A.36)

The regular part is written as:
\[
\psi_r(r) = i k \sum_{n=0}^{\infty} \sum_{m=-n}^{n} C_n^m R_n^m(r), \quad C_n^m = \sum_{r_q < R_{max}} Q_q S_n^{-m}(r_q)
\] (A.37)

\( C_n^m \) are coefficients that be precomputed for each source location. The series is typically truncated at \( p \ll N \), making the evaluation of summation in Eq. [A.37] of order \( O(p^2) \) which is significantly smaller than \( N \) sources for very large inputs.

### A.2.3 Scattering from a Sphere

Let a sphere centered at \( r = r_0 \), with radius \( a \) be exposed to an incident field \( \psi_{in} \) that is governed by the Helmholtz equation around the sphere. The total potential outside the sphere can be calculated as:
\[
\psi(r) = \psi_{in}(r) + \psi_{scat}(r)
\] (A.38)

\( \psi_{scat} \) is the potential of the scattered field at \( |r - r_0| > a \). Given \( \psi_{in}(r) \), we need to calculate \( \psi_{scat}(r) \) or \( \psi(r) \). On the surface of the sphere, the following boundary
condition is satisfied:

$$|r - r_0| = a : \frac{\partial \psi(r)}{\partial n} + i\sigma \psi(r) = 0,$$

(A.39)

$\sigma$ is the complex admittance, whereas $\partial/\partial n$ is the derivative at norm pointing outside of the surface.

### A.2.4 Solution

$\psi_{in}(r)$ is nonsingular function with a radius $b > a$ that is concentric with the scatterer. It can be expanded about $r = r_0$ as follows:

$$\psi_{in}(r) = \sum_{n=0}^{\infty} \sum_{m=-n}^{n} E_m^n R_n^m (r - r_0)$$

(A.40)

The coefficients $E_m^n$ are based on the incident field. For a plane wave with intensity $\psi_{in}(r_0) = Q$, we have:

$$\phi_{in}(r) = Q e^{ik(r-r_0)} = 4\pi Q \sum_{n=0}^{\infty} \sum_{m=-n}^{n} i^n Y_n^{-m}(\theta_k, \varphi_k) R_n^m (r - r_0)$$

$$E_m^n = 4\pi Q i^n Y_n^{-m}(\phi_k, \varphi_k), \quad n = 0, 1, ..., m = -n, ..., n.$$  

(A.41)

The direction of the wave is characterized by the polar angles $\theta_k$ and $\varphi_k$. Accordingly, the incident wave is generated by $Q$ that is located at $r = r_s$. It can be expanded as follows:

$$\psi_{in}(r) = QG(r - r_s) = Qik \sum_{n=0}^{\infty} \sum_{m=-n}^{n} S_n^{-m}(r_s - r_0) R_n^m (r - r_0),$$

$$E_n^m = Qik S_n^{-m}(r_s - r_0), \quad n = 0, 1, ..., m = -n, ..., n.$$  

(A.42)

The scattered field, which is a radiating function, can be constructed as the sum of singular spherical bases:
\[ \psi_{\text{scat}}(r) = \sum_{n=0}^{\infty} \sum_{m=-n}^{n} A_n^m S_n^m (r - r_0) \] (A.43)

\( A_n^m \) can be determined from the boundary conditions. The total potential can be written as:

\[ \psi(r) = \sum_{n=0}^{\infty} \sum_{m=-n}^{n} \left[ E_n^m j_n(k|r - r_0|) + A_n^m h_n(k|r - r_0|) \right] Y_n^m(s), \]

\[ s = \frac{r - r_0}{|r - r_0|} \] (A.44)

To fulfill the boundary conditions in Eq. [A.39] at \(|r - r_0| = a\), we have:

\[ \sum_{n=0}^{\infty} \sum_{m=-n}^{n} k [E_n^m j_n'(ka) + A_n^m h_n'(ka)] + \sigma [E_n^m j_n(ka) + A_n^m h_n(ka)] Y_n^m(s) = 0 \] (A.45)

where \( s = (r - r_0)/a \). Each term in this summation should be zero, due to completeness and orthogonality of the spherical harmonics, which means \( A_n^m \) should be:

\[ A_n^m = -\frac{j_n'(ka) + (\sigma/k) j_n(ka)}{h_n'(ka) + (\sigma/k) h_n(ka)} E_n^m, \quad n = 0, 1..., m = -n, ..., n \] (A.46)

For sound-hard and sound-soft cases, we have:

\[ \sigma = 0 : A_n^m = -\frac{j_n'(ka)}{h_n'(ka)} E_n^m, \sigma = \infty : A_n^m = -\frac{j_n(ka)}{h_n(ka)} E_n^m, \]

\[ n = 0, 1..., m = -n, ..., n \] (A.47)

These cases are realized for \( \sigma \ll k \) and \( \sigma \gg k \). For the region outside the sphere surrounding the scatterer, the solution can be found using Eq. [A.43]. In general, the expansion terms have a linear relationship with the incident field’s coefficients. This is because of the linearity of the Helmholtz equations and boundary conditions. Hence,
we can write:

\[ A_n^m = \sum_{n'=0}^{\infty} \sum_{m'=-n}^{n'} T_{nn'}^{mm'} E_{n'n'}^m, n = 0, 1, \ldots, m = -n, \ldots, n \]  

(A.48)

\( T_{nn'}^{mm'} \) are elements of the T-matrix. Apparently, this matrix is diagonal:

\[ T_{nn'}^{mm'} = -\frac{j_n'(ka) + (\sigma/k)j_n(ka)}{h_n'(ka) + (\sigma/k)h_n(ka)} \delta_{nn'} \delta_{mm'}, n, n' = 0, 1, \ldots, m = -n, \ldots, n, m' = -n, \ldots, n' \]  

(A.49)

A.2.5 Surface function

The potential and its derivative should be determined on the surface of the body. The expression for that can utilize the Wronskian for spherical Bessel functions:

\[ W\{j_n(ka), h_n(ka)\} = j_n(ka)h'_n(ka) - j'_n(ka)h_n(ka) = i(ka)^{-2} \]  

(A.50)

Substituting Eq. A.46 into Eq. A.44 at \(|r - r_0| = a\), we have:

\[ \psi|_s = \sum_{n=0}^{\infty} \sum_{m=-n}^{n} \left[ j_n(ka) - \frac{j'_n(ka) + (\sigma/k)j_n(ka)}{h'_n(ka) + (\sigma/k)h_n(ka)} h_n(ka) \right] E_n^m Y_n^m(s) = \frac{i}{(ka)^2} \sum_{n=0}^{\infty} \frac{1}{h'_n(ka) + (\sigma/k)h_n(ka)} \sum_{m=-n}^{n} E_n^m Y_n^m(s) \]  

(A.51)

The normal derivative can be found using boundary conditions in Eq. A.39:

\[ \frac{\partial \psi}{\partial n}|_s = -i\sigma \psi|_s = \frac{\sigma}{(ka)^2} \sum_{n=0}^{\infty} \frac{1}{h'_n(ka) + (\sigma/k)h_n(ka)} \sum_{m=-n}^{n} E_n^m Y_n^m(s) \]  

(A.52)
For the cases pertaining to the Neumann and Dirichlet boundary conditions, we have:

\[
\sigma = 0: \psi|_S = \frac{i}{(ka)^2} \sum_{n=0}^{\infty} \frac{1}{h_n'(ka)} \sum_{m=-n}^{n} E_n^m Y_n^m(s), \frac{\partial \psi}{\partial n}|_S = 0, \\
\sigma = \infty: \psi|_S = 0, \frac{\partial \psi}{\partial n}|_S = \frac{i}{(ka)^2} \sum_{n=0}^{\infty} \frac{1}{h_n'(ka)} \sum_{m=-n}^{n} E_n^m Y_n^m(s),
\]

(A.53)

For typical incident fields of plane waves. Using the addition theorem of spherical harmonics \(E_n^m\) in Eq. (A.41), we have:

\[\sum_{m=-n}^{n} E_n^m Y_n^m(s) = Q(2n + 1)i^n P_n(\cos(\theta)), \quad \cos(\theta) = \frac{k_s}{k}\]

(A.54)

Accordingly, we obtain:

\[
\psi|_S = \frac{iQ}{(ka)^2} \sum_{n=0}^{\infty} \frac{(2n + 1)i^n P_n(\cos \theta)}{h_n'(ka) + (\sigma/k)h_n(ka)} \\
\frac{\partial \psi}{\partial n}|_S = \frac{\sigma Q}{(ka)^2} \sum_{n=0}^{\infty} \frac{(2n + 1)i^n P_n(\cos \theta)}{h_n'(ka) + (\sigma/k)h_n(ka)}
\]

(A.55)

For monopole and \(E_n^m\) sources, we can still use the addition theorem to get:

\[
\sum_{m=-n}^{n} E_n^m Y_n^m(s) = \frac{ikQ}{4\pi} h_n(kd)(2n + 1)P_n(\cos \theta), \\
\cos \theta = \frac{(r_s - r_0)s}{d}, \quad d = |r_s - r_0|
\]

(A.56)

Therefore, the surface potential and its derivative are:

\[
\psi|_S = -\frac{Q}{4\pi ka^2} \sum_{n=0}^{\infty} \frac{(2n + 1)h_n(kd)P_n(\cos \theta)}{h_n'(ka) + (\sigma/k)h_n(ka)} \\
\frac{\partial \psi}{\partial n}|_S = -\frac{\sigma Q}{4\pi ka^2} \sum_{n=0}^{\infty} \frac{(2n + 1)h_n(kd)P_n(\cos \theta)}{h_n'(ka) + (\sigma/k)h_n(ka)},
\]

(A.57)
A.3 Duffy Transformation for $1/R$ Self-singularity

Duffy’s method is a technique that employs a change of integration variable, where each transformation scheme results in different forms of Jacobian that cancel the singularity in the kernel [106]. This method is mainly applied to $1/R$ self-singularity cases. Figure A.1 gives a bird’s eye view of the variable transformation undertaken by Duffy’s method, by which the singularity is canceled through multi-transformations, and then the integral can be evaluated using standard numerical techniques such as Gaussian integration rule. Following summarizes the multi-transformations done by the Duffy’s method:

1. Figure A.1(a) shows the integration to evaluate over curvilinear source patch, where the field point lies inside the patch.

2. Figure A.1(b) presents the integration mapping to the reference triangle.

3. The reference triangle is partitioned into three sub-triangles that share the singular point.
4. The integration on the reference triangle can be obtained by the summation of the integrations on the three sub-triangles: T1, T2 and T3.

5. Figure A.1(c) illustrates the mapping into a \((u, v)\) space of Figure A.1(b) sub-triangle, which in turn maps the singular point to the origin point.

6. Figure A.1(d) exhibits the \((u, v)\) space mapping into a square \((\epsilon, t)\) space, by which it elapses the singularity at one point to one edge of the unitary patch.

**Adjacent Singularity:** The oscillatory behavior that inhibits convergence of the Greens function occurs when two points are close to each other (i.e., adjacent points) but each one resides in a different patch. For a particular case, the so-called \textit{P-refinement} is exploited to tackle such oscillatory behavior. \textit{P-refinement} is a well-known technique that simply adds more integration points within the underlying triangular elements, through which we increase the elements’ polynomial degrees \[144\]. This allows convergence of the Green’s function at a faster rate.

### A.4 Data-level Parallelism

Contemporary processing hardware is equipped with Instruction Set Architecture (ISA) that supports Single Instruction, Multiple Data (SIMD) operations on many vectorized data items. For instance, Intel Xeon Skylake architecture implements two 512-bit Vector Processing Units (VPUs) per core, by which a single arithmetic instruction can be performed on a large subset of independent, distinct data items. In the context of our highly optimized FMM Helmholtz kernels, we undertake two different vectorization approaches: 1) we handwrite the vector code for the key kernels using AVX-512 intrinsics, and 2) we further optimize and fine-tune the kernels to aid the Intel compiler to automatically generate efficient vector codes.

**Validating the Compiler’s Loop Choice:** Treating singularity in the innermost loop of the 3D Helmholtz kernel depicted by Listing A.1 through iterating over
the high degree Gauss quadrature points within the Particle-to-Particle (P2P) and Source-to-Target (S2T) routines of FMM involves, in principle, complicated nested for loops that involve many conditional statements. When we analyze the Intel compiler’s report of vectorization generated by the Intel Advisor assistance tool, we find out that the compiler tends to vectorize the innermost loops by default. However, this does not result in an additional advantage from vectorization, since the spatial and temporal locality of references are well-preserved when the outer loops are vectorized and strided \cite{145, 97, 96}. One way to vectorize such kernels is through populating the scalar loop’s data using several vector broadcast instructions, as opposed to fetching a unit stride from a cache line via \texttt{vmov} instruction. Thus, using \texttt{vbroadcast} instructions imposes lower latency and reciprocal throughput \cite{146}. Furthermore, lowest latency and highest spatial and temporal locality of reference are achieved with outer loop vectorization, especially in balanced (equal-sized) chunks inside the nested for loops. In order to construct such vector code, either the compiler needs to be instructed via \texttt{#pragma simd}, or more aggressively, writing the vector code manually through utilizing intrinsics. Indeed, both approaches require certain loop optimization techniques (e.g., transformation and unrolling) \cite{131, 147}. Relying on the compiler to auto-vectorize the code, whenever it is possible, is definitely the right approach and is highly recommended to guarantee portability and resilience. Nonetheless, in many cases the compiler could fail to extract the correct or efficient vector code due to assumed data dependencies imposed by the data structures.

Writing SIMDizable Code: Since some arithmetics are known to be expensive in terms of latency, which could squander many CPU cycles, modern compilers are designed to avoid such arithmetic as much as possible. For instance, square root and division operations are very often replaced by their reciprocal counterpart, whenever the code is compiled with certain optimization flags. Nevertheless, sometimes the compiler’s auto-generated vector codes is suboptimal, which primarily de-
for ( ; i<ni; ++i) {
    vi_r = real(Bi.SRC); vi_i = imag(Bi.SRC);
    for (j=0; j<nj; ++j) {
        dX = xi-xj;
        R2 = norm(dX);
        if(Bi.PATCH!=Bj.PATCH && R2!=0) {
            real_t R=sqrt(R2);
            if(R<=near_patch_distance) {
                for (k=0; k<gauss_quad_points; ++k) {
                    near patch singularity treatment
                }
            } else {
                vj_r = real(Bj.SRC); vj_i = imag(Bj.SRC);
                src2_r = vi_r*vj_r-vi_i*vj_i;
                src2_i = vi_r*vj_i+vi_i*vj_r;
                invR = 1.0/sqrt(R);
                eikr = 1.0/exp(wave_i*R);
                eikr *= invR;
                eikr_r = cos(wave_r*R)*eikr;
                eikr_i = sin(wave_r*R)*eikr;
                pot_r += src2_r*eikr_r-src2_i*eikr_i;
                pot_i += src2_r*eikr_i+src2_i*eikr_r;
            }
        }
    Bi.TRG += complex(pot_r, pot_i);
}
pends on the scalar source code. Hence, writing intrinsics seems to be inevitable in such cases. For example, consider $1/\sqrt{R}$ of line 19 in Listing A.1 the corresponding assembly code of the Intel compiler comprises of 2 `vmovups`, 3 `vmulps`, 1 `vrsqrt14ps`, 1 `vfmsub213ps`. This is fairly a reasonable approach that the compiler adopts to build portable, efficient vector code. However, one can write a more efficient code, which uses only `vmovups` and `vrsqrt14ps`, via an explicit call to `_mm512_rsqrt14_ps(r)` intrinsic. On the other hand, a smarter way can achieve both (i.e., efficient vector code generated by the compiler while avoiding writing explicit SIMD code) through breaking down the $eikr = (\sqrt{R} \times e^{ikR})^{-1}$ operation to lines 19, 20, and 21 of Listing A.1. Thereby, the compiler automatically understands this transformation, and would extract the most cost-effective, well-optimized vector code.

**Optimizing Memory Access:** It is well-understood that memory bandwidth is a critical obstacle that limits the performance of modern HPC architectures. As a consequence, one must carefully inspect how cache lines or memory words are fetched into the vector units, especially since most modern x86 architectures are mounted on dual-socket NUMA nodes, in which data might physically reside on different address spaces [131]. Therefore, we develop the FMM core kernels to allocate and reference the particles and tree cells data structures in the form of Array-of-Structs (AoS). In addition, AoS enhances the locality of references for interacting particles after they are sorted and indexed based on their Morton order. Cells maintain both indexes and counts of the encapsulated set of particles (see Listing A.2). This is somehow a simple and compact version of a hash map associative array abstract data type to map keys to values (i.e., index and count). Hence, strided memory access via AVX-512 intrinsics can be utilized to efficiently reference the SRC data structure of line 2 of Listing A.1. However, having to carry out such low-level manipulation with intrinsics might not be attainable except with a great deal of coding effort, since it requires manipulation of
memory addresses using shuffle, permute, gather and scatter instructions, which results in non-portable, error-prone, compiler-specific code. Thus, we code the low-level kernels in such a way that the compiler can extract the most optimal vector code without the need to explicitly use handwritten intrinsics.

### A.5 Convergence Effects of the Singularity Treatments

Figure A.2 shows the convergence behavior of the solver where we calculate the far scattered field interactions by a sphere of radius 1\(m\) using different singularity treatment modes. The Y-axis shows the number of iterations required for GMRES convergence, whereas the X-axis represents the GMRES relative residual norm. The self-singularity treatment creates higher order Gauss quadrature points around the sources and targets falling exactly on the diagonal, or having \([R < \epsilon]\) from geometrical perspective. Such points are ignored in typical FMM implementations. Near-singularity scheme treats points that fall within \([R < \text{near}]\) radius (\text{near} is a code parameter). It can be clearly depicted in Figure A.2 that when singularity is ignored, we exhibit a lower convergence rate that is driven and dominated by the singularity of the Green’s function. However, when treating only the true singularity, a remarkably improved convergence rate is observed. Finally, when considering the self- and near-singularity treatment schemes, the relative 2-norm residual accuracy reached 1.0e-4
within just less than 50 iterations. This accuracy corresponds to 1.0e-3 to 1.0e-4 error with respect to the analytical solution, and it cannot be further improved even with less relative tolerance of the underlying iterative solution. Hence, our GMRES solver is configured to exit at 1.0e-4 relative 2-norm residual accuracy.

Figure A.2: Convergence effects of self- and near-singularity treatments.
Relevant publications, which are co-authored by me with first or second author contributions, are listed below in chronological order of their acceptance/submission:

- Mustafa Abduljabbar and Rio Yokota “N-body Methods”, *Published as a Book Chapter 10 in High Performance Parallelism Pearls (HPPP’14)*.
- Mani Zandifar, Mustafa Abduljabbar, Alireza Majidi, Nancy Amato, and Lawrence Rauchwerger “Composing Algorithmic Skeletons to Express High-Performance Scientific Applications”, *Published in the Proceedings of the International Conference of Supercomputing (ICS’15)*.