Method of Local Corrections Solver for Manycore Architectures

by

Brian Van Straalen

A dissertation submitted in partial satisfaction of the

requirements for the degree of

Doctor of Philosophy

in

Computer Science

and the Designated Emphasis

in

Computational and Data Science and Engineering

in the

Graduate Division

of the

University of California, Berkeley

Summer 2018

Method of Local Corrections Solver for Manycore Architectures

Abstract

Method of Local Corrections Solver for Manycore Architectures

by

Brian Van Straalen

Doctor of Philosophy in Computer Science

University of California, Berkeley

Professor Phillip Colella, Chair

Microprocessor designs are now changing to reflect the ending of Dennard Scaling. This leads to a reconsideration of design tradeoffs for designing discretization methods for PDEs based on simplified performance models like Roofline.

In this work we carry out an end-to-end analysis and implementation study on a Cray XC40 with Intel®Xeon^{TM} E5-2698 v3 processors for the Method of Local Corrections (MLC). MLC is a non-iterative method for solving Poisson's Equation on locally rectangular meshes. The Roofline model predicts that MLC should have faster time to solution than traditional iterative methods such as Geometric Multigrid. We find that Roofline is a useful guide for performance engineering and obtain performance within a factor of 3 the Roofline performance upper bound. We determine that the algorithm is limited by identified architectural features that are not captured in the Roofline model, are quantifiable, and can be addressed in future implementations.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Hardware Trends

Microprocessor design in the past decade have been shaped by two dominant physical effects: Our continued ability to reduce feature size as discussed in the *ITRS: The international technology roadmap for semiconductors* [43] and the end of *Dennard Scaling* [30, 32]. Dennard (1974) observed that voltage and current should be proportional to the linear dimensions of a transistor. Thus, as transistors shrank, so did necessary voltage and current; power is proportional to the area of the transistor. So smaller features were matched by lower power trends and the product of the two, power density, remained stable. Dennard scaling ignored the *leakage current* and *threshold voltage*, which establish a baseline of power per transistor. Now, as transistors get smaller, power density increases. Thus we have the *power wall* that has limited practical processor frequency to around 4 GHz since 2006. This kicked off our current era of using more processing *cores* that distributed the thermal load, and fewer transistors dedicated to mitigating memory latency, which have higher power density. These changes were well documented in the National Academy Study *The Future of Computing Performance*[37]. The general term for this architectural shift is *manycore*.

To capture this complicated architectural landscape it is helpful to reference more simplified computing abstractions and performance models. The Roofline model [67, 66] describes the upper limits of attainable performance in terms of bandwidth and arithmetic operations. The name *Roofline* is evocative of a ceiling on attainable performance for algorithms that can be expressed as a streaming computation. For a given ratio of arithmetic operations that need to be performed and bytes of memory that need to be accessed, and a data working set size, the Roofline predicts the maximum performance that can be obtained from an optimal algorithm implementation. The trend can be seen in two successive generations of Intel®microprocessor-based supercomputer platforms from the Department of Energy National Energy Research Supercomputer Center in figure 1.1. The vertical axis is in units of GFLOP/s. The horizontal axis is the *arithmetic intensity*, specified as the number of FLOPs performed per byte of memory accessed. The slanted bandwidth lines (L1, L2, L3, DRAM) all have a slope of 1/8 (8 being the number of bytes in a double-precision floating-point data value). Each line represents the peak GFLOP/s acheivable for a streaming compute kernel that is bandwidth-bound whose working set fits with the named level of the cache hieararchy. *Bandwidth-bound* means that the processor is capable of executing floating-point operations faster, but the memory subsystem is not capable of delivering bytes to and from the functional units to keep them fully active. The processor will execute some number of `NOP` instructions while waiting for outstanding load/store instructions to complete. The horizontal lines represent the maximum GFLOP/s achievable set by the ability of the processor to execute floating-point instructions. Multiply/Add shows the peak performance for motifs that utilitze a Fused-Multiply-Add instruction. Add stands in for all other instructions that are the eqivalent of a single floating-point operation. Computional motifs that are constrained by the horizontal red lines are said to be *Compute-bound*. The Roofline defines a peak performance attainable. Many architectural complexities are elided here, but all other features will only lower performance relative to this performance envelope.

Several classic HPC motifs are plotted in these Roofline plots in magenta: Algebraic Multigrid/Sparse Matrix-Vector Multiply (AMG/SpMV), Geometric Multigrid (GMG), Geometric Multigrid with the more computationally dense GMG scheme where several core computational phases have been manually fused and tuned (GMG (fused)), Fast Fourier Transform, both small (FFT(1K)), and large (FFT (8K)) computed as $5N\log_2(N)/4 \cdot 8 \cdot N$, and Dense Double-Precision Matrix-Matrix Multiply (DGEMM, $m{\cdot}n = 128$). In the Roofline Model a computational motif is entirely characterized by its arithmetic intensity, to these are shown as vertical lines. For example, we can make a prediction for a Sparse Matrix-Vector multiply kernel on the Edison platform where the working set fits entirely in the L2 level of cache. We follow the AMG/SpMV line that defines the arithmetic intensity of this algorithm (AI=0.125) upwards until we encounter the rate-limiting ceiling on performance, which shows that these algorithm motifs are bandwidth-bound by the available L2 bandwidth and the expected upper bound performance can be read from the intersection of with the L2 bandwidth limit (in this case roughly 63 GFLOPS).

The Roofline model highlights the need to place less emphasis on algorithms that emphasize minimum floating-point and towards algorithms that minimize data movement. This concept has been well studied for dense and sparse communication-avoiding linear algebra [29, 4, 28] and extended to what Demmel and Ballard refers to computations that "smell like" triply-nested loops [11, 10]. For algorithms that have a SpMV-kernel at their core you can see from the Roofline model that communication-avoiding should be the entire focus.

We will investigate the utility of a Roofline performance model in guiding the development of a new fast Poisson solver algorithm.

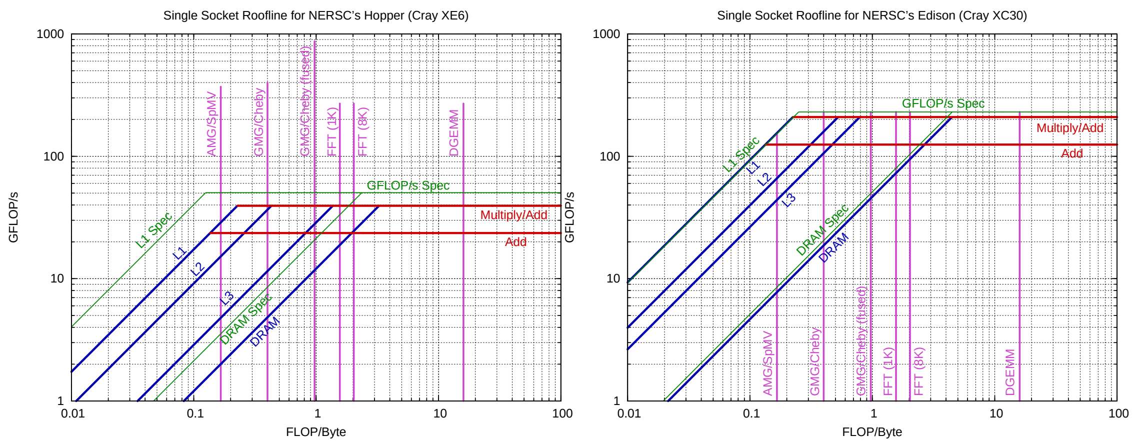


Figure 1.1: Single Socket Roofline plots for NERSC *Hopper* (Cray XE6) and *Edison* (Cray XC30) platforms. Generating using the *Empirical Roofline Toolkit* [50]. The theoretical peak performance from hardware specifications is given in green.

1.2 Poisson's Equation

We are seeking to solve Poisson's Equation in free space

$$\Delta(\Phi) = f \text{ where } supp(f) = D \text{ bounded domain in } \mathbb{R}^3 \tag{1.1}$$

$$\Phi, f : \mathbb{R}^3 \to \mathbb{R}$$

$$\Phi(\boldsymbol{x}) = \frac{Q}{4\pi||\boldsymbol{x}||} + o\left(\frac{1}{||\boldsymbol{x}||}\right) \text{ as } ||\boldsymbol{x}|| \to \infty \tag{1.2}$$

$$Q = \int_D f\,dx$$

which can be expressed as convolution with the Green's function

$$\Phi(\boldsymbol{x}) = \int_D G(\boldsymbol{x} - \boldsymbol{y}) f(\boldsymbol{y}) d\boldsymbol{y} \equiv (G * f)(\boldsymbol{x}) \ , \ G(\boldsymbol{x}) = \frac{1}{4\pi||\boldsymbol{x}||}. \tag{1.3}$$

We seek solutions to these forms of equations on the emerging manycore computing architectures.

A wide range of relevant scientific computing applications require the solution to Poisson's equation including gravity in cosmological and astrophysical models, electrostatic potential in kinetic plasma physics and Coulomb potentials in molecular dynamics and projection methods for solving viscous incompressible flows [21, 22, 51, 42].

1.3 Fast Solvers for Poisson's Equation

A simple and correct method for solving (1.3) would be to discretize space and compute the integral with numerical quadrature. While correct this results in an $O(N^2)$ algorithm where N is the number of grid points. Expressing the discretization of equation (1.3) by finite-difference methods and constructing a linear system of equations to solve also works and Krylov-based iterative methods like preconditioned conjugate gradient can solve these well-conditioned systems of equations in $O(N^k)$ for k only lightly larger than 1. Unfortunately even a small increment in k above 1 leads to a large increment in cost when N becomes large.

Solutions to (1.3) have the property that Φ is a smooth function away from the support of the charge D. Differentiating (1.3) we see that

$$\nabla^w \Phi(\boldsymbol{x}) = O\left(\frac{1}{l^{||\boldsymbol{w}||_1+1}}\right) \tag{1.4}$$

where $\boldsymbol{w} = [w_1, w_2, w_3]$, w_i are non-negative integers, $||\boldsymbol{w}||_1 = w_1 + w_2 + w_3$, and $l =$ distance between $\boldsymbol{x}$ and the support of f. The field induced by distant charges has derivatives that decay like an inverse power law. Thus it should be possible to represent the the nonlocal dependence of the field on the charge using less computational effort. This observation underlies all fast Poisson solvers, and several algorithms have been developed that exploit this property.

For a uniform rectangular grid discretization of Φ and f, the Fast Fourier Transform (FFT) can be used to solve Poisson's equation on a periodic domain, combined with one of a number of techniques for correcting for the infinite-domain boundary conditions. The core FFT solver takes $5Nlog_2(N)$ floating-point operations. Distributed Fast Fourier Transform Techniques [2, 5, 15] are efficient and fast.

Geometric Multigrid [63, 18] is an iterative method for solving grid-based discretizations of Poisson's equation. It uses local $O(N)$ relaxation techniques to reduce high-wave-number components of the error, and the effect of the smooth non-local coupling is efficiently computed by solving the equations on a coarsened grid, with iteration between the coarse and fine representations to obtain the solution. The approach is applied recursively in refinement level. This leads to an $O(N)$ method, where the constant depends on the specific discretization of the Laplacian, but is at least 10x larger than the corresponding constant of 15 in the FFT solver. However, geometric multigrid can be applied to a much broader range of spatial discretizations, such as mapped grids and nested locally refined rectangular grids. Multigrid has been shown to be scalable to O(100K) cores on homogeneous supercomputers with $O(10^9)$ unknowns[64].

The Fast Multipole Method (FMM)[39, 38, 33] and related tree-based fast Poisson solvers[12] are numerical techniques that were developed to speed up the calculation of long-range forces in the n-body problem. FMM does this by expanding the system Green's function using a multipole expansion. Recent implementations have shown good scaling on modern heterogeneous architectures [48]. The Generalized Fast Multipole Method [68] does exploit a uniform grid sampling and is not limited to problems that have an analytic Green's function but has several quadrature integrations at every step in the algorithm and has increasing computational complexity *and* communication for higher degrees of accuracy.

In this work we will be investigating **The Method of Local Corrections** (MLC) [46, 53] defined on structured hierarchically adaptive structured grids, commonly referred to as *Structured Adaptive Mesh Refinement* (AMR) data layouts [3, 31, 60, 8, 24, 23]. MLC breaks the integral form of equation (1.3) into near-field operations that are computed using direct convolutions on small structured patches to neighbors, and a far-field contribution that is solved on a coarser grid refinement. The coarser solve is handled with a similar decomposition into near and far contributions on the next coarser grid.

MLC proceeds in three steps: **(i)** a loop over the fine disjoint patches and the computation of local potentials induced by the charge restricted to those patches on sufficiently large extensions of their support (*downward pass*); **(ii)** a global coarse-grid Poisson solve with a right hand side computed by applying the coarse-grid Laplacian to the local potentials of step (i) *bottom solve*; and **(iii)** a correction of the local solutions computed in step (i) on the boundaries of the fine disjoint patches based on interpolating the global coarse solution from which the contributions from the local solutions have been subtracted (*upward pass*). These boundary conditions are propagated into the interior of the patches by performing Dirichlet solves on each patch. This method is applied recursively to progressively coarser grid refinements.

Structurally MLC is most closely related to Multigrid as it is deployed in AMR algorithms [64]. There is a downward pass where a modified source term is generated, a bottom solve, and an upward pass where corrections to the fine grid solution from the coarse-grid solutions are interpolated. Parallel implementations are based on domain decomposition, and the gridding structure is compatible with more general applications that use AMR grids and require the solution to Poisson's equation as a substep. MLC, unlike Multigrid, is not iterative, and has as a compute kernel small batched FFTs to compute discrete convolutions, rather than stencil operations.

Fast Multipole Methods are also similar to MLC in that they both directly utilize the convolution form of Poisson's Equation, and have a natural hierarchy of scales that leads to a recursive traversal through the data structures. However, mapping the classic spherical multipole basis onto AMR hierarchies adds many complexities of interpolation and quadrature.

1.4 Thesis

At the core of MLC is the opportunity to trade-off more floating-point operations for less data movement with the end result of faster time to solution with the coming hardware trends. The near-field convolutions can be performed using the Fast-Fourier Transform using a variant of the Convolution Theorem and identities for turning linear convolutions into circular convolutions. These convolution operations will be shown to dominate the computation cost of the algorithm, and they can have highly efficient implementations on modern manycore architectures. We want to assess to what extent we are capable of implementing MLC up to its potential of a Roofline model. A look at figure 1.2 shows that for an algorithm composed of collections of FFTs of roughly 2M elements should execute within the floating-point dominated regime of a contemporary processor.

In this thesis we will study the performance potential of MLC theoretically in Chapter 4, then implement and analyze several progressively improved implemenatations in Chapter 5 using a current HPC platform Cori at the NERSC supercomputer center. Cori has compute nodes with Two 2.3 GHz 16-core Haswell processors per node. Each core has its own L1 and L2 caches, with 64 KB (32 KB instruction cache, 32 KB data) and 256 KB, respectively; there is also a 40-MB shared L3 cache per socket. Nodes are connected for distributed computing using a Cray Aries high speed "dragonfly" topology interconnect. The target Roofline model for this work is given in figure 1.2.

Outline for Thesis

First we will define our numerical scheme for representing the discrete problem definition from the continuous operator, a description of Hockney's discrete convolution algorithm, introduce domain decomposition, and then the Method of Local Corrections. Second a performance model is given for the overall MLC algorithm and its key discrete convolution kernel. Third we present descriptions of four algorithm variants, their measured performance, and further performance analysis and motivations for each implementation variant. Finally, a deep analysis of the current best algorithm implemented and future directions.

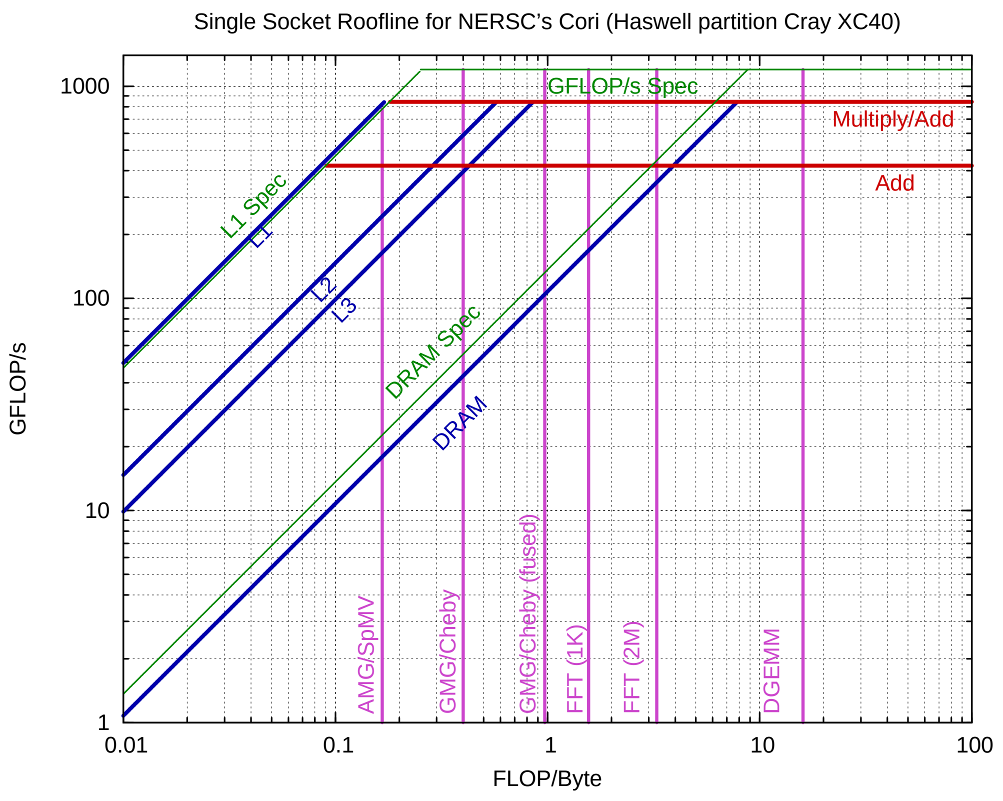


Figure 1.2: Single Socket Roofline plots for NERSC *Cori* (Cray XC40 Haswell nodes). Generated using the *Empirical Roofline Toolkit* [50]. The theoretical peak performance from hardware specifications is given in green.

Chapter 2

Algorithm

2.1 Discretization

Domain decomposition and Adaptive Structured Grids

We denote by $D^h, \Omega^h \dots \subset \mathbb{Z}^3$ grids with grid spacing h of discrete points in physical space: $\{\boldsymbol{g}h : \boldsymbol{g} \in D^h\}$. Arrays of values defined over such sets will approximate functions on subsets of $\mathbb{R}^3$, i.e. if $\psi = \psi(\boldsymbol{x})$ is a function on $D \subset \mathbb{R}^3$, then $\psi^h[\boldsymbol{g}] \approx \psi(\boldsymbol{g}h)$. We denote operators on arrays over grids of mesh spacing h by $L^h, \Delta^h, \dots$; $L^h(\phi^h) : D^h \to \mathbb{R}$. Such operators are also defined on functions of $\boldsymbol{x} \in \mathbb{R}^3$, and on arrays defined on coarser grids $\phi^{h'}$, $h = Nh', N \in \mathbb{N}_+$, by sampling: $L^h(\phi) \equiv L^h(\mathcal{S}^h(\phi))$, $\mathcal{S}^h(\phi)[\boldsymbol{g}] \equiv \phi(\boldsymbol{g}h)$; $L^h(\phi^{h'}) \equiv L^h(\mathcal{S}^h(\phi^{h'}))$, $\mathcal{S}^h(\phi^{h'})[\boldsymbol{g}] \equiv \phi^{h'}[N\boldsymbol{g}]$.

For a rectangle $D = [\boldsymbol{l}, \boldsymbol{u}]$, defined by its low and upper corners $\boldsymbol{l}, \boldsymbol{u} \in \mathbb{Z}^3$, we define the operators

$$\mathcal{G}(D, r) = [\boldsymbol{l} - (r, r, r), \boldsymbol{u} + (r, r, r)], r \in \mathbb{Z}$$
$$\mathcal{C}(D) = \Big[\Big\lfloor \frac{\boldsymbol{l}}{N_{ref}} \Big\rfloor, \Big\lceil \frac{\boldsymbol{u}}{N_{ref}} \Big\rceil \Big]$$

Throughout this paper, we will use $N_{ref} = 4$ for the refinement ratio between levels.

The discrete Laplacian operator L^h

We begin our discussion presenting the finite difference discretizations of (1.3) that we will be using throughout this work and some of their properties that pertain to the Method of Local Corrections.

We are employing Mehrstellen discretizations [25] (also referred to as compact finite difference discretizations) of the 3D Laplace operator

$$(\Delta^h \phi^h)_{\boldsymbol{g}} = \sum_{\boldsymbol{s} \in [-s, s]^3} a_{\boldsymbol{s}} \phi^h_{\boldsymbol{g}+\boldsymbol{s}}, a_{\boldsymbol{s}} \in \mathbb{R}. \tag{2.1}$$

The associated truncation error $\tau^h \equiv (\Delta^h - \Delta)(\phi) = -\Delta^h(\phi^h - \phi)$ for the Mehrstellen discrete Laplace operator is of the form

$$\tau^h(\phi) = C_2 h^2 \Delta(\Delta\phi) + \sum_{q'=2}^{\frac{q}{2}-1} h^{2q'} \mathcal{L}^{2q'}(\Delta\phi) + h^q L^{q+2}(\phi) + O(h^{q+2}), \tag{2.2}$$

where q is even and $\mathcal{L}^{2q'}$ and L^{q+2} are constant-coefficient differential operators that are homogeneous, i.e. for which all terms are derivatives of order $2q'$ and $q+2$, respectively. For the 27-pt Laplacian operator considered here, $C_2 = \frac{1}{12}$. In general, the truncation error is $O(h^2)$. However, if $\Delta\phi = 0$ in a neighborhood of $\boldsymbol{x}$,

$$\tau^h(\phi)(\boldsymbol{x}) = \Delta^h(\phi)(\boldsymbol{x}) = h^q L^{q+2}(\phi)(\boldsymbol{x}) + O(h^{q+2}). \tag{2.3}$$

In our specific numerical test cases we make use 27-point (L^h_{27}) Mehrstellen stencil [61] that is described in the Appendix (Section B), for which $q = 6$. In general, it is possible to define operators for which $s = \lfloor \frac{q}{4} \rfloor$ for any even q, using higher-order Taylor expansions and repeated applications of the identity

$$\frac{\partial^{2r}\phi}{\partial x_d^{2r}} = \frac{\partial^{2r-2}}{\partial x_d^{2r-2}}(\Delta\phi) - \sum_{d' \neq d} \frac{\partial^{2r}}{\partial x_{d'}^{2r-2} \partial x_d^2}(\phi).$$

The discrete Green's Function

With a definition of the discrete Laplacian operator we can specify the discrete Green's Function G^h from the definition:

$$L^h G^h = 0 \qquad \text{when } \boldsymbol{i} \neq 0 \tag{2.4}$$

$$= 1 \qquad \boldsymbol{i} = 0 \tag{2.5}$$

Given such a G^h then the following property is true:

$$(G^h * f^h) = (\Delta^h)^{-1}(f^h) \ ,(G^h * f^h)[\boldsymbol{g}] \equiv \sum_{\boldsymbol{g}' \in \mathbb{Z}^3} h^3 G^h[\boldsymbol{g} - \boldsymbol{g}'] f[\boldsymbol{g}']^h \tag{2.6}$$

In traditional Green's function solvers the analytic form of the Green's Function in frequency space $\hat{\mathcal{G}}(k) = \frac{1}{k_0^2 - \boldsymbol{k}^2}$ for the Laplacian operator is often used. The analytic Green's Function limits your overall algorithm accurac to just 2nd-order.

For simple operators, like the 2D 5- or 9-point Laplacian it is possible to use analytic methods to compute the discrete Green's function as done by Buneman [19]. While the method can possibly be extended to higher-order schemes the procedure is cumbersome and error prone. We need to compute an approximation to the discrete Green's function G^h for the 27-point operator, restricted to a domain of the form $D = [-n, n]^3$. We do this by solving the following inhomogeneous Dirichlet problem on a larger domain $D_\zeta = [-\zeta n, \zeta n]^3$.

$$\begin{aligned}(L^{h=1}G^{h=1})[\boldsymbol{g}] =& \delta_{\mathbf{0}}[\boldsymbol{g}] \text{ for } \boldsymbol{g} \in \mathcal{G}(D_\zeta, -1),\\ G^{h=1}[\boldsymbol{g}] =& G(\boldsymbol{g}) \text{ for } \boldsymbol{g} \in D_\zeta - \mathcal{G}(D_\zeta, -1).\end{aligned}$$

Then our approximation to $G^{h=1}$ on D is the solution computed on D_ζ, restricted to D. To compute this solution, we put the inhomogeneous boundary condition into residual-correction form, and solve the resulting homogeneous Dirichlet problem using the discrete sine transform. In the calculations presented here, we computed $G^{h=1}$ using $n \geq 128$ and $\zeta = 2$, leading to at least 10 digits of accuracy for $G^{h=1}$.

2.2 The Two-Level Method of Local Corrections

We can describe the two-level algorithm here, but the analysis carries over to arbitrary depth of hierarchy. Given Δ^h, we define the discrete Green's function $G^h : \mathbb{Z}^3 \rightarrow \mathbb{R}$ by $\Delta^h G^h = h^{-3}\delta_{\mathbf{0}}$, where $\delta_{\mathbf{0}}$ is the 3D discrete delta function centered at $\mathbf{0}$. We have the relationships

$$\begin{aligned}G^h &= h^{-1}G^{h=1}\\ \Delta^{h=1}(G^{h=1,e})_{\boldsymbol{i}} &= O(||\boldsymbol{i}||_2^{-q-3})\\ \Rightarrow G^{h=1,e} - G^{h=1} &= O\left(||\boldsymbol{z}||_2^{-q}\right).\end{aligned}$$

To compute G^h on any bounded domain to any degree of accuracy, one computes and stores $G^{h=1}$ to any precision using a fast Poisson solver with Dirichlet boundary conditions given by $G^{h=1,e}$, or for sufficiently large $||\boldsymbol{i}||_2$, uses the approximation $G^{h=1}_{\boldsymbol{i}} \approx G^{h=1,e}_{\boldsymbol{i}}$.

Given these definitions, the two-level MLC computes

$$(G^h * f^h)_{\boldsymbol{i}} = h^3 \sum_{\boldsymbol{j} \in \mathbb{Z}^3} G^h_{\boldsymbol{i}-\boldsymbol{j}} f^h_{\boldsymbol{j}} \tag{2.7}$$

using a collection of local fields induced by the charge restricted to cubes of size $B_R \equiv [-R, R]^3$ combined with a single coarse-grid solution representing the global coupling. The local solutions are represented by the convolution for points near to the support of the charge (a box of size $B_{\alpha R}, \alpha > 1$) combined with a reduced representation given by the convolution of G^h with the low-order terms in the Legendre expansion of f^h for points at intermediate distances (a translate of the region $B_{\beta R} - B_{\alpha R}, \beta > \alpha$) from the support. The regions are shown for the 2D case in Figure 2.1. The same representation is used to compute the coarse-grid version of the charge whose field we use to represent the global coupling.

We denote by $f^{h,k} = f^h|_{B_R + 2\alpha R + 2kR}$, and $\mathbb{P}_{\mathcal{L}}$ the projection operator onto the Legendre polynomials of degree $\leq P - 1$ defined on $B_R + 2kR$ (the method in [53] corresponds to the case $P = 1$).

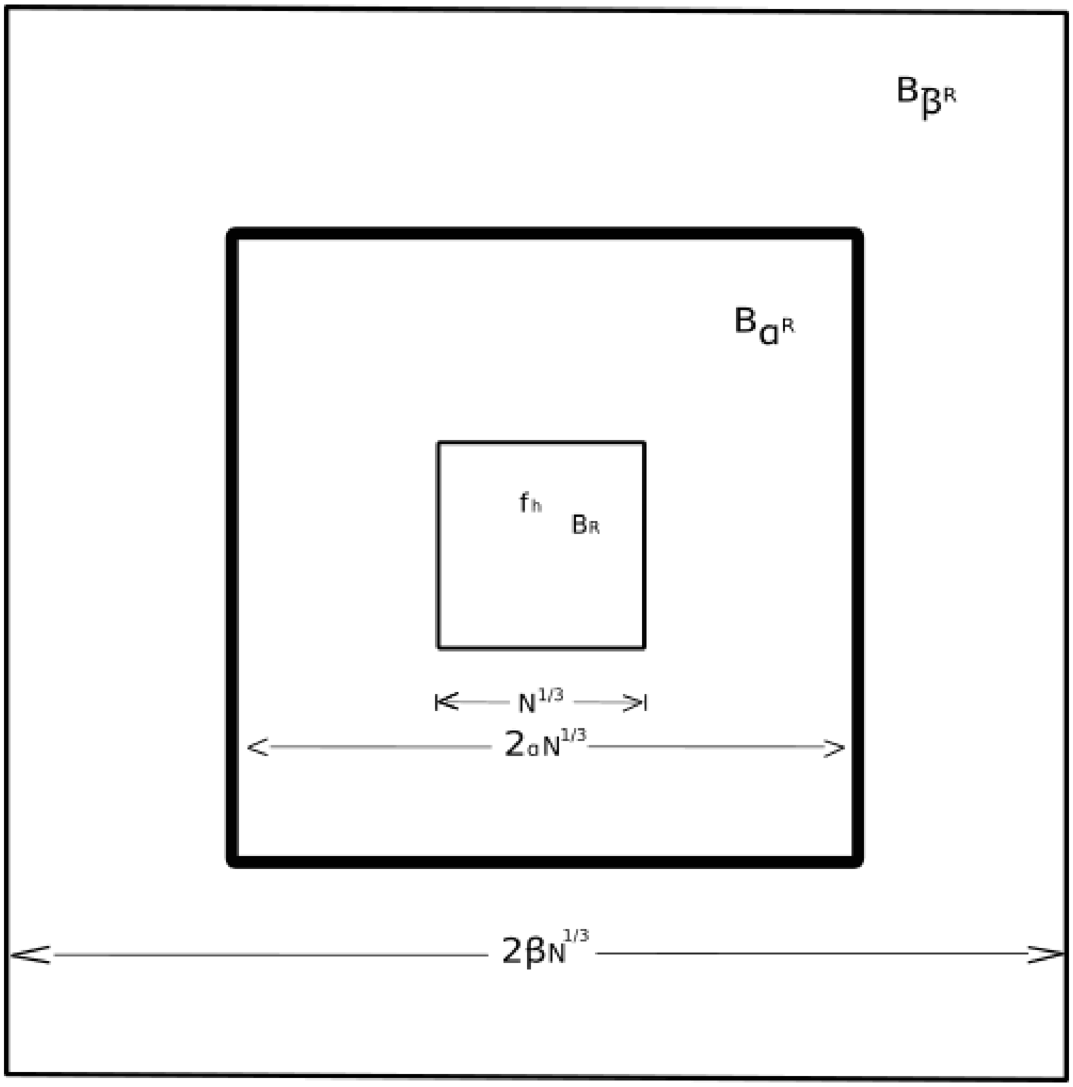


Figure 2.1: Regions about the patch k.

For $H = rh$, $r > 1$ is a positive integer, we define the coarse-grid right-hand side F^H

$$\begin{aligned}
F^{H,k} =& \Delta^H(G^h * f^{h,k}) \textbf{ on } B_{\alpha R} + 2kR && (2.8)\\
=& \Delta^H(G^{h,e} * (\mathbb{P}_{\mathcal{L}} f^{h,k})) \textbf{ on } (B_{\beta R} + 2kR) - B_{\alpha R} && (2.9)\\
=& 0 \text{ otherwise}\\
F^H =& \sum_k F^{H,k}.
\end{aligned}$$

A solution is computed on grid resolution H

$$\phi^H = G^H * F^H. \tag{2.10}$$

Then local corrections are computed as

$$\begin{aligned}
\Delta^h(\phi^h) &= f^h \text{ in } \ B_R && (2.11)\\
\phi^h &= \phi^h + \frac{h^2}{12} f^h && (2.12)\\
\phi^h_{B_R} &= \phi^{loc,\boldsymbol{i}}_{\boldsymbol{i}} + \mathcal{I}(\phi^H - \phi^{loc,\boldsymbol{i}})_{\boldsymbol{i}} \text{ on } \partial B_R && (2.13)\\
\phi^{loc,\boldsymbol{i}}_{\boldsymbol{i}'} &= \sum_{k:\mathcal{S}(\boldsymbol{i}) \subset B_{\alpha R}} +(G^h * f^{h,k})_{\boldsymbol{i}'} && (2.14)\\
&+ \sum_{k:\mathcal{S}(\boldsymbol{i}) \subseteq B_{\beta R} - B_{\alpha R}} (G^{h,e} * (\mathbb{P}_{\mathcal{L}m} f^{h,k}))_{\boldsymbol{i}'}. && (2.15)
\end{aligned}$$

Here $\mathcal{I}$ is a q_I^{th}-order accurate interpolation operator from the H grid to the h grid with stencil $\mathcal{S}(\boldsymbol{i})$ required to compute the interpolated value at $\boldsymbol{i}$. In practice, it is most efficient and accurate to construct $\phi^h_{\boldsymbol{i}}$ using the above construction only on the boundaries of each of the fine-grid patches as described in Equation (2.13). Then solve the Dirichlet problem from Equation (2.11) using a Discrete Sine Transform to compute the solution in the interior of the patch. In that way, we can also add the Mehrstellen corrections to the right-hand side to obtain high-order accuracy.

The error estimate of this scheme reduces to

$$\begin{aligned}
\epsilon^h = \phi^h - (G * f)^{h,e} = O(h^{q_0}) && (2.16a)\\
+ O\Big(h^{q_P}\Big(\frac{H}{\alpha R}\Big)^{q_\alpha}\Big) && (2.16b)\\
+ O(h^I) && (2.16c)\\
+ O\Big(\frac{H}{\beta R}\Big)^{q_\alpha}. && (2.16d)
\end{aligned}$$

In this error estimate, all the terms except the first are proportional to $||f||_\infty$. The first term (2.16a) is the contribution from the truncation error from the local convolution

$G * f^{h,k}$ and involves higher derivatives of f. Equation (2.16b) is the error contribution from evaluating the convolution on the reduced Legendre basis. P higher than q has diminishing gains from higher-order polynomial expansions. Equation (2.16c) is from the interpolation of the coarse-grid solution to the finer-grid. Equation (2.16d) is the result of discarding the contribution out past the limit β. As can be seen for a fixed ratio $\frac{H}{R}$, you can control this error term with either a wider region or a higher-order scheme. The high-order schemes q cost very little in this framework, whereas computation effort grows with the cube of β.

2.3 Analogy with FAS

As an explanatory aid, the Method of Local Corrections can be described as a modification of a *Full Approximation Scheme Multigrid Method* [17], where specific steps have been altered.

The two-level FAS scheme can be expressed as

$$\text{relax: } L^f(\phi^f) = F^f \tag{2.17}$$

$$\text{restrict: } F^c = L^c(I^c_f\phi^f) + I^c_f[F^f - L^f(\phi^f)] \tag{2.18}$$

$$\text{solve: } L^c(\phi^c) = F^c \tag{2.19}$$

$$\text{prolong: } \phi^f = \phi^f + I^f_c[\phi^c - I^c_f\phi^f] \tag{2.20}$$

$$\text{relax: } L^f(\phi^f) = F^f \tag{2.21}$$

FAS and MLC are structurally similar, with the differences allowing MLC to obtain a solution in one iteration. The first step is replaced by computing a solve with infinite-domain boundary conditions in MLC. In both cases, this step is the place where a fine-grid representation of the local high-wavenumber contribution is captured. In FAS, this is done only approximately, and with Dirichlet boundary conditions. The complete representation of the local contributions to the solution, as well as the correct far-field boundary conditions, are captured by iterating. In MLC this is done by computing a local convolution, which represents correctly the effect of the far-field boundary conditions on the local solutions.

The second step almost identical in MLC and FAS, since the fine-grid residual for the latter is identically zero in in the first step. The difference is that, in MLC the coarse-grid RHS is computed on a set of overlapping patches, and summed. The degree of overlap governs the accuracy of the final solution.

For both cases, the third step interpolates on the fine grid the coarse-grid representation of the nonlocal part of the solution. In the MLC case, this is done only on the boundaries of patches, with the solution in the interior computed by performing a final Dirichlet solve.

The final step computes a corrected version of the local solution using as inputs a version of the solution corrected for the effect of the smooth far-field behavior from distant patches. In the FAS case, this is again only done approximately, with iteration required to get a solution to the original discretized system. In the MLC case, the representation of the boundary conditions using the combination of nearby local convolutions and the global coarse-grid solution is used to obtain an approximate solution to the original discrete equations without iterating.

2.4 General Multilevel Algorithm

The complete algorithm description is now given. The pseudocode makes references to mathematical descriptions of operations as well as software implementation details. The implementation has been done as an extension to the Chombo C++ class library described in section 3.3. Data structures from Chombo are called out using a `Teletype` font using a "camel-case" naming convention.

Notation

- `Box`: C++ class representing a region of space in $\mathbb{Z}^D$. In math notation this corresponds to a specific Ω^l_k.
- `NodeFArrayBox`: C++ class representing scalar field over a `Box`
- `LevelData<NodeFArrayBox>`: C++ class representing a scalar field over a collection of `Box`s and meant to represent a field at a specific level of hierarchal grid refinement
- $\Omega^l = \bigcup_k \Omega^l_k$,$l = 0, \ldots, l_{\max}$ is a hierarchy of node-centered, nested grids with fixed-size `Box`es Ω^l_k. The size of each `Box` is $N^3, N = 2^M + 1$. We assume a refinement ratio of $r = 4$ between levels, and that the grids conform to an oct-tree structure. $\Omega^l_{k,0}$ denotes the interior points of Ω^l_k, $\partial\Omega^l_k = \Omega^l_k - \Omega^l_{k,0}$.
- $\Omega^{l;I}_k$ for interval I is set of all points $\boldsymbol{i}$ such that $||\boldsymbol{i} - \boldsymbol{i}_0||_\infty/2^{M-1}$ is in interval I, where $\boldsymbol{i}_0$ is the center of Ω^l_k.
- $f^l_k : \Omega^l_k \rightarrow \mathbb{R}$ defines the charge distribution on level l. Since we are computing the solution using linear superposition, can have different parts of the charge represented on overlapping regions of space on different levels.
- $\mathcal{C}$ is the coarsening operator, obtained for node-centered grids or data by sampling. $\mathbb{P}_{\mathcal{L}}$ is the projection onto the space spanned by the Legendre polynomials up to degree $P-1$. L^l is a Mehrstellen discretization of the Laplacian, at level l, G^l is the corresponding discrete Green's function, and $\mathcal{L}^l$ is the Mehrstellen correction operator applied to the right-hand side. Note that the latter is applied only to $f^{l,k}$, not $\tilde{f}^{l,k}$.
- $\mathcal{S}_d(\boldsymbol{i})$ is the set of all coarse grid points required to interpolate a value in dimension d at fine grid point $\boldsymbol{i}$. Define the interpolation radius b to be the minimum number such that for all $\Lambda \subset \mathbb{Z}^D$, $\mathcal{G}(\mathcal{C}(\Lambda), b)$ contains $\mathcal{S}_d(\Lambda)$.

Class `ExpansionCoefficients` is defined by a `Box` and an `int` number of components. We use the following data holders:

- Initial right-hand side for $l = 0, \dots, l_{\max}$:
  `LevelData<NodeFArrayBox>` f^l where $f^{l,k}$ is on Ω^l_k,
  stored as cubic `NodeFArrayBox` of length $N = 2^M$.

- Modified right-hand side for $l = 1, \dots, l_{\max}$:
  `LevelData<NodeFArrayBox>` $\tilde{f}^l$ where $\tilde{f}^{l,k}$ is on Ω^l_k,
  stored as cubic `NodeFArrayBox` of length $N = 2^M$.

- Legendre polynomial coefficients for $l = 1, \dots, l_{\max}$:
  `BoxLayoutData<ExpansionCoefficients>` a^l where $a^{l,k}$ is on Ω^l_k,
  stored as `ExpansionCoefficients` of length $\frac{(P+1)(P+2)(P+3)}{6}$ on $\Omega^{l;[0,\beta_l]}_k$.

- Difference with interpolated coarsened modified solution on faces $d\pm$ for $l = 1, \dots, l_{\max}$:
  `BoxLayoutData<FArrayBox>` $\delta^l_{d\pm}$ where
  $\delta^{l,k}_{d\pm}$ is on $\mathcal{G}_d(\partial^{\pm}_d \Omega^l_k, (\alpha_l - 1)2^{M-1})$,
  stored as `FArrayBox` of size $\alpha_l 2^M \times \alpha_l 2^M \times 1$.

- Modified coarsened right-hand side for $l = 1, \dots, l_{\max}$:
  `BoxLayoutData<NodeFArrayBox>` ρ^l where $\rho^{l,k}$ is on $\mathcal{G}(\Omega^{l;[0,\alpha_l]}_k, -s)$,
  stored as cubic `NodeFArrayBox` of length $\alpha_l 2^M / r - 2s$.

- Solution for $l = 0, \dots, l_{\max}$:
  `LevelData<NodeFArrayBox>` ϕ^l where $\phi^{l,k}$ is on Ω^l_k,
  stored as cubic `NodeFArrayBox` of length $N = 2^M$.

- `Labels` reference code instrumentation points used for performance measurement and referenced in chapter 4.

```
procedure MLC
    (Get $\tilde{f}$ at finest level, $l_{\max}$:)
    for each Box $k$ at level $l_{\max}$ do
        set $\tilde{f}^{l_{\max},k} = \chi^{\mathrm{w}}_{\Omega^{l_{\max}}_k}(f^{l_{\max},k})$ on $\Omega^{l_{\max}}_k$.
            (Save $\tilde{f}^{l_{\max},k}$ as cubic NodeFArrayBox of length $N = 2^M$.)
    end for

    for $l = l_{\max}, \dots, 1$ do                                        ▷ downwardPass
        (Get $a$ and $\tilde{\phi}$ and $\rho$ at this level, $l$:)
        for each Box $k$ at level $l$ do
            find coefficients $a^{l,k}$ of $f^{l,k;\mathrm{P}} = \mathbb{P}_{\mathcal{L}}(f^{l,k})$.
                (Save $a^{l,k}$ as ExpansionCoefficients of length $\frac{(P+1)(P+2)(P+3)}{6}$.)
            set $\tilde{\phi}^{l,k} = G^l * \tilde{f}^{l,k}$ on $\mathcal{G}(\Omega^{l;[0,\alpha_l]}_k, rb)$     ▷ Hockney::transform
                from $\tilde{f}^{l,k}$ on $\Omega^l_k$.
                (Find $\tilde{\phi}^{l,k}$ as cubic FArrayBox of length $\alpha_l 2^M + 2rb + 1$.)
            set $\delta^{l,k}_{d\pm} = \tilde{\phi}^{l,k} - \mathcal{I}_d(\mathcal{C}(\tilde{\phi}^{l,k}))$ on $\mathcal{G}_d(\partial^\pm_d\ \Omega^l_k,\ (\alpha_l - 1)2^M)$      ▷ InterpOnFace
                from $\tilde{\phi}^{l,k}$ on $\mathcal{G}_d(\partial^\pm_d \Omega^l_k, (\alpha_l - 1)2^M)$
                and $\mathcal{C}(\tilde{\phi}^{l,k})$ on $\mathcal{G}_d(\mathcal{C}(\partial^\pm_d \Omega^l_k), (\alpha_l - 1)2^M/r + b)$.
                (Save $\delta^{l,k}_{d,\pm}$ on $\mathcal{G}_d(\partial^\pm_d \Omega^l_k, (\alpha_l - 1)2^M)$      ▷ MLCFaces::setFaces
                as FArrayBox of size $\alpha_l 2^M \times \alpha_l 2^M \times 1$.)
            set formally $\tilde{\phi}^{l,k;\mathrm{P}} = G^l * f^{l,k;\mathrm{P}}$.
                (Coefficients $a^{l,k}$ of basis functions $G^l * Q^{l,k}_n$.)
            set $\rho^{l,k} = L^{l-1}(\mathcal{C}(\tilde{\phi}^{l,k}))$ on $\mathcal{G}(\mathcal{C}(\Omega^{l;[0,\alpha_l]}_k), -s)$      ▷ projectToCoarseT
                from $\mathcal{C}(\tilde{\phi}^{l,k})$ on $\mathcal{C}(\Omega^{l;[0,\alpha_l]}_k)$.
                (Save $\rho^{l,k}$ as cubic NodeFArrayBox of length $\alpha_l 2^M/r - 2s$.)
            set formally $\rho^{l,k;\mathrm{P}} = L^{l-1}(\mathcal{C}(\tilde{\phi}^{l,k;\mathrm{P}}))$ on      ▷ setLegendreCoeffs
                $\mathcal{G}(\mathcal{C}(\Omega^{l;[0,\beta_l]}_k), -s) - \mathcal{G}(\mathcal{C}(\Omega^{l;[0,\alpha_l]}_k), -s)$.
                (Coefficients $a^{l,k}$ of basis functions $L^{l-1}(\mathcal{C}(G^l * Q^{l,k}_n))$.)
        end for
        (Get $\tilde{f}$ at next coarser level, $l-1$:)
        for each Box $k'$ at level $l-1$ do
            initialize $\tilde{f}^{l-1,k'} = f^{l-1,k'}|_{\Omega^{l-1}_{k'} - \mathcal{C}(\Omega^l)}$ on $\Omega^{l-1}_{k'}$.
                (Save $\tilde{f}^{l-1,k'}$ as cubic NodeFArrayBox of length $N = 2^M$.)
            for each Box $k$ at level $l$ do
                increment $\tilde{f}^{l-1,k'} \overset{\pm}{=} \chi^{\mathrm{w}}_{\Omega^{l-1}_{k'}}(\rho^{l,k})$      ▷ generalCopyTo on a_rho
                    on $\Omega^{l-1}_{k'} \cap \mathcal{G}(\mathcal{C}(\Omega^{l;[0,\beta_l]}_k), -s)$.
                    from $\rho^{l,k}$ on $\Omega^{l-1}_{k'} \cap \mathcal{G}(\mathcal{C}(\Omega^{l;[0,\alpha_l]}_k), -s)$.
                increment $\tilde{f}^{l-1,k'} \overset{\pm}{=} \chi^{\mathrm{w}}_{\Omega^{l-1}_{k'}}(\rho^{l,k;\mathrm{P}})$      ▷ m_legendreCoeffsSampled
                    on $\Omega^{l-1}_{k'} \cap \big(\mathcal{G}(\mathcal{C}(\Omega^{l;[0,\beta_l]}_k), -s) - \mathcal{G}(\mathcal{C}(\Omega^{l;[0,\alpha_l]}_k), -s)\big)$.
                    (Evaluate with coefficients $a^{l,k}$ of basis functions
                    $L^{l-1}(\mathcal{C}(G^l * Q^{l,k}_n))$.)      ▷ addPolyConvolutionCoarseLaplacianVect
            end for
        end for
    end for
    (Get $\phi$ at coarsest level, 0:)
    set $\phi^0 = G^0 * \tilde{f}^0$ on $\Omega^0$      ▷ bottomSolve
        from $\tilde{f}^0$ on $\Omega^0$.
        (Save $\phi^0$ as cubic NodeFArrayBox of length $N = 2^M$.)

    for $l = 1, \ldots, l_{\max}$ do      ▷ upwardPass
        (Get $\phi$ at this level, $l$:)
        set formally $\phi^{l,\mathrm{loc}}_{\boldsymbol{i}} = \sum_{k':\mathcal{S}_d(\boldsymbol{i}) \subset \mathcal{C}(\Omega^{l;[0,\alpha]}_{k'})} \tilde{\phi}^{l,k'}_{\boldsymbol{i}} + \sum_{k':\mathcal{S}_d(\boldsymbol{i}) \subset \mathcal{C}(\Omega^{l;(\alpha,\beta]}_{k'})} \tilde{\phi}^{l,k';\mathrm{P}}_{\boldsymbol{i}}$.
        for each Box $k$ at level $l$ do
            for each face $\partial^\pm_d \Omega^l_k$ of $\partial\Omega^l_k$ do
```

```
                initialize $\psi^{l,k,d\pm} = \mathcal{I}_d(\phi^{l-1})$ on $\partial_d^{\pm}\Omega_k^l$,
                    from saved $\phi^{l-1}$ on $\mathcal{S}_d(\partial_d^{\pm}\Omega_k^l)$.
                    (Store $\psi^{l,k,d\pm}$ as FArrayBox of size $(N+1)\times(N+1)\times 1$.)
                for each Box $k'$ at level $l$ do
                    increment $\psi^{l,k,d\pm} \overset{\pm}{=} \tilde{\phi}^{l,k'} - \mathcal{I}_d(\mathcal{C}(\tilde{\phi}^{l,k'}))$ on $\partial_d^{\pm}\Omega_k^l \cap \Omega_{k'}^{l;[0,\alpha]}$,    ▷ m_phiDeltaMLCFaces.apply
                    increment $\psi^{l,k,d\pm} \overset{\pm}{=} \tilde{\phi}^{l,k';\mathrm{P}} - \mathcal{I}_d(\mathcal{C}(\tilde{\phi}^{l,k';\mathrm{P}}))$    ▷ addPolyConvolutionFineInterpDiffMLCFacesVect and m_legendreCoeffsAlphaBar
                        on $\partial_d^{\pm}\Omega_k^l \cap \Omega_{k'}^{l;(\alpha,\beta]}$.
                        (Evaluate with coefficients $a^{l,k'}$ of basis functions
                        $G^l * Q_n^{l,k'} - \mathcal{I}_d(\mathcal{C}(G^l * Q_n^{l,k'}))$.)
                end for
            end for
            solve $\phi^{l,k}$ on $\Omega_k^l$:    ▷ solveInhomogeneousBCInPlace

                    $L^l\phi^{l,k} = \tilde{f}^{l,k}$ on $\Omega_{k,0}^l$;
                    $\phi^{l,k}|_{\partial_d^{\pm}\Omega_k^l} = \psi^{l,k,d\pm}$ ,for each face $\partial_d^{\pm}\Omega_k^l \subset \partial\Omega_k^l$

            (Save $\phi^{l,k}$ as cubic NodeFArrayBox of length $N = 2^M$.)
        end for
    end for

    for $l = 0, \ldots, l_{\max}$ do    ▷ Mehrstellen
        for each Box $k$ at level $l$ do
            increment $\phi^{l,k} \overset{\pm}{=} \frac{h_{l_{\max}}^2}{12} f^{l,k}$ on $\Omega_k^l$.
        end for
    end for
end procedure
```

Notes

1. Here $1 < \alpha < \beta$. In our current stage of testing, typical values are $M = 5$, $\alpha = 1.5 - 2$, $\beta = 3 - 6$. The proper nesting conditions are that $\mathcal{C}(\Omega_k^{l;[0,\alpha]}) \subseteq \Omega^{l-1}$, and that $\mathcal{C}(\Omega_k^{1;[0,\beta]}) \subseteq \Omega^0$.

2. The convolutions $G^l * \tilde{f}^{l,k}$, $G^l * f^{l,k';\mathrm{P}}$ are computed once, and used multiple times: once in computing the $\tilde{f}$'s, the second time in computing the ϕ^{loc}'s in the boundary conditions for the final solves. In both cases, we only use a reduced subset of all of the values.

3. Note that we have $f^{l,k;\mathrm{P}} = \mathbb{P}_{\mathcal{L}}(f^{l,k})$, not $\mathbb{P}_{\mathcal{L}}(\tilde{f}^{l,k})$.

4. Note that interpolations are 2D, not 3D.

5. For order higher than 4th, Mehrstellen correction must be done on the right-hand side in the Dirichlet problem, not applied to the solution later. For 4th order, it works out to be the same.

Use of Legendre polynomial expansions

For each $l = 1, \ldots, l_{\max}$, and for each patch k at level l, we take the projection of $f^{l,k}$ onto the Legendre polynomial basis functions $Q_n^{l,k}$:

$$f^{l,k;\mathrm{P}} = \mathbb{P}_{\mathcal{L}}(f^{l,k}).$$

The coefficients are $a^{l,k}$ for Ω_k^l, saved as `ExpansionCoefficients` of length $(P+1)(P+2)(P+3)/6$.

We need to evaluate the following functions of the Legendre polynomials and multiply them by the saved coefficients:

- For each $l = 1, \ldots, l_{\max}$, and for each patch k at level l, we need
  $L^{l-1}(\mathcal{C}(G^l * Q_n^{l,k}))$ on $\mathcal{G}(\mathcal{C}(\Omega_k^{l;(\alpha_l,\beta_l]}), -s)$.
  These are used in forming the modified right-hand side in the downward pass.

- For each $l = 1, \ldots, l_{\max}$, and for each patch k at level l, for each dimension d and for each $\pm$ we need
  $G^l * Q_n^{l,k} - \mathcal{I}_d(\mathcal{C}(G^l * Q_n^{l,k}))$ on $\mathcal{G}_d(\partial_d^{\pm}\Omega_k^l, (\beta_l - 1)2^{M-1})$ shifted by $\pm 2^M i$ in dimension d, for $0 \le i \le \lfloor(\beta - 1)/2\rfloor$. So in each dimension there are $2\lfloor(\beta+1)/2\rfloor$ faces.
  These are used in finding the boundary values on box faces in the upward pass.

Polynomial $Q_n^{l,k}$ is centered at the center of box Ω_k^l. Take $Q_n(x, y, z)$ to be the n^{th} Legendre polynomial basis function centered at the origin. Then we evaluate functions of Q_n at the following points:

- $L(\mathcal{C}(G * Q_n))$ on $[-\beta - s\frac{r}{2^{M-1}}, \beta + s\frac{r}{2^{M-1}}]^3$ with grid spacing $\frac{r}{2^{M-1}}$;

- $G * Q_n - \mathcal{I}_d(\mathcal{C}(G * Q_n))$ for each dimension d, on the faces that are i in dimension d, and $[-\beta, \beta]$ in the other two dimensions, for each *odd* i such that $|i| \le \beta$, with grid spacing $\frac{1}{2^{M-1}}$.

2.5 Hockney's Algorithm: The Heart of MLC

While the fully-defined MLC algorithm has many components, the dominant computational kernel is the discrete free-space convolution operation

$$G^{l+1} * \tilde{f}^{l+1,k'} \tag{2.22}$$

which we compute using *Hockney's Algorithm* [41]. For *any* discrete convolution of a compact source term f^h, we can compute a solution to the full space convolution with just a finite circular convolution on a domain that is double the size of the support of f^h where f^h set to be identically zero on the extended domain.

$$\Phi^h(\boldsymbol{j}) = (G^h * f^h)(\boldsymbol{j}) = (2n)^3 \mathcal{F}^{-1} \left[\sum_{\boldsymbol{k} \in [-n+1,n]^3} \hat{G}^h_{\boldsymbol{k}} \hat{f}^h_{\boldsymbol{k}} z^{\boldsymbol{k}\cdot\boldsymbol{j}} \right]. \tag{2.23}$$

A detailed derivation of this fact is given in Appendix C. $\hat{G}^h$ can be precomputed and stored. Additionally $\hat{G}^h$ has several symmetries, and hence we only calulate and load a $\frac{1}{8}$th octant.

The important observation in that this results in performing many small batch *Fast Fourier Transforms.* The 3D FFT for a small patches $N \leq 64^3$ has not received much attention in the literature. Even Williams [67] only starts with 128 cubed. Similar results are published for FFTW [36] and Spiral. At this small a size all Fast FFT schemes seem to get the same performance [45].

While the derivation of Hockney's Algorithm are expressed in what is called *Domain doubling*, in practice you do **not** solve an 8 times larger complex DFT transform padded out with zeros. First, the inputs $\hat{f}$ and G are real-valued, and thus so are the outputs. Second, there is no need to compute the DFT of a zero vector. With a suitable change in variables the 2D free-space convolution can be diagrammed as in figure 2.2. The domain has been sized to show the amount of storage needed.

1. Loading in this specific patch's source term f.

2. The i-direction transforms are done on half the domain size since they are Real-to-Complex. We do not need to transform the $2(n_s + n_d + 1)$ real-values in the i-direction since the majority of them are zero-valued, only n_s of them are non-trivial.

3. The j-direction Forward FFT transform is a full Complex-to-Complex transform. In the 3D case this one is has a large trivial transform space in the k-direction that is excluded.

4. Multiplication by the symbol. The algorithm can exploit two facts here. Since G is real and symmetric about the origin, we know that $\hat{G}$ is also real-valued and symmetric about the origin. We use this fact to reduced storage and load-store traffic.

5. Inverse FFT transform in the j-direction. As with step 3, there is a gain to be had here, not because the transforms are trivial, but because we are only interested in specific values on the output. In 2D this is not evident but in 3D we can forgo half the transforms.

6. Complex-to-Real FFT transform. In this case we are only interested in the transforms in the range $n_s + n_d + 1 > j > n_s$.

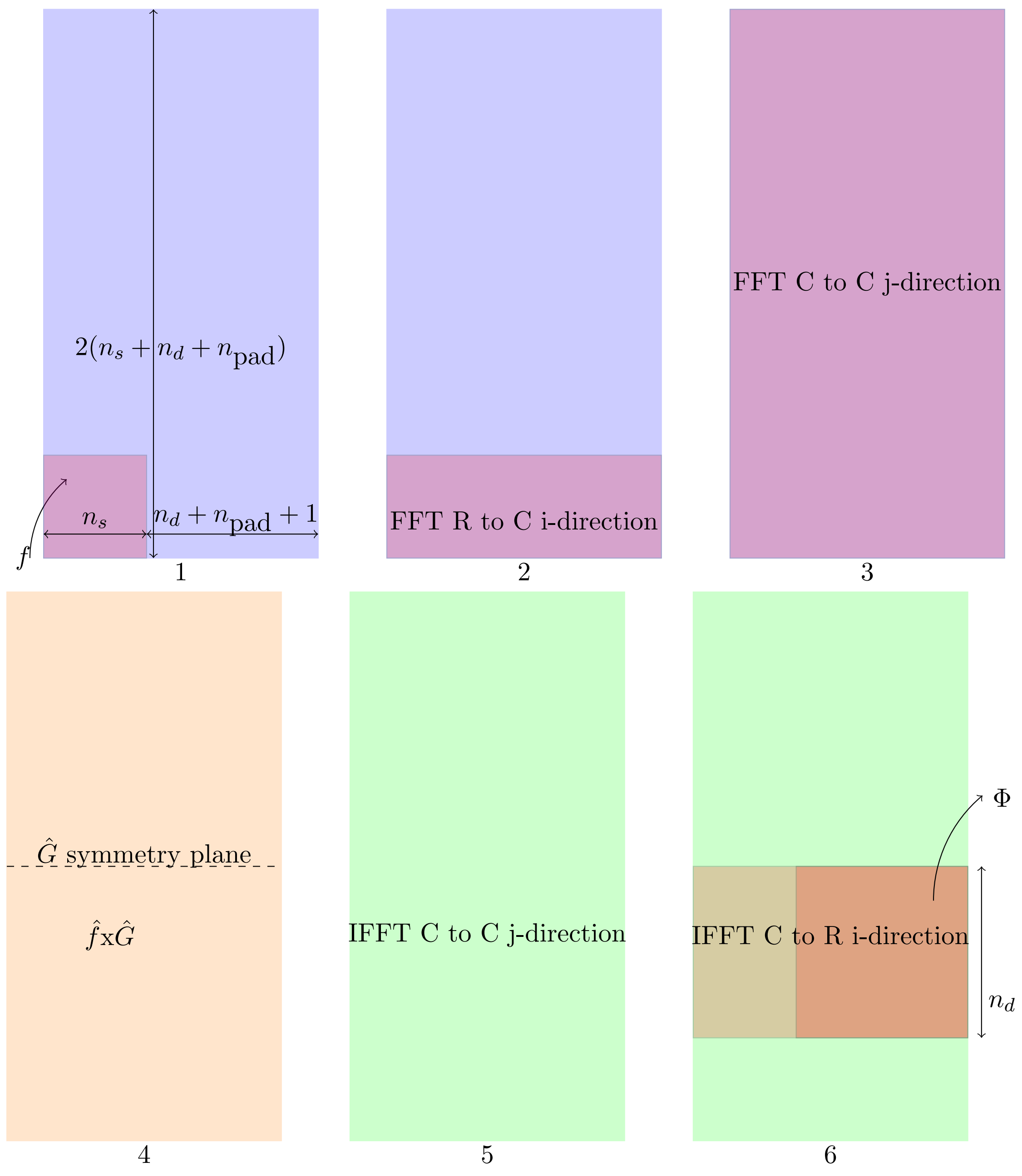


Figure 2.2: 2D Convolution

MLC exploits the specific zero-structure of the inputs (zero padding for free-space solver), and subsampling of the outputs (sampling onto the coarser grid, and the `MLCFaces` on the same level of refinement). These optimizations are generally referred to as *pruning.* In the 1D case there a modest gain on modern superscalar architectures from exploiting pruning, on the order of 10-30% [34, 35]. For higher dimension transforms the reduction in floating-point operations can be helpful, but the real wins are from the loads and stores you do not end up having to perform. For small compact 3D Hockney transformations we replace the typical $5N\log_2(N)$ operation cost model with

$$\begin{aligned}
&\text{Let } n_s = N^{\frac{1}{3}}, n_d = \alpha n_s, M = n_s + n_d \\
&\quad \frac{5}{2}[n_s n_s M \log_2(M) + n_s M M \log_2(M) + 2MMM\log_2(M) \\
&\quad + 2n_d M M \log_2(M) + n_d n_d M \log_2(M)] \\
&\quad = [10(\alpha^3+1) + 25(\alpha^2+\alpha)]N\log_2(M) \qquad (2.24)
\end{aligned}$$

where 5 is turned into $\frac{5}{2}$ since these are all real, not complex transforms. As expected the convolution is cubic in α for cost. This is quite a bit better than a full doubled-domain FFT which would require $80M\log_2(8M)$ operations.

In this work there are several approaches to parallelism implemented and compared. All are expressed in terms of *hybrid parallelism* using MPI as the mechanism for managing computations in distributed memory domains and OpenMP for executing and coordinating computations within shared memory domains.

In MPI each parallel execution unit is referred to as a *rank.* In distributed memory computation the address space of each rank is private and data is transmitted to other ranks by messages.

In OpenMP the unit of execution is a *thread.* In OpenMP programming the address space of each thread is by default shared. OpenMP is a fork-join parallelism model based on source-level code annotation and OpenMP-aware compilers and an OpenMP runtime.

Variant 1 and 2: Flat MPI and Coarse-grained threading

In the first hybrid variation of the MLC algorithm traditional patch-based parallelism design of Chombo is preserved. In this execution-mode the outer patch-by-patch loop is first load balanced across the MPI ranks. Within each MPI rank execution each for-loop over these patches is assigned to indepedent threads of execution. This is the least disruptive code modification to the Chombo framework. The code style looks like

```
DataIterator dit = level.dataIterator();
#omp parallel for
for(int i=0; i<dit.nlocalPatches(); i++)
{
  const NodeFArrayBox& f = level[dit(i)]; //retrieve i'th patch on this rank
```

operation	howmany	input stride	input dist	output stride	output dist
R2C DFT i-	n_s^2	1	N	1	N/2+1
DFT j-	$n_s(\frac{N}{2}+1)$	$\frac{N}{2}+1$	1	$\frac{N}{2}+1$	1
DFT k-	$N(\frac{N}{2}+1)$	$N(\frac{N}{2}+1)$	1	$N(\frac{N}{2}+1)$	1
Multiply by $\hat{G}$	$N(\frac{N}{2}+1)$				
IDFT k-	$N(\frac{N}{2}+1)$	$N(\frac{N}{2}+1)$	1	$N(\frac{N}{2}+1)$	1
IDFT j-	$n_d(\frac{N}{2}+1)$	$\frac{N}{2}+1$	1	$\frac{N}{2}+1$	1
C2R i-	n_d^2	1	$\frac{N}{2}+1$	1	N

Table 2.1: Steps for coarse-grained Hockney transforms. The column headings are a naming convention used by the FFTW API and described in section 3.4.

```
  expansionCoefficients[dit(i)].computeLegendreCoefficients(f);
  .
  hockney.transform(f, phi_alpha);
  MLCFaces.addTo(phi_apha);
  fCoarse.addto(coarsen(phi_alpha));
  .
  .
}
```

This source code can be compiled and executed with no change for a *flat MPI* execution, where the user does not have or wish to use an OpenMP-aware compiler.

The Hockney Transform is executed in a sequence of 7 steps. All the FFT transforms are rank=1 batched operations of size $N = n_s + n_d + 1 +$ padding. Since the free-space circular convolution is zero-padded we have a degree of flexiblity in picking a positive integer degree of padding to achieve highly composite transforms.

Variant 3: Fine-grained threading

In variant 2 of the Hockney convolution the outer parallelism is managed by the MPI rank in the usual SPMD parallel execution model. The difference now is that the transform itself is performed cooperatively amongst the threads in shared memory. All steps are executed as a single OpenMP parallel region. While the algorithm has 7 stages it is decomposed to have only 3 fork-join thread regions statically scheduled with `omp for` directives.

Variant 4: Fine-grained compact threading

Variant 1 and 2 both have the same data layout and require a transient data container of size $(\frac{N}{2}+1)$xNxN complex values for $\hat{f}$. In this variant we perform the foward k-direction transform, multiply by the real-valued symbol $\hat{G}$, and perform the inverse k-direction transform in succession in a temporary thread-private data buffer. In this case the working set

`omp for`	operation	howmany	input stride	input dist	out stride	out dist
n_s	R2C DFT i-	n_s	1	N	1	N/2+1
	DFT j-	$\frac{N}{2}+1$	$\frac{N}{2}+1$	1	$\frac{N}{2}+1$	1
N	DFT k-	$N(\frac{N}{2}+1)$	$N(\frac{N}{2}+1)$	1	$N(\frac{N}{2}+1)$	1
	Multiply by $\hat{G}$	$N(\frac{N}{2}+1)$				
	IDFT k-	$N(\frac{N}{2}+1)$	$N(\frac{N}{2}+1)$	1	$N(\frac{N}{2}+1)$	1
n_d	IDFT j-	$\frac{N}{2}+1$	$\frac{N}{2}+1$	1	$\frac{N}{2}+1$	1
	C2R i-	n_d	1	$\frac{N}{2}+1$	1	N

Table 2.2: Steps for fine-grained threaded Hockney transforms. A new column now appears that describes how threads are distributed across the operations within the transform.

`omp for`	operation	howmany	input stride	input dist	out stride	out dist
n_s	R2C DFT i-	n_s	1	N	1	N/2+1
	DFT j-	$\frac{N}{2}+1$	$\frac{N}{2}+1$	1	$\frac{N}{2}+1$	1
$\frac{N^2}{2n_v}$	DFT k-	n_v	$N(\frac{N}{2}+1)$	1	$N(\frac{N}{2}+1)$	1
	Multiply by $\hat{G}$	n_v				
	IDFT k-	n_v	$N(\frac{N}{2}+1)$	1	$N(\frac{N}{2}+1)$	1
n_d	IDFT j-	$\frac{N}{2}+1$	$\frac{N}{2}+1$	1	$\frac{N}{2}+1$	1
	C2R i-	n_d	1	$\frac{N}{2}+1$	1	N

Table 2.3: Steps for compact fine-grained threaded Hockney transforms

is reduced to $\left(\frac{N}{2}+1\right)xNxn_d$+nthreadsx$Nxn_v$ where n_v is a tunable parameter. We call this the *k pencil* and typically choose a size to be a multiple of the vector architecture size. The scheduling is the same to variant 2 in term of fork-join structure. There is the added option of how the threads are scheduled to execute the pencil transforms.

The final variant can be written as a more readable subroutine in `Hockney` outlined in algorithm 1. below:

Algorithm 1 Hockney Variant 4 expressed as pseudocode

```
procedure HOCKNEY(Real f[ns,ns,ns],Real $\tilde{f}$[nf,nf,nf], Real $f_c$, Real $\Phi_f$(faces), Real $\hat{G}$)
    Complex fc[N/2+1,N,nd]
    Complex pencil[v,1,N] threadprivate
    for k do=0 ... ns over nthreads
        fc[0:ns/2-1,0:ns/2-1,k] $\leftarrow$ f[0:ns-1,0:ns-1,k]
        fc[ns:N-1, 0:ns-1,k] $\leftarrow$ 0
        fftw_execute_dft_r2c(multiR2C, fc[0,0,k])
        fc[0:N/2+1,ns:N-1,k] $\leftarrow$ 0
        fftw_execute_dft(multiJ, fc[0,0,k])
    end for
    for j do=0 ... N over nthreads
        for i do=0 ... N/2+1 increment v
            Pencil[0:v,0,0:ns] $\leftarrow$ fc[i:i+v,j,0:ns]
            Pencil[0:v,0,ns+1:N] $\leftarrow$ 0
            fftw_execute_dft(vPencilK, Pencil[0,0,0])
            Pencil $\leftarrow$ Pencil*$\hat{G}$[i:i+v,j,0:N]
            fftw_execute_dft(vPencilKinv, Pencil[0,0,0])
            fc[i:i+v,j,0:nd] $\leftarrow$ Pencil[0:v,0,0:nd]
        end for
    end for
    for k do=0 ... nd over nthreads
        fftw_execute_dft(multiJinv, fc[0,0,k])
        fftw_execute_dft_c2r(multiC2R, fc[0,0,k])
        $f_c$ += L(sample(fc[0:nd,0:nd,k], 4))                    $\triangleright$ L/S $f_c$
        $\Phi_f$(faces) += fc[0:nd,0:nd,k]                        $\triangleright$ L/S $\Phi_f$(faces)
        $\tilde{f}$ += fc[0:nd,0:nd,k]            $\triangleright$ L/S $\tilde{f}$ (only used for 117-pt operator
    end for
end procedure
```

Chapter 3

Software and Tools

3.1 Block-Structured Adaptive Mesh Refinement

The Method of Local Corrections (MLC) is motivated by the need to solve Poisson's equation where the source function is defined on logically cartesian-structured grid points given on the hierarchy of properly nested structured grids. This class of algorithms is generally referred to as *Block-Structured Adaptive Mesh Refinement* (SAMR). This arrangement is motivated by many high-performance computing simulation techniques applied to partial differential equations. This was formally specified in section 2.1.

- Kernels executing on small, structured grid patches with arithmetic intensity in the range of .1 to 1

- Patches have *ghost cells* which are local, user-mananaged caches of data from adjacent patches. Ghost cells are filled in at specific user-controlled points in the calculation to allow each patch to be processed independently in a SPMD compute model. The number ghost cells depends on support required for stencil evaluation. Typcially stencils have a radius of 1 to 6 cells.

- Ghost cells are an application level technique to protect against read-after-write data hazards.

- Ghost cells are initialized via a communication phase from either adjacent patches at the same level of refinement or from interpolation of data on coarser refinement levels.

The performance of AMR algorithms can critically depend on the choice of how many ghost cells to use. Choosing a small number of ghost cells (1 or 2) has the following advanges: Fewer redundant calculations, smaller memory footprint, less data traffic, small communication volume, better utilization of cache. The disadvantages are the need for more frequent communication phases which will usually result in loss of performance due to memory and network latency and time managing the data choreography from data marshaling routines.

For hyperbolic partial differential equations, the advantages of adaptive mesh refinement outweigh disadvantages of added algorithm complexity and complex data management. For elliptic systems like Poisson's equation, the dominant technique for solving these equations is geometric multigrid. The design space for optimizing geometric multigrid by manipulating the tradeoff between ghost cell size and architecture latency is explored in [66, 14, 65]. Even with these communication-avoiding optimizations, multigrid remains a latency-bound algorithm and will become increasingly so in exascale platforms.

Software for AMR is optimized around a design point where the patches have an ordinal dimension between the 32 and the 64. Going much lower than 32 increases the relative cost of adaptivity. Going much larger than 64 eliminates most of the benefits that adaptivity was giving you. Given this design point, AMR software libraries are tuned for ghost cells in the range from 2 to 8 in the extreme cases.

We will discuss augmentations to the Chombo block-structured adaptive mesh refinement C++ package in section 3.3.

3.2 Modalities of Parallel Processing

Distributed Memory Computing

Chombo and HDF5 both help manage distributed parallel computing built on top of the MPI message-passing interface [56]. Given a hierarchy of block-structured, properly nested, logically rectangular patches, the Chombo load-balancing algorithms will assign patches to MPI ranks. The communication between ranks is needed on the *downsweep* of MLC to accumulate partial contributions on to neighboring patch faces as well as propagate the coarsened charge on to the next coarser level of refinement. The *upsweep* phase can run fully asynchronous.

At the distributed-memory level parallelism, MLC functions as an SPMD program. Communication phases are handled as distinct epochs that are highlighted in the timing data as the `copyTo` signifier.

Shared Memory Multithreading

Although contemporary multiprocessing cores can be treated as ranks in a valid distributed computing execution, the slight overheads of SPMD programming model have reached the point of diminishing returns. The clock speeds have not been increasing. The amount of cache hierarchy available to the processing core and the amount of off socket network interconnect relative to the core have all stalled out. Hence, MLC is implemented in a hybrid parallel model where the threading is handled using OpenMP [57]. OpenMP is a portable shared memory API for fork-join parallelism. Two styles of MPI plus Open MP designs have been deployed in this thesis. The first I call *coarse threading*. This is the simplest to implement. In this model, OpenMP threads emulate MPI ranks. This has several advantages. The strict semantics of distributed memory SPMD programming means that it is very easy to write race-free hybrid programs. These *fat* MPI ranks can share their per-rank metadata which at extreme levels of concurrency can limit the ability to utilize the available random access memory.

MPI+OpenMP *fine threading* leaves the MPI-distributed software design as-is and implements fork-joined fine-grained OpenMP threading within each of the sequentially invoked kernels in the algorithm. This has the same advantages of the MPI fat node design in terms of metadata usage. There is a more intrusive software rearchitecting. The advantages are more flexibility in terms of using threads in a different scheduling design space and alters the data reuse patterns. It has the potential for better cache utilization. These advantages need to be weighed against the added penalties of OpenMP Library fork-join overheads. Ultimately, hybrid designs are employed in the final configuration.

3.3 Chombo

Chombo is a C++ Class Library used for implementing solvers for partial differential equations on block-structured adaptive grids. It has been developed in the Applied Numerical Algorithms Group at Lawrence Berkeley National Laboratory over the last 20 years with myself as one of the primary authors [24].

Augmentations to Chombo for MLC

In contrast to traditional AMR discretizations, the intralevel communication in MLC is not volumetric over the ghost region but over the surface of the ghost region. Additionally, the ghost regions are defined by their parameters α and β (see figure 2.1) which can range from 1.5 to 2.5 multiplying the ordinal dimension. Hence, ghost regions can range from 25 up to 100 ghost cells. This is well outside the classic design space for these software packages. The existing code base uses an $O(\log_2 N)$ algorithm to compute nearest-neighbor communication patterns based on patches having a lexicographic ordering. Lexicographic ordering proved to have a large constant multiplier for extreme ghost-cell sizes. A new algorithm was developed using a hashing function based on *Morton ordering* [55]. A fast conversion from `Box` coordinates in physical space to a unique hash value was used from the graphics community [9]. It uses a fast lookup table method and some bit-twiddling operations.

A new class called `MLCFaces` was created in Chombo to represent the α and β faces and is represented in the MLC pseudocode as $\delta^{l,k}_{d\pm}$. In the existing Chombo framework each + and - face in each direction would require its own bulk synchronous communication phase. This effect was swamping the communication gains over iterative methods. Rolling all directions together in one pass required extending Chombo. In addtion, this communication phase is fused with the destination in-place update logic to better utilize the memory caching.

3.4 FFTW

The Fastest Fourier Transform In The West (FFTW) [35] is a software library for computing discrete Fourier Transforms developed by Matteo Frigo and Steven Johnson at Massachusetts Institute of Technology written in a combination of C and OCaml. FFTW utilizes a combination of symbolic simplification, code-generation and auto-tuning. This is given as a C library with function definitions given in `fftw.h` and a static archive linked against `libfftw.a`. A user defines DFT transforms using the *Advanced Interface*:

```
fftw_plan fftw_plan_many_dft_r2c(int rank, const int *n, int howmany,
double *in, const int *inembed, int istride, int idist, fftw_complex
*out, const int *onembed, int ostride, int odist, unsigned flags);

fftw_plan fftw_plan_many_dft_c2r(int rank, const int *n, int howmany,
fftw_complex *in, const int *inembed, int istride, int idist, double
*out, const int *onembed, int ostride, int odist, unsigned flags);

fftw_plan fftw_plan_many_dft(int rank, const int *n, int howmany,
fftw_complex *in, const int *inembed, int istride, int idist,
fftw_complex *out, const int *onembed, int ostride, int odist, int
sign, unsigned flags);
```

These `istride, idist, onembed, ostride, odist` parameters are used in the descriptions of the Hockney kernel variants.

Since advanced transform pruning options are not available in standard FFTW the automatic thread planning offered in "*5.2 Usage of Multi-threaded FFTW*" is not utilized.

3.5 HDF5

The Hierarchical Data Format Version 5 [62] is used as my parallel IO middleware. It provides a cross-platform distributed computing parallel file system interface. This is used for loading in pre-computed convolutions and Poisson Green's Functions and writing solutions in parallel. The HDF5 interface only currently supports an MPI parallel IO API at this time. The discrete Green's function G^h and the the Laplacian of the coarsened convolutions for the n Legendre basis function $L^{l-1}(\mathcal{C}(G^l * Q_n^{l,k}))$ are precomputed values and are loaded into memory at the start of execution. Doing this efficiently in parallel requires care and we utilized the Cray Burst Buffer[16, 58] to preload these data structures from the hard disk into faster non-volatile memory before the execution begins. Then HDF5 parallel IO is used to bring this data into DRAM.

Chapter 4

Performance Models

The key computational kernel in the algorithm is Hockney's algorithm which is analyzed here in more detail. It is stated as a Discrete Fourier Transform on a doubled domain. In our case with domain decomposition we are interested in quite small 3D FFT computations.

The classic 3D FFT is built from computations of the 1D FFT and data movement. So first we need the complexity of a 1D FFT: $5N\log(N)$ flops. The original derivation from Cooley-Tukey is from 1965 [26]. All fantastic modern versions of FFT can change the constant by about 15% (often times higher) but arrange the flops in different ways that are friendly to the processor and memory system. It is important to note that the log is base 2, not e or 10. That makes a difference of $\log_{10}(2) = 0.69$ which is a bigger effect in all the analysis than the other factors. A 3D FFT over a rectangular domain is decomposed into a sequence of batched 1D FFTs. N defined on a rectangular domain as $N = n_1 n_2 n_3$. 3D FFT is done as $n_1 n_2 5 n_3 \log(n_3) + n_2 n_3 5 n_1 \log(n_1) + n_1 n_3 5 n_2 \log(n_2) = 5N\log(N)$. To use FFT for convolutions $G * f$ you need three transforms: $\text{FFT}(G^h) = \hat{G}^h, \text{FFT}(f^h) = \hat{f}^h, \text{FFT}^{-1}(\hat{G}\hat{f}) = G^h * f^h$. $\hat{G} = \text{FFT}(G^h)$ is precomputed in this algorithm.

Complexity Analysis in big O Notation

FFT, GMG V-cycle. Full Multigrid V-cycle and MLC are compared in overall complexity analysis in table 4.1. It is hard to make any definitive statements about the relative merits of each method with standard complexity analysis. In the long range it resembles GMG, but the constants matter in these matrix-free methods and GMG is an iterative method.

It should be noted that this analysis ignores effects like local memory bandwidth and bisection limitations on proposed exascale hardware. As noted in [27] the true limiting effect for FFT seems to be a combination of link contention for transposes, and local memory bandwidth for very large local transpose operations. MG and MLC both scale down their number of messages and bandwidth with longer distance communication and saturate bandwidth only in a nearest-neighbor fashion. In [15] there is a discussion of trading off more messages of a smaller size while having the same total amount of communication. Chan

Scheme	Flops	# Messages	# Words
V-cycle GMG	N	$\log(N)$	$O(p(\frac{N}{p})^{\frac{2}{3}}(1+\frac{1}{8}+\frac{1}{64}+...) = p^{\frac{1}{3}}N^{\frac{2}{3}} + p\log(p)$
FMG	N	$\log^2(N)$	$p^{\frac{1}{3}}N^{\frac{2}{3}} + p\log(p)\log(N)$
FFT	$N\log(N)$	$\min(p^2, N^{\frac{2}{3}})$	N
MLC	$N\log(N/d)$	$\log(N)$	$p^{\frac{1}{3}}N^{\frac{2}{3}}$

Table 4.1: big O Complexity for a given p for GMG V-cycle, FMG, FFT and MLC. N grid points, p processors. d refers to the amount of domain decomposition used in MLC. FFT is done as a single large FFT operation, while MLC first decomposes the domain into d compact disjoint regions.

et al. [20] considered the use of 2D decompositions instead of 1D decompositions and get different trade offs in messages which seems to approach the pencil limit case in [15].

Operation	Flops/Gridpoint
$\mathbb{P}_{\mathcal{L}}f^h$	3Γ
$\Delta^H(G^h * f^{h,k})$	$[10(\alpha^3+1)+25(\alpha^2+\alpha)]\log_2[(\alpha+1)N^{\frac{1}{3}}]$
$\Delta^H(G^h * (\mathbb{P}_{\mathcal{L}}f^{h,k}))$	$\Gamma(2\beta)^3/r^3 + 6\lfloor\beta\rfloor\Gamma/N^{\frac{1}{3}}$
$\Delta^{-1}\left(f^h + \frac{h^2}{12}\Delta^h(f^h)\right)$	$\frac{5}{2}\log(N) + qn + 1$

Operation	`load`	`store`
$\mathbb{P}_{\mathcal{L}}f^h$	1	Γ/N
$\Delta^H(G^h * f^{h,k})$	0*	$6\lfloor\alpha\rfloor/N^{\frac{1}{3}} + (2\alpha)^3/r^3$
$\Delta^H(G^h * (\mathbb{P}_{\mathcal{L}}f^{h,k}))$	$(2\beta)^3/r^3 + 6\lfloor\beta\rfloor/N^{\frac{1}{3}}$	$6\lfloor\beta\rfloor/N^{\frac{1}{3}} + (2\beta)^3/r^3$
$\Delta^{-1}\left(f^h + \frac{h^2}{12}\Delta^h(f^h)\right)$	$6/N^{\frac{1}{3}} + 2$	1

Table 4.2: Computational Complexity per finest level grid point for Method of Local Corrections. Refinement ratio r, N grid points in B_R, Γ: number of basis functions for polynomial expansion. qn are the number of points in the Laplacian stencil. * f^h loaded in step 1 gets reused in step 2 and we assume that $\hat{G}$ can fit inside the near-memory working set and does not need to be reloaded across subsequent convolution operations.

4.1 Opportunities for Optimization

Algorithmic Choices

$\{N, \alpha, \beta, q, p\}$ all relate to the error of the scheme as well as how the problem is decomposed. q can be raised for little true computational cost if r is large. Table 4.2 has cubic powers of α and β in many places which can be made smaller at the cost of raising q with higher-order

Operation	Flops/gridpoint	`load`	`store`
relax: 2x $\phi^h_{n+1} = \lambda(f^h - \Delta^h \phi^h_n)$	$20qn$	20	10
restrict	$r^3 + 1$	10	$10/r^3$
prolong	$2r^3 + 1$	$10 + 10/r^3$	1
relax: 2x $\phi^h_{n+1} = \lambda(f^h - \Delta^h \phi^h_n)$	$20qn$	20	10

Table 4.3: Computational Complexity per grid point for 2-level Geometric Multigrid (GMG). Assume proper MG convergence rates and a standard 4 relaxations V-cycle for 10 iterations to have comparable error tolerance as MLC. qn points in Laplacian stencil. Assumes a communication-avoiding implementation with extra ghost cells. The relaxation steps have been fused into a skewed loop (wavefront or diamond). Assumes no agglomeration, so not a true V-cycle, just the finest level solver

due to finite difference localization[52]. N can be made larger or smaller to fit into a fast local memory working set but at the cost of a larger value of k, the patch count. The working set can be made smaller by applying domain decomposition to the Hockney Algorithm itself at the cost of increased computation.

There is the question of what is precomputed and loaded versus computed as the calculation progresses. Table 4.2 made the assumption that $\hat{G}$ is precomputed and loaded for the convolution. In the region $B_{\alpha R}$ this cannot be avoided as computing the discrete Green's function for a high q operator is computationally prohibitive to be done redundantly on every patch. We can, however, exploit the fact that $\hat{G}$ **on** $B_{\alpha R}$ has rotation and mirror symmetries $\hat{G}(x, y, z) = \hat{G}(-x, y, z) = \hat{G}(z, -y, z) = \hat{G}(x, y, -z) = \hat{G}(x, -y, -z) = \hat{G}(-x, -y, z) = \hat{G}(-x, -y, -z) = \hat{G}(-x, y, -z)$ which mean we can just load 1/8th of the total values and perform reversed indexing for the other quadrants. These are then all various flavors of matrix-vector multiply with short and wide matrices. If fast local memory is really limited, there is one more level of symmetry that can be exploited since $\hat{G}(x, y, z) = \hat{G}(y, x, z)$, etc. To exploit this requires an irregular data access technique for packed symmetric blocks similar to dense symmetric matrices.

Outside $B_{\alpha R}$ we can use the linearity of both convolution and the Laplacian operator to precompute $\Delta^H \hat{G} * \mathbb{P}_{\mathcal{L}i}$ for all i basis functions up to the order of our Legendre expansion. For all the even Legendre basis functions the symmetries in $\hat{G}$ are preserved. Unfortunately if any of the tensor product basis functions are odd, this symmetry is broken. Fortunately these precomputed values are only needed on the coarser grid that is r^3 times smaller in data volume for computing F^H.

On the faces of the k patches B_R also requires the interpolation of the interpolation of the convolution. Since convolution and interpolation are both linear operators, this can also be expressed as a short-wide matrix multiplying a tall skinny-matrix.

Computing the Legendre polynomial coefficients can be done with Boole's Rule integration to get sufficient accuracy which is expressed as a stencil computation into an accumulation variable. Additionally since Legendre polynomials can be constructed by a recurrence relation, these loops can be chained together in a wave-front or tiled algorithm.

Algorithm	Flops/Gridpoint	`load`	`store`	AI
GMG	1210	61	21	1.8
MLC	60+4085+55+58=4625	5.375	6.02	50.7

Table 4.4: Multigrid vs MLC. $\alpha = 2.25, \beta = 3.25, N = 33^3, r = 4, qn = 27, \Gamma = 20$. The dominant compute kernel Hockney Transform is shown in red. You have fractional values for MLC as Φ is subsampled on output to adjacent `MLCFaces` and the coarser grid

If we rewrite the information in Table 4.4 in terms of *cycles* for the Edison Cray XC30 HPC platform and assume best-case scenarios for latency, we can get a total cycle time estimate. It is a bit tricky to turn both algorithms into a normalized system based on grid points, but it can be done. The important thing to pick up is the trend. FMM, FFT, and MG all have the pattern of stressing the network latency or DRAM access. Only MLC skews the entire algorithm to the floating-point units.

Algorithm	cycles/Gridpoint:FLOPS	`load`	`store`
GMG	302	1906	900
MLC	2048	291	164

Table 4.5: A restatement of Table 4.4 expressed in common units of *cycles* for the Edison XC30 platform within a single node. FLOPS are turned into cycles using the peak FMA performance. load/stores are turned into a common unit of cycles based on peak DRAM bandwith.

Table 4.5 is an idealized estimate, but the order of the terms is consistent with observations. GMG is essentially limited by data movement as is expected for a stencil operator. The trend for Multigrid is the exact opposite of what we would desire from an energy standpoint. MLC stresses on-core flops. Multigrid stresses on-core loads from DRAM and network latency even at the finest grid resolution.

Chapter 5

Experiments

5.1 Correctness of implementation

These cases were originally defined in [53] for a previous simpler version of MLC published in 2007. The first case is a simple smooth charge case defined on a simple nested hieararchy. The second case considers several localized non-overlapping and highly oscillatory charge sources.

A smooth charge test case

The first test case we are considering involves computing the potential induced by a smooth charge. The charge density is given by:

$$f(\mathbf{x}) = \begin{cases} (r - r^2)^4, & r < 1 \\ 0, & r \geq 1 \end{cases}, \quad r = \frac{1}{R_o}\|\mathbf{x} - \mathbf{x}_o\| \tag{5.1}$$

and the support of the charge is a sphere of radius $R_o = \frac{1}{4}$, centered at point $\mathbf{x}_o = \frac{1}{2}\mathbf{1}$. The computational domain is the unit cube $\Omega = [0,\ 1]^3$. For this problem the exact solution is known and is given by:

$$\phi(\mathbf{x}) = R_o^2 \begin{cases} \frac{r^6}{42} - \frac{r^7}{14} + \frac{r^8}{12} - \frac{2r^9}{45} + \frac{r^{10}}{110} - \frac{1}{1260}, & r < 1 \\ -\frac{1}{2310r}, & r \geq 1 \end{cases}$$

Note that for $r \geq 1$ the exact solution is a pure monopole. We are considering node centered cubic patches of size *33* and a fixed refinement factor of value *4* among all levels.

An oscillatory charge test case

We further consider a case of three oscillatory charges that has been previously studied in [53]. Here we define a local charge density, whose support is a sphere of radius R_o centered at point $\mathbf{x}_o$, by:

$$f_{\mathbf{x}_o}(\mathbf{x}) = \begin{cases} \frac{1}{R_o^3}(r-r^2)^2 \sin^2(\frac{\beta}{2}r), & r < 1 \\ 0, & r \geq 1 \end{cases}, \quad r = \frac{1}{R_o}\|\mathbf{x}-\mathbf{x}_o\|,\ \beta = 4\mu\pi,\ \mu \in \mathbb{N} \tag{5.2}$$

The exact solution associated with this charge density is given by:

$$\phi_{\mathbf{x}_o}(\mathbf{x}) = \frac{1}{R_o} \begin{cases} -\frac{1}{120} - \frac{6}{\beta^4} & , \ r = 0 \\ \frac{r^6}{84} - \frac{r^5}{30} + \frac{r^4}{40} + \frac{60}{\beta^6} - \frac{9}{\beta^4} - \frac{1}{120} + \frac{120}{\beta^6 r} & \\ + \left(-\frac{120}{\beta^6 r} - \frac{9}{\beta^4} + \frac{300}{\beta^6} + \frac{36r}{\beta^4} + \frac{r^2}{2\beta^2} - \frac{30r^2}{\beta^4} - \frac{r^3}{\beta^2} + \frac{r^4}{2\beta^2}\right)\cos(\beta r) & \\ + \left(\frac{12}{\beta^5 r} - \frac{360}{\beta^7 r} - \frac{96}{\beta^5} + \frac{120r}{\beta^5} - \frac{3r}{\beta^3} + \frac{8r^2}{\beta^3} - \frac{5r^3}{\beta^3}\right) sin(\beta r) & , \ r < 1 \\ \left(-\frac{1}{210} - \frac{12}{\beta^4} + \frac{360}{\beta^6}\right)\frac{1}{r} & , \ r \geq 1 \end{cases}$$

and is a pure monopole for $r \geq 1$. For our test case we consider three charges of the form (5.2), with radius $R_o = \frac{5}{100}$, centered at points $\mathbf{c}_1 = \left(\frac{3}{16}, \frac{7}{16}, \frac{13}{16}\right)$, $\mathbf{c}_2 = \left(\frac{7}{16}, \frac{13}{16}, \frac{3}{16}\right)$ and $\mathbf{c}_3 = \left(\frac{13}{16}, \frac{3}{16}, \frac{7}{16}\right)$. The computational domain is the unit cube $\Omega = [0,\ 1]^3$ and the total charge and total potential are given via linear superposition by:

$$\begin{aligned} f(\mathbf{x}) &= f_{\mathbf{c}_1}(\mathbf{x}) + f_{\mathbf{c}_2}(\mathbf{x}) + f_{\mathbf{c}_3}(\mathbf{x}) \\ \phi(\mathbf{x}) &= \phi_{\mathbf{c}_1}(\mathbf{x}) + \phi_{\mathbf{c}_2}(\mathbf{x}) + \phi_{\mathbf{c}_3}(\mathbf{x}) \end{aligned}$$

It is noted that the layouts we are using are not the same with those in [53] leading to different local projection errors and hence slightly different results.

Accuracy Results

Detailed and exhaustive convergence tests for this algorithm are beyond the scope of this thesis write-up but the first comprehensive error analysis has been performed in [47]. The algorithm and implementation hve been altered since then so it is good to verify that the accuracy has not been altered.

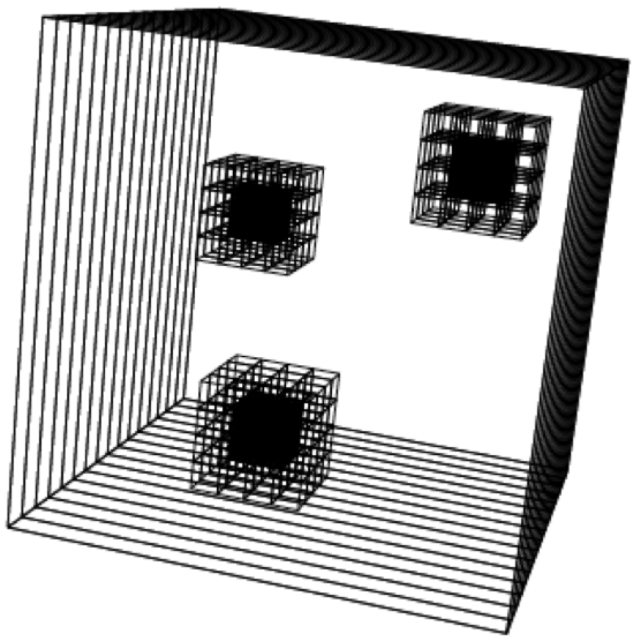
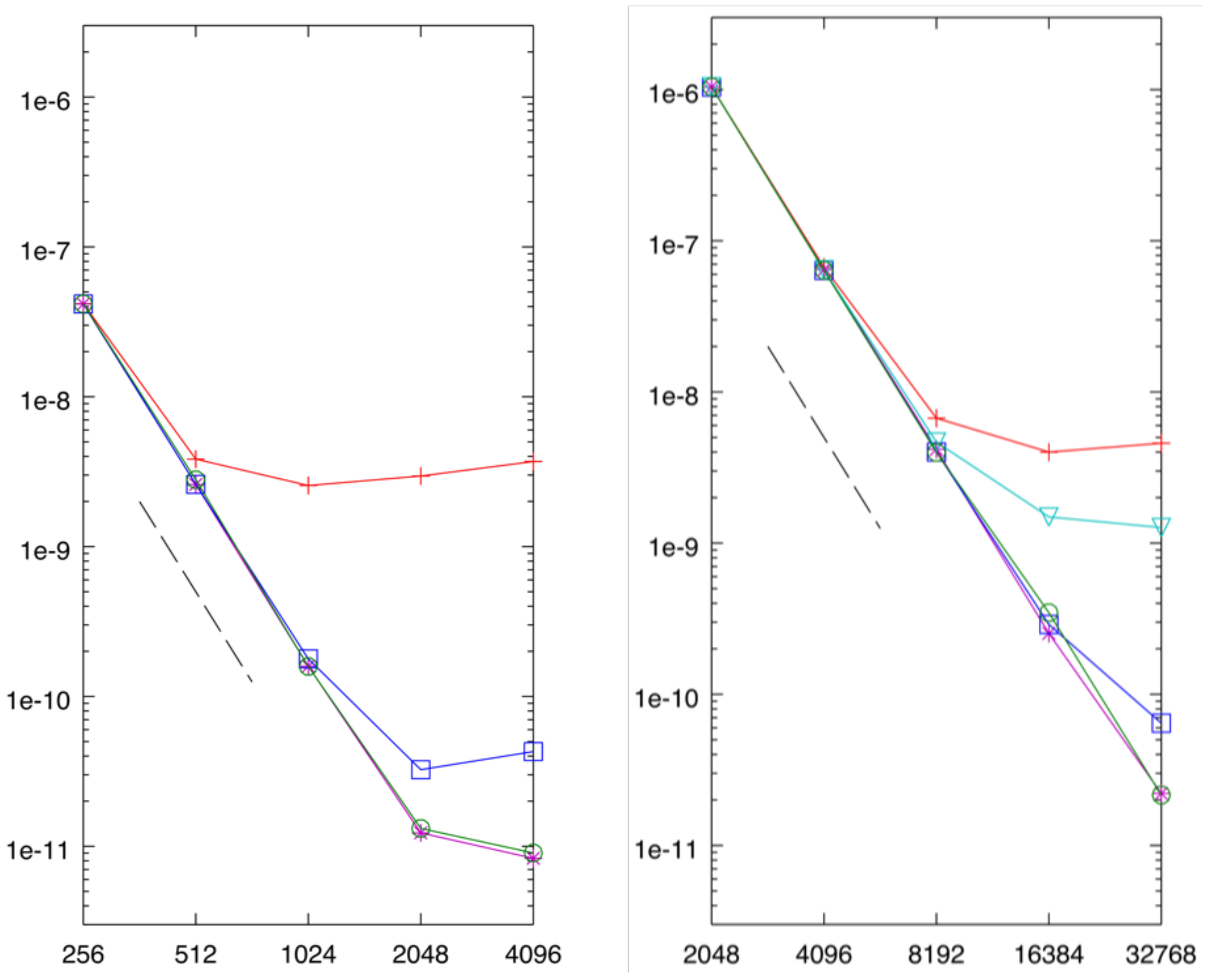


Figure 5.1: Adaptive mesh hierarchy for oscillatory problem. 3 level of refinement. Factor of 4 refinement ratio. Each outlined patch is a 33x33x33 set of structured grid points. The base grid dimensions are varied through the course of a convergence study while the grid geometries and ratios are preserved.

Experimental Platform

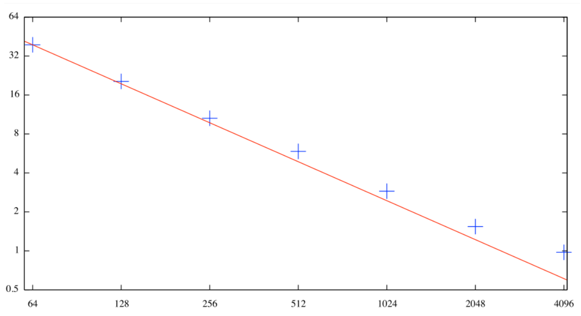
These experiments were performed using the Cori supercomputer and the National Energy Research Scientific Computing Center (NERSC). The analysis and experimental results have been focused on the Haswell partition. This is a Cray XC40 supercomputer.

- icpc (ICC) 18.0.1 20171018 Copyright (C) 1985-2017 Intel®Corporation.
- Cray-tuned FFTW3 libraries Version 3.3.6.3
- Each node has
  - Two sockets, each socket is populated with a 16-core Intel®Xeon^{TM} Processor E5-2698 v3 ("Haswell") at 2.3 GHz
  - 128 GB DDR4 2133 MHz memory (four 16 GB DIMMs per socket)
  - 2 x 40-MB shared L3 cache
- Each core has
  - 32KB L1 data cache

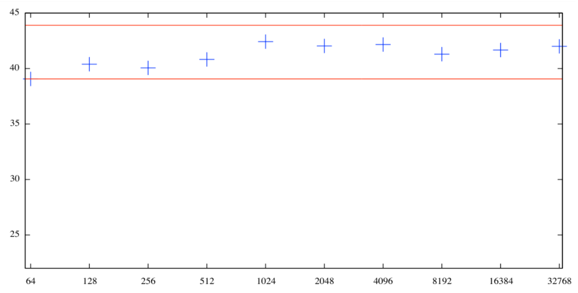


(a) Uniform charge problem convergence results (b) Oscillatory adaptive grid convergence results

Figure 5.2: $+$ 27-pt operator $N = 32^3, \alpha = 2.25, \beta = 3.25$, $\bigtriangledown$ 27-pt operator $N = 64^3, \alpha = 1.625, \beta = 2.125$, $\square$ 27-pt operator $N = 64^3, \alpha = 2.25, \beta = 3.25$, $*$ 117-pt operator $N = 32^3, \alpha = 2.25, \beta = 3.25$, $\bigcirc$ 117-pt operator $N = 64^3, \alpha = 2.25, \beta = 3.25$. X-axis is $\frac{1}{h}$ of the base grid. Max norm error. 4th-order slope drawn as dashed line.



(a) Strong scaling. 10^9 unknowns.



(b) Weak scaling. 10^9 unknowns base case. 5.1x10^{11} unknowns at 32K cores and effective uniform grid resolution of $64K^3$=2.8x10^{14} unkowns.

Figure 5.3: Scaling results for Cori Haswell XC40 supercomputer. N=33^3, 27-point operator, 3rd-order Legendre Polynomial expansions, α=2.25, β=3.25. Vertical axis is time, in seconds. Horizontal axis is total number of cores.

  - 256 KB L2 unified cache

- Cray®Aries Interconnect with Dragonfly topology and 5.625 TB/s global bandwidth

Scaling Performance

For an HPC implementation of a Poisson solver effective utilization of distributed memory is as important as maximizing utilization of the target CPU architecture. As discussed in Chapter 2, the algorithm was implemented as a hybrid MPI+OpenMP parallel algorithm. It is necessary to show that this approach has been implemented in a scalable fashion. Figure 5.3 shows the performance for the case of the N=33^3, 27-point operator, 3rd-order Legendre Polynomial expansions, α=2.25, β=3.25. While there are a dozen other configurations of discrete operators and barrier error targets, all these variations have little impact on the quality of these scaling results. The largest effect on both strong and weak scaling is increasing α or β. The runtime is completely unaffected by the choice of a 27-point operator, the 117-point operator, or the application of Mehrstellen correction.

As a point of comparison we also benchmark the HPGMG benchmark [1] for a comparable configuration. Since HPGMG does not have an adaptive-refinement implementation we compare problems with similar number of degrees-of-freedom. For 1E9 grid points and boxes of dimension N=32^3 at 256 cores HPGMG solves to a comparable truncation error in 9.0 seconds. MLC solves this same scale of problem in 9.9 seconds. The effective grid resolution for the MLC solver is 8 times finer in each dimension.

```
downwardPass 12.63
   setPhiDeltaAndRho 11.94
      HockneyFFTWserial::solve 11.35
      subtractInterpolantLocal 0.22
      MLCFaces::incrementFaces 0.05
      projectToCoarse on FArrayBox 0.04
      evalOperator on phiCoarseFab 0.01
   setRhsModCoarser 0.40
      generalCopyTo on a_rho 0.27
         MPI_Waitall 0.27
      generalCopyTo on rhsSampled 0.10
         MPI_Waitall 0.10
      generalCopyTo on m_legendreCoeffsSampled 0.017
         MPI_Waitall 0.017
upwardPass 3.20
    addPolyConvolutionFineInterpDiffMLCFacesVect 2.45
    PoissonDirichletFFTW::solveInhomogeneousBCInPlace 0.32
    m_phiDeltaMLCFaces.exchange() 0.21 generalCopyTo on
    m_legendreCoeffsAlphaBar 0.11
```

Table 5.1: Extracted performance call-graph for Variant 1. Flat MPI execution

Performance Breakdown for Hockney Variant 1

This version of the algorithm would be considered the base implementation case. This baseline already includes many optimizations for an efficient implementation. For instance messages between ranks are aggregated during communication phases and local data manipulation is overlapped with communication. It also uses manually `pruned` DFT transforms as discussed in Chapter 4, exploits the symmetries of the operations, and the real-valued properties of the inputs, outputs, and $\hat{G}$.

Performance Breakdown for Hockney Variant 2

In code variant 2, threads are introducted as computational peers to ranks. At the instruction, execution, and scheduling level, this is the same as the variant 1 computation. The main benefits are realized through utilization of shared memory. Thread peers that share an address space can perform communication between SPMD domains with load store instructions and bypass the networking layer. This shared memory bypass can then be overlapped with the distributed communication messaging phase where modern interconnects like Aries can perform true messaging offloading. Thread peers can also share a single copy of the pre computed data structures. This includes $\hat{G}$ and the pre-convolved Legendre expansions. Each thread execution is manipulating its independent f to ϕ calculations. While the domain is naturally decomposed into patches, at this level of granularity each thread's individual working set combined easily exceeds the L3 cache capacity for this architecture. It is also noted at this variant that it is necessary to restrict the hybrid parallel execution to each rank occupying a unified memory accessible domain. When threads cross NUMA boundaries, performance degrades below that of a flat MPI implementation.

The effect of sharing $\hat{G}$ had little impact on the time to solution. Variant 2 does not have a performance breakdown table presented as it is the same as variant 1. The main victory with coarse-grained parallelism was the fact that the code could execute across the entire scaling ranges needed for this study. The flat MPI code will often crash the system with Out-Of-Memory errors. Partly this is from sharing meta data like Legendre expansions and frequency-space Green's function $\hat{G}$. There is less asynchronous buffering needed in Chombo, more data is transmitted via shared memory`load/store` operation, but there are buffers that Cray MPI maintains in the background that consumes roughly 200MB per MPI rank. We do control the degree of asynchronous buffering with the MPICH_GNI_NDREG_MAXSIZE environment variable (the amount of outstanding synchronous communication that can be outstanding before MPI switches to a rendezvous protocol). Experiments with smaller buffer sizes degrades communication speed.

Measurements with the Intel VTune tools indicate a large amount of DRAM traffic in this variant. To see the benefits of a memory caching subsystem, a finer level of decomposition is necessary.

Performance Breakdown for Hockney Variant 3

In Variant 3, threads collaboratively execute the kernels in the downward-pass of the algorithm. The hypothesis is that on a per-rank-basis each kernel would have access to a larger effective memory cache. In addition, as the threads execute different directions of the FFT algorithm they will encounter memory previously loaded into memory by another thread during a previous phase of the computation (ie. temporal locality). A filtered output from the instrumented code is shown in table 5.2.

Performance Breakdown for Hockney Variant 4

In Hockney variant 4 we fuse the forward k-direction FFT transform with the multiplication by the symbol $\hat{G}$ and the inverse FFT transform. This operation is done using a thread-private buffer called `pencilBox` which is a tiling in the i-j plane and the full transform size in the k-direction. The gains here are impressive. It is now instructive to assess the performance of the Hockney kernel with respect to a performance model to assess how effective the implementation has become. The next section is a presentation of an augmented Roofline Model [67] for the convolution operation.

```
downwardPass 6.54
  setPhiDeltaAndRho 6.09
      HockneyCompact::transform 5.49
      MLCFaces::setFaces 0.09
      projectToCoarseT on NodeFArrayBox 0.07
      evalOperator on phiCoarseFab T 0.06
      setLegendreCoeffs 0.25
   setRhsModCoarser 0.19
      generalCopyTo on rhsSampled 0.07
         MPI_Waitall 0.06
      addPolyConvolutionCoarseLaplacianVect 0.06
      generalCopyTo on a_rho 0.028
      generalCopyTo on m_legendreCoeffsSampled 0.02
         MPI_Waitall 0.02
upwardPass 3.09
      addPolyConvolutionFineInterpDiffMLCFacesVect 1.21
      PoissonDirichletFFTW::solveInhomogeneousBCInPlace 0.31
      m_phiDeltaMLCFaces.exchange() 1.05
      generalCopyTo on m_legendreCoeffsAlphaBar 0.12
```

Table 5.2: Performance breakdown for Variant 3 of MLC

Arithmetic Intensity Analysis for Modified Hockney's Algorithm

The classic Roofline model is meant for streaming computation kernels. A key factor in utilizing a streaming model is to determine what level of the memory hierarchy your kernel exhibits streaming behavior.

Using the Experimental Roofline Toolkit (ERT) [50] you can measure the effective Roofline performance of the target architecture. You can refer back to figure 1.2 for the Roofline plot for this architecture. The different bandwidth lines are generated based on different *working set* sizes: How much memory do you access before touching a part of the memory again and getting temporal data re-use.

Under the hood, ERT can also show a user more detailed performance characteristics. Figures 5.4 and 5.5 show the effect of varying how parallelism is expressed. The ERT code is embarrassingly parallel, with each thread executing on its own working set.

The X-axis shows the working set size and the Y-axis shows the available bandwidth that can be achieved. For very small working sets that easily fit into L1 the performance is actually limited by factors that are not tied to load/store performance. Once the working set reaches 10KB you see the effective L1 bandwidth. For the higher number of OpenMP threads you can see that the effective cache capacity tails off well before you reach the total combined L1 cache capacity (500KB out of a total of 1024KB) the performance transitions

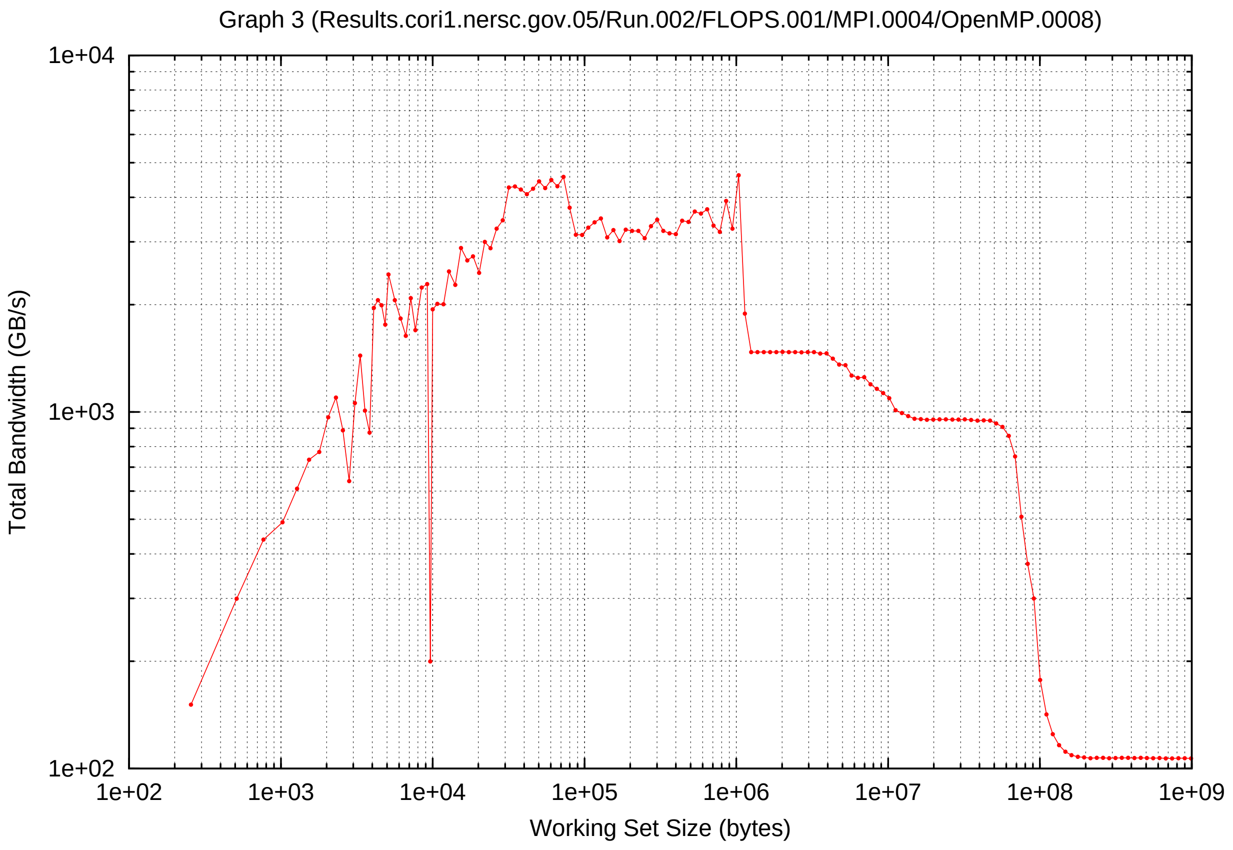


Figure 5.4: Experimentally-derived bandwidth measurements of 1 node (2 sockets, 32 cores) experimental platform for 2 FLOP computational kernel using 4 MPI ranks, each with 8 OpenMP threads

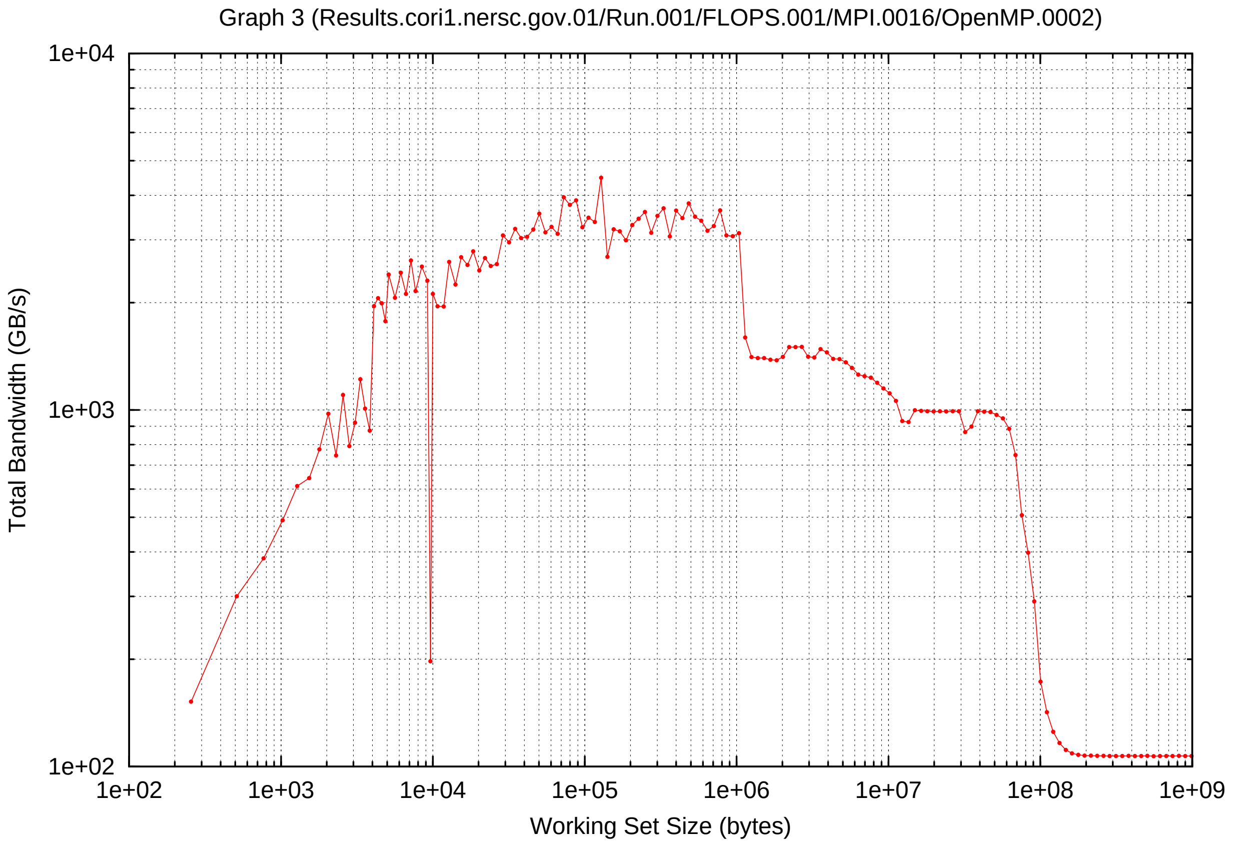


Figure 5.5: Experimentally-derived bandwidth measurements of 1 node (2 sockets, 32 cores) experimental platform for 2 FLOP computational kernel using 16 MPI ranks, each with 2 OpenMP threads

to the speed of the L2 cache. The 1024KB L1 capacity is more clearly visible in the flatter MPI execution model. The effective L2 capacity and bandwidth are similarly impacted by the interference from the threading behavior. What is the most similar between all the execution models is the L2 and L3 cache capacity (the place where each performance plateau rolls off). The processor is spec'd from the manufacturer to have 8MB L2 cache and both sweep plots show a roll off in this area. Both sweeps also have an obvious roll off to DRAM bandwidth just before the capacity of the 80MB L3 shared cache.

Critical observations here are that Roofline is not a good performance model in regimes where data bandwidth effects are not dominant, like in the build-up phase with very small working sets. There is a terrible penalty for falling out of the L3 data cache working set size.

When analyzing the Hockney convolution we will track the effective working set size as well as load/store traffic and floating-point operations. This worksheet is presented in table 5.6.

Phase	flops	ld/st (MB)	working set (MB)
33x33 1D FFT r2c	$\frac{1}{2}$x33x33xT	33^3x8=.27	$(33^2$x 130x8=1.0$)_2$
33x66 1D FFT c2c	33x66xT		$($33x66x130x2x8=4.3$)_2$
$\hat{f}$ =**66x130 1D FFT c2c**	66x130xT		$($66x130x130x2x8=17$)_3$
$\hat{f} *= \hat{G}$	66x130x130x2	$(\frac{130^3}{8}$x8=2.1$)_1$	**4x8x130x2x8=.06**
66x130 1D IFFT c2c	66x130xT		
66x97 1D IFFT c2c	66x97xT		66x97x130x2x8=12.7
97x97 1D IFFT c2r	$\frac{1}{2}$x97x97xT		$($97x97x130x8=9.3$)_2$
$B^{\alpha R}$ = 97^3x8=7			
$\sum_i \Phi_c$		$\frac{2}{64}B^{\alpha R}$	
$\sum_i \Phi_f(faces)$		$\frac{22}{97}B^{\alpha R}$	
B=load+store			
T=5x130x$\log_2$(130)	F=**143.7e6**	B=**2.21e6**	ws=**12.8MB**

Figure 5.6: Hockney Convolution Worksheet DRAM Arithmetic Intensity AI=$\frac{F}{B}$=**61**: 27-point Laplacian operator, zero structure exploited, steps 3-5 (k-pencils) operations fused, n_s=33, n_d=97. 1. Assumes that a single MPI rank will perform many Hockney convolutions and the load of $\hat{G}$ is ammortized away. 2. These operations share a common work array with inverse J-transform and so do not count towards the total working set. 3. This would be the peak working set size if we did not block and fuse the k-direction operations.

Table 5.6 shows that for one Hockney convolution we need to manipulate a 12.8MB working set. Given an 80MB L3 cache capacity it would suggest that we can work on 4 convolutions concurrently and avoid DRAM access. This a bit optimistic since we elided the copy of $\hat{G}$ that all ranks are accessing. Since $\hat{G}$ is a read-only data structure and will be stored in L3 is a *shared* state of the Modified-Exclusive-Shared-Invalid-Forward MESIF cacheline finite-state machine policy used in Intel Xeon *ccNUMA* [40, 54, 44]. We will end

```
downwardPass 4.92640
  setPhiDeltaAndRho 4.20626
      HockneyCompact::transform 3.63739
      InterpOnFace::interpolate 0.32554
      MLCFaces::setFaces 0.08553
      setLegendreCoeffs 0.22360
      projectToCoarseT on NodeFArrayBox 0.06523
      evalOperator on phiCoarseFab T 0.05755
   setRhsModCoarser 0.49412
      generalCopyTo  on a_rho 0.24737
        MPI_Waitall 0.22376
      generalCopyTo on m_legendreCoeffsSampled 0.10679
        MPI_Waitall 0.10235
      generalCopyTo on rhsSampled 0.06421
        MPI_Waitall 0.05899
      addPolyConvolutionCoarseLaplacianVect 0.06356
 upwardPass 3.02574
      addPolyConvolutionFineInterpDiffMLCFacesVect 1.25917
      PoissonDirichletFFTW::solveInhomogeneousBCInPlace 0.31140
      m_phiDeltaMLCFaces.exchange() 0.97926
      generalCopyTo on m_legendreCoeffsAlphaBar 0.13527
```

Table 5.3: Performance breakdown for Variant 4 of MLC

up with two copies of $\hat{G}$, 2.1MB resident on each socket. It does suggest the optimal level of shared memory parallelism is 4 simultaneous transforms can be supported by this platform. Variant 4 was run using 4 MPI ranks per node and 8 threads collectively performing the Hockney transform. The results of this experiment are shown in table 5.3. We refer to this variant as the *fine compact* variant where the k-direction transforms have been fused with multiplication against $\hat{G}$ in a thread-local buffer.

For comparison to geometric multigrid we set up the HPGMG HPC benchmark [1] to run the same number of degrees of freedom problem.

```
cc  -Ofast -xAVX2 -fopenmp level.c operators.27pt.c \
    mg.c solvers.c hpgmg-fv.c timers.c -DUSE_MPI -DUSE_SUBCOMM  \
   -DUSE_VCYCLES -DUSE_GSRB -DUSE_BICGSTAB -o run.haswell
srun --ntasks=16 --nodes=8 --ntasks-per-node=2 --cpus-per-task=16 \
--cpu_bind=verbose,threads ../run.haswell 5 2048
```

This solver converges in 9 iterations. The performance breakdown is shown in table 5.4

We can note that the solver is competitive even in a uniform-grid head-to-head comparison against what is considered a well-written geometric multigrid solver.

```
===== Timing Breakdown ===============================
level                                  0            1
level dimension                   1024^3        512^3
box dimension                       32^3         16^3          total
------------------          ------------ ------------ -- ------------
smooth                          4.383103     0.667028       5.176560
residual                        0.824394     0.064869       0.899360
applyOp                         0.000000     0.000000       0.000011
BLAS1                           0.189618     0.031581       0.226203
Boundary Conditions             0.116326     0.027812       0.157308
Restriction                     0.123759     0.020223       0.147309
  local restriction             0.123739     0.020208       0.146710
  unpack MPI buffers            0.000000     0.000000       0.000068
  MPI_Waitall                   0.000000     0.000000       0.000383
Interpolation                   0.216392     0.030635       0.251998
  local interpolation           0.216376     0.030622       0.251383
  MPI_Isend                     0.000000     0.000000       0.000157
  MPI_Waitall                   0.000000     0.000000       0.000243
Ghost Zone Exchange             1.639090     0.468217       2.312578
  local exchange                1.312621     0.348386       1.784754
  pack MPI buffers              0.054460     0.017689       0.084142
  unpack MPI buffers            0.064791     0.018500       0.091890
  MPI_Isend                     0.002670     0.002578       0.010909
  MPI_Irecv                     0.000642     0.000648       0.002551
  MPI_Waitall                   0.203557     0.080027       0.336246
MPI_collectives                 0.028289     0.000000       0.028338
------------------          ------------ ------------ -- ------------
Total by level                  7.507876     1.252272       9.108185

   Total time in MGBuild        0.864459 seconds
   Total time in MGSolve        9.177287 seconds
```

Table 5.4: HPGMG Performance on uniform grid with domain decomposition and hybrid MPI+OpenMP execution

The Hockney kernel achieves a final aggregate performance of 155 GFLOPS on the Cori Haswell compute node. This can be compared with the peak experimentally achieved performance on this node of 844 GFLOPS from the ERT tool. This requires some explanation, as looking at the AI worksheet for this kernel we have an AI of 10.7. At an AI of 10 we should witness full processing capacity.

First we can configure ERT to run a kernel that has an arithmetic intensity closer to our kernel, the results are shown in figure 5.7. At this level of compute intensity you can see that all the levels of cache disappear as the processor can amortize cache latency until DRAM access. The other thing to note is that at this higher FLOP rate the peak acheivable performance has dropped slightly. With some simple algebra you end up with a peak performance of 700 GFLOPS. So, higher arithmetic kernels can fall away from the global maximum.

Next, the ERT kernel is specifically written to utilize Fused-Multiply-Add (FMA) instructions on every clock cycle. While HPGMG `relax, restrict, prolong` all are dominated by FMA, the FFT codelets are not. In pure Cooley-Tukey FFT every stage is expressed as FMA, many of these operations are done against trivial twiddle factors. FFTW uses symbolic processing in OCAML to group common subexpressions and eliminate trivial multiplications. For the 130-sized transform FFTW selects a 10-by-13 split-radix implementation. Each codelet variant is chosen by an autotuning preprocessing step. In this case our transforms make use of 4 FFTW codelets

- `n1bv_13` no-twiddle, variant 1, backward transform, size 13, vectorized 31 additions, 6 multiplications, 57 fused multiply/add
- `t1fuv_10` twiddle, unpacked, forward transform, size 10, vectorized 33 additions, 22 multiplications, 18 fused multiply/add
- `t2bv_10` twiddle form 2, backward vectorized 33 additions, 22 multiplications, 18 fused multiply/add
- `n1fv_13` no twiddle forward transform vectorized 31 additions, 6 multiplications, 57 fused multiply/add

Using these figures, and profiling for the exact number of times each of these codelets is called, we can compute that only 36.8% of the floating-point instructions being executed are FMA. The remaining floating-point operations are multiplication and addition. For highly-tuned FFT transform kernels the realistic peak performance based on optimal instruction issue would be 485 GFLOPS.

Using the Intel VTune tools it is verified that the kernel does indeed hold the working set entirely in L3 cache. Data from the hardware counters in a typical run produce some more interesting results:

```
CYCLE_ACTIVITY.CYCLES_NO_EXECUTE                40670061005
```

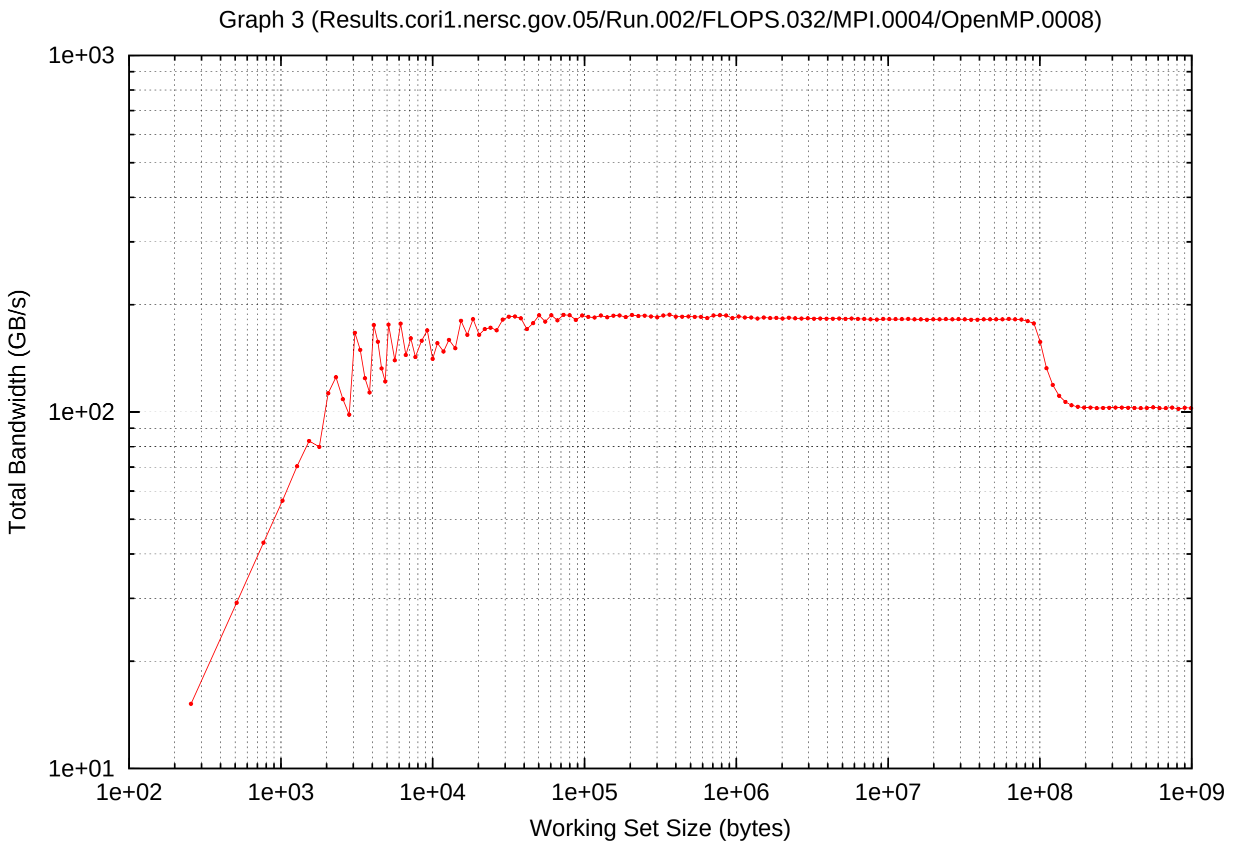


Figure 5.7: Experimentally-derived bandwidth measurements of 1 node (2 sockets, 32 cores) experimental platform for 32 FLOP computational kernel using 4 MPI ranks, each with 8 OpenMP threads

```
RESOURCE_STALLS.SB                              25298037947
MEM_LOAD_UOPS_L3_MISS_RETIRED.LOCAL_DRAM_PS        61601848
MEM_LOAD_UOPS_L3_MISS_RETIRED.REMOTE_DRAM_PS              0
MEM_LOAD_UOPS_L3_MISS_RETIRED.REMOTE_HITM_PS              0
```

As intended, there are no DRAM accesses to a remote DRAM address. The code is not crossing Unified Memory domains. The number of L3 misses is a minute fraction of the total micro-ops the processor is executing. The dominant factor in processor stalls is not bandwidth itself, but an exhaustion of the *Store Buffer*. The Haswell processor has 72 entries in its Load Buffer, and 42 entries in its Store Buffer. So, more than half of the processor stalls are the result of waiting on space in the store buffer. Running the Hockney kernel through the Intel Software Development Emulator produces a clearer picture of how this behavior manifests (shown in table 5.5). First, we can verify that the majority of the floating-point operations are indeed done as 4-way SIMD instructions, but a non-trivial number of 2-way SIMD instructions are part of the mix. The Haswell processor does not have an 8-way SIMD instruction.

elements-fp-double-1	**223500125**
elements-fp-double-2	**4626104600**
elements-fp-double-4	**17559860530**
elements-fp-double-8	**0**
Total single-precision FLOPs	**0**
Total double-precision FLOPs	**79715151445**
mem-read-1	**39409184**
mem-read-2	**10812**
mem-read-4	**402017620**
mem-read-8	**8963523874**
mem-read-16	**5788181617**
mem-read-32	**6580397550**
mem-write-1	**2199321767**
mem-write-2	**4368**
mem-write-4	**69478325**
mem-write-8	**3254986928**
mem-write-16	**4540548184**
mem-write-32	**1453802572**

Table 5.5: SDE Output from Variant 4 Kernel Benchmark

So, the first thing to note is that the generated FFTW codelets still make use of some scalar operations and short vector operations. 28% of the floating-point operations are done on smaller units. This makes sense as our codelets and our FFT batch operations are not nice multiples of the built-in vector length.

What is *not* well vectorized in FFTW are the load and store operations. Essentially, the store buffer is overwhelmed with small read and write instructions. This is hard to avoid in FFT, as `Complex double` is not a native type in modern processors. While the loads are moderately well vectorized, the stores are overwhelmingly a single complex-valued double (mem-write-16) or single double (mem-write-8). Native complex values in C99 are interleaved. Since most of the writes here are subject to later reads in the next phase of the kernel there is little to be gained by using write-through techniques. There are several instructions needed to load and store complex values into vector registers to make them suitable for AVX instructions. Most of the time the processor spends stalled is due to exhaustion of the store buffer.

We attempted to remedy this poor load/store vectorization by switching to a non-interleaved representation of complex-valued data. The is referred to in the FFTW manual as *split layout* where the real and the imaginary values remain in their own disjoint data holders. This experiment did not result in any performance gains. With a closer look it turns out that FFTW passes the split representation through a data transform phase to land in the expected interleaved codelets. To take this approach further would require dismantling the symbolic engine and back-end code generation of FFTW to respect a split layout end-to-end. That is left for future work.

With those major factors now accounted for this implementation is as close to optimal as we can likely attain without a significant rewrite of the code generation for FFT. It is also clear that the Hockney kernel itself is not the only target for further optimization of the overall MLC algorithm.

Another important observation from both HPGMG and MLC is the communication costs across the disctributed memory system. Our models from Chapter 4 suggested that there should be a clear advantage to the non-iterative approach for minimizing time-to-solution by reducing the amount of communication, but looking at tables 5.3 and 5.4 we can see that *neither* application is really limited by the communication library calls. MLC spends 0.35 seconds in communication, and HPGMG spends 0.45 seconds in the MPI operations. What both algorithms suffer from is the work the CPU and local memory system must do handle what we refer to as the *data choreography* involved in the distributed memory aspects of the algorithm. Both algorithms utilize space-filling curve methods to minimize surface-to-volume effects and thus maximize on-node memory movement and minimize off-node data traffic. Inter-node communication stress the ability for the CPU to marshal data movement on `Box` surfaces. This ends up stressing the processor's ability to use `mem-write-8` instructions. ie. the compiled codelets make poor use of the vectorized load/store instructions. This exhaustion of the store buffer effect ends up dominating the communication phases.

Chapter 6

Conclusions

The Roofline model predicts that MLC can be an effective solver for Poisson's equation for emerging computer architectures. The reduced memory access gains can be realized at an acceptable cost in additional floating-point operations. In this work, we implemented several progressively improved variants of the MLC algorithm guided by the Roofline performance model focusing on the Hockney convolution and the Intel®Haswell processor.

Key factors in moving the algorithm upwards in the Roofline plot was managing the working set size and eliminating OpenMP and MPI overheads. The instruction-level parallelism and vector-level parallelism were provided by the code generation technology in the FFTW package. This was verified through hardware performance measurements.

For the instruction mix in a modern FFT algorithm, the best we could hope to achieve on the target platform is 485 GFLOPS. And yet the arithmetic intensity calculation for the Hockney convolution, 10.7, should be able to achieve peak performance, and yet we stop at 155 GFLOPS. While this surpasses all of the FFT-based HPC algorithms currently running at the NERSC supercomputer center, it still requires an explanation. This is where the limitations of the simplified Roofline performance model need to be investigated. If you recall, the Roofline performance model assumes a simple processing unit and memory hierarchy. The critical architectural feature not captured in the Roofline performance model was the exhaustion of the store buffer. While FFTW code generation makes adequate use of vectorized load instructions, it make significantly poor use of vectorized store instructions. Overall there are an equivalent number of loads and stores in FFT. Perversely the Intel®Haswell processor is biased with more load buffer capacity (72 slots) than store buffer capacity (42 slots). In the end, 62% of the processor stall cycles are caused by an exhaustion of available slots in the store buffer. This is not a fundamental limitation of FFT but a deficiency in the current code generation technology as discussed below.

In a similar way, the ability of the processor to maximize the potential of the inter-node communication library is limited by the processor's ability to marshal data movement effectively between distributed memory systems. A similar exhaustion of the load and store buffers prevent the CPU from realizing the benefits of our existing fast interconnect hardware.

Future Work

We have only just begun to map out the performance space for the MLC algorithm. The error estimate for MLC given in(2.16a)-(2.16d) suggests that raising the order of the discrete operator can allow us to reduce the necessary overlap regions like α for the same effective accuracy. The dominant computational kernel (Hockney) is already a complete discrete convolution and the computational costs are fixed. $\{N, \alpha, \beta, q_\alpha, q_p\}$ can be tuned for the architecture and accuracy and this space requires greater exploration. Preliminary work using a 10th-order 117-point operator look promising.

The *split* representation of FFTW *should* have allowed the codelets to use fully-vectorized load/store instructions and achieved full peak performance, but the current design is biased to the *interleaved* representation. *split* inputs are still manipulated into interleaved codelets in the interest of simplified software design. Perhaps there is an assumption that since C/C++/Fortran have adopted `complex` as a fundamental data type that hardware will follow suit and vendors will provide vectorized complex operation. We do not think this will be happening. For FFT applications that target convolutions the split representation is more natural and effort needs to be expended to make this path through the code generation mechanism preserve the vector-friendly data layout that the applications are already using.

For applications that operate on complex-valued inputs/outputs it is possible to retrofit the FFTW package itself to perform a minor increase of number of register-to-register operations to assemble larger contiguous memory regions for which vectorized store operations can be performed. This would relieve the burden on the over-taxed store buffer. The Xeon family of processors can do this operation effectively since we have not exhausted the register file or the reorder buffer. “Minor” in this case is actually an understatement. The change would have to be effected not in the simple vectorizing macros for FFTW but in the symbolic engine written in the OCaml language[49] that underlies FFTW `genfft`, as these operations would need to be scheduled correctly within the superscaler optimization engine that is also inside of the symbolic processor. *Every* code variant that the FFTW code generator currently produces suffers from this deficiency. Therefore, there is no way to auto-tune our way out of this problem. Furthermore, this will not be the last critical architectural feature that will need to be factored into the code generation technology in the coming years. We are seeing an increase in diversity in the HPC hardware architecture space, and FFT based kernels are only going to become more prevalent. As a near-term example of this problem, a GPU doesn’t use a store buffer as the Xeon does. It has a completely different memory technology called *memory coalescing* which is not optimized by vectorizing your store operations. FFTW itself no longer has its original development team involved in the project. It receives minor updates but nothing on the scale that is required.

For the communication-phase of MLC (and GMG, and likely all HPC motifs) there is the perverse and pervasive problem of asking a modern vectorized microprocessor to manage the data choreography for inter-node communication. One approach being explored just this year is to move to units of data communication that *are* vectorized by a code generator [69], where halo cells are represented as lists of *bricks* that are naturally sized to fit the native hardware cacheline size. As outlined in *Attack of the Killer Microsecond*[13], there is an entirely different argument that asks "How to Waste a Fast Datacenter Processor?". We need to stop using general-purpose processor like a Xeon$^{\text{TM}}$ to manage these kinds of operations and instead invoke the next-generation *Remote Direct Memory Access* (RDMA) offload capabilities of modern interconnects and assign low-power ASICS to offload these transfer operations. It has been the case that the host CPU is the rate-limiting factor in HPC inter-node communication for some time already[59]. UPC++ [70, 7, 6] is the kind of library where asynchronous strided `rput/rget` would disintermediate the CPU from these data transfer operations.

The bigger picture: We should not expect HPC developers to undertake the protocols demonstrated in this thesis for each application domain and discipline that use FFT-based kernels. While the store buffer limitation can be overcome within FFTW, we would not have hit this performance wall without the manipulations we made to schedule and fuse the entire convolution. A new spectral code generation infrastructure to handle FFT-like transforms with plans that span different sizes, ranks, pruning, and convolution kernels is needed. In general, FFT and BLAS needed a unified code generation mechanism to achieve the kind of performance people are expecting from future architectures. These generated plans need to also schedule threading and communication phases.

Appendix A

Notation

For reference in this thesis this is a handy table for notation

$\mathbb{Z}^D$	Integer space of dimension D
$\mathbb{R}^D$	Real space over dimension D
B_R	Ball of radius R in the space $\mathbb{Z}^D$
$B_{\alpha R}$	Ball of radius αR for some $\alpha > 1$
$B_{\beta R}$	Ball or radius βR for some $\beta > \alpha$
L	Linear Operator. In this document L will always be the Laplacian
Δ	Continuous Laplacian Operator
Φ	Scalar value field defined over space $\mathbb{R}^D$
i, j	Points in $\mathbb{Z}^D$
$\boldsymbol{x}, \boldsymbol{y}$	Real vector in $\mathbb{R}^D$ used to represent a position
$\boldsymbol{z}$	displacement in $\mathbb{R}^D$ away from B_R
$*$	Convolution operator, both in continuous and discrete space
$\boldsymbol{w}$	Mixed partial derivative order. Vector valued
f	Scalar field defined $\mathbb{R}^D$, used to represent the input charge distribution
O	Order of limiting behavior
q	degree of first truncation error term
q_P	degree of Legendre Polynomial
$q_{\mathcal{I}}$	degree of coarse-fine Interpolation
q_α	degree of first trunction error term of discrete Laplacian in α region
h	Two uses: 1. mesh spacing on the fine grid in $\mathbb{R}$
	2. operations/fields that are referenced/sampled to the fine grid spacing
H	Two uses: 1. mesh spacing on the coarse grid in $\mathbb{R}$
	2. operations/fields that are referenced/sampled to the coarse grid spacing
r	Integer refinement ratio between grid levels
$\mathcal{I}$	reference to interpolation operations
G	Green's Function
$\mathcal{F}$	Fourier Transform

G^h	Discrete Green's Function of the Laplacian operator on the fine grid
G^H	Discrete Green's Function of the Laplacian operator in the coarse grid
N	Number of grid points
$\hat{T}$	The Fourier Transform of T. It is used on $\hat{f}$, $\hat{f}^h$, $\hat{G}^h$, etc.
	LaTeX hat means Fourier Transform.
$\mathbb{P}_{\mathcal{L}}$	Projection onto the Lengendre Polynomial Basis functions
$P_{m,n}$	The n'th Lengendre basis function of order m
$a_{m,n}$	The coefficient of P_{m_n}
Γ	number of coefficients in $\mathbb{P}_{\mathcal{L}} f^h$
qn	number of points in a qth order stencil
p	Processors

Table A.1: Notation

Appendix B

L^h_{19} and L^h_{27} Mehrstellen Discretizations of the Laplacian

The stencil coefficients for the L^h_{19} and L^h_{27} Mehrstellen Laplacians are $a_{\boldsymbol{j}} = \frac{1}{h^2} b_{|\boldsymbol{j}|}$, where $|\boldsymbol{j}|$ is the number of non-zero components of $\boldsymbol{j}$ and b_k are defined as:

$$\begin{array}{lllll} b_0 = -4, & b_1 = \dfrac{1}{3}, & b_2 = \dfrac{1}{6}, & b_3 = 0, & \text{19-point stencil} \\ b_0 = -\dfrac{64}{15}, & b_1 = \dfrac{7}{15}, & b_2 = \dfrac{1}{10}, & b_3 = \dfrac{1}{30}, & \text{27-point stencil} \end{array}$$

The corresponding expressions for the truncation errors τ^h_{19}, τ^h_{27} for L^h_{19} , L^h_{27} , are given by:

$$\tau^h_{19}(\phi) = \frac{h^2}{12}(\Delta(\Delta\phi)) + h^4 L^{(6)}(\phi) + O(h^6)$$

and

$$\tau^h_{27}(\phi) = \frac{h^2}{12}(\Delta(\Delta\phi)) + \frac{h^4}{360}\left(\left(\Delta^2 + 2\left(\frac{\partial^4}{\partial x^2 \partial y^2} + \frac{\partial^4}{\partial y^2 \partial z^2} + \frac{\partial^4}{\partial z^2 \partial x^2}\right)\right)(\Delta\phi)\right) + h^6 L^{(8)}(\phi) + O(h^8) \quad \text{(B.1)}$$

where the $L^{(q)}$'s are homogeneous constant–coefficient q^{th}-order differential operators.

Appendix C

Deriving Hockney's Algorithm

A key building block of the MLC algorithm is computing free-space convolutions quickly using the Fast Fourier Transform, *Hockney's Algorithm*. When Hockney first presented this algorithm it took several puzzling years for people to accept the idea that a suitably padded DFT operation would generate the free-space solution of a compact source term.

For *any* discrete convolution of a compact source term f we can compute a solution to the full space convolution with just a finite convolution on a domain that is at double the size of the support of f.

$$\Phi(j) = (G * f)(j) = \sum_{l \in \mathbb{Z}^D} G(j-l)f(l) \tag{C.1}$$

$$j \in [0...n-1]^D \tag{C.2}$$

$$f(l) \equiv 0 \; l \notin [0...n-1]^D \tag{C.3}$$

$$\tag{C.4}$$

define $\tilde{l} = j - l, l = j - \tilde{l}$

$$\Phi(j) = (G * f)(j) = \sum_{\tilde{l} \in \mathbb{Z}^D} G(\tilde{l})f(j-\tilde{l}) \tag{C.5}$$

$$f(j-\tilde{l}) \equiv 0 \; j - \tilde{l} \notin [0...n-1]^D \text{ or rewritten as} \tag{C.6}$$

$$\tilde{l} \notin [-n+1...n-1]^D \tag{C.7}$$

which can be rewritten as

$$\Phi(j) = \sum_{\tilde{l}=-n+1}^{n-1} G(\tilde{l})f(j-\tilde{l}). \tag{C.8}$$

Over the finite discrete space $[-n+1..n-1]^D$ we can define the Discrete Fourier Transforms

$$f(j) = \sum_{k=-n+1}^{n-1} \hat{f}_k z^{kj} \text{ and } \qquad \hat{f}_k = \frac{1}{2n} \sum_{j-n+1}^{n-1} f(j) z^{-kj} \tag{C.9}$$

$$G(\tilde{l}) = \sum_{k=-n+1}^{n-1} \hat{G}_k z^{k\tilde{l}} \text{ and } \qquad \hat{G}_k = \frac{1}{2n} \sum_{\tilde{l}=-n+1}^{n-1} G(\tilde{l}) z^{-k\tilde{l}} \tag{C.10}$$

$$\text{So, } f(j-\tilde{l}) = \sum_{k=-n+1}^{n} \hat{f}_k z^{k(j-\tilde{l})}. \tag{C.11}$$

Now (C.8) becomes

$$\sum_{\tilde{l}} G(\tilde{l}) f(j-\tilde{l}) = \sum_{\tilde{l}} \sum_{k} G(\tilde{l}) z^{-k\tilde{l}} \hat{f}_k z^{kj} \tag{C.12}$$

$$\sum_{k} \sum_{\tilde{l}} G(\tilde{l}) z^{-k\tilde{l}} \hat{f}_k z^{kj} \tag{C.13}$$

$$2n \sum_{k} \hat{G}_k \hat{f}_k z^{kj} \tag{C.14}$$

This *diagonalizes* the problem and allows for more efficient computation techniques to be applied. In particular, ((C.14)) is the Fast Fourier Transform (FFT). That is, can use the *Circular Convolution Theorem.* Hockney's algorithm can be generalized to handle the solution at any $(n)^D$ destination region j by evaluating shifting the domain of G.

Appendix D

Computing $\mathbb{P}_{\mathcal{L}}(f^h)$

In the shell region between αR and βR We perform a projection onto the Lengendre Polynomial basis. In 3D the Legendre basis are the tensor products of the 1D polynomials. In 1D you have the generator Rodrigues' formula:

$$P_n(x) = \frac{1}{2^n n!}\frac{d^n}{dx^n}((x^2-1)^n) \tag{D.1}$$

A short tabulation of the 2D tensor products shows how higher-order polynomials are built

$$\begin{matrix} & & 1 & & \\ & & x \quad y & & \\ & \frac{1}{2}(3x^2-1) & xy & \frac{1}{2}(3y^2-1) & \\ \frac{1}{2}(5x^3-3x) & \frac{1}{2}y(3x^2-1) & & \frac{1}{2}x(3y^2-1) & \frac{1}{2}(5y^3-3y) \end{matrix}$$

and for 3D we have

$$P_{m,n}(x,y,z) = \prod P^r(x)P^s(y)P^t(z) \text{ such that } n = r+s+t, 0 \le m < \frac{n(n+1)}{2} \tag{D.2}$$

For an arbitrary f we can express it's value as the sum of weighted Legendre basis functions

$$\text{let } \boldsymbol{x} = [x,y,z] \tag{D.3}$$

$$f(\boldsymbol{x}) \approx f^{;P} = \sum_{0}^{m,n} a_{m,n}P_{m,n}(\boldsymbol{x}). \tag{D.4}$$

where n is the order of the polynomial and m is the m'th polynomial of that order.

We assume we can treat a region Ω_k as being normalized to the region $[-1, 1]$. Multiplying both sides of (D.4) by $P_n(\boldsymbol{x})$ and using the fact that P is an orthogonal basis and the identity

$$\int_{-1}^{1} P_{m,n}(\boldsymbol{x}) P_{m,n}(\boldsymbol{x}) = \prod \frac{2}{2r+1} \frac{2}{2s+1} \frac{2}{2t+1} := g_{m,n} \tag{D.5}$$

we can compute the polynomial coefficients as

$$a_{m,n} = \frac{1}{g_{m,n}} \int_{-1}^{1} f(\boldsymbol{x}) P_{m,n}(\boldsymbol{x}) d\boldsymbol{x}. \tag{D.6}$$

For a 4-th order accurate truncation error we need the set of polynomials up to $n = 3$, which for 3D requires 20 coefficients. For $n = 4$ there would be 35 coefficients to compute. To compute the integral in (D.6) we can perform numerical integration to sufficient accuracy using a uniform sampling of $P_{m,n}(\boldsymbol{x}) f(\boldsymbol{x})$ on the grid Ω_k. The 1D *Trapezoidal Rule* given n_i points normalized to the region from $[-1, 1]$ for a sampled function b is

$$T_{n_i} = \frac{2}{n_i} \left[\sum_{j=1}^{n_i - 1} b_j + \frac{1}{2}(b_0 + b_{n_i}) \right] \tag{D.7}$$

The higher-dimension Trapezoid integrals are tensor products of the 1D integrals. In 3D that give the corners a weighting of 1/8, the edge points a weight of 1/4 and the face points a weight of 1/2 to generate T_N. $T_{N/2}$ is the same integral with sampling every other point. *Simpson's Rule* can then be generated by the extrapolation of the Trapezoid rule, $S_N = \frac{4}{3} T_N - \frac{1}{3} T_{N/2}$. Extrapolation using Simpson's Rule generates *Boole's Rule*, $B_N = \frac{16}{15} S_N - \frac{1}{15} S_{N/2}$. Boole integration has an error that is $O(h^6)$, which is adequate for the order of polynomial integration we require for computing our coefficients for $n = [3, 4]$. To perform Boole integration we will use tensor products of the composite Boole Integration

$$\begin{aligned} h &= \frac{1 - (-1)}{4m} \\ x_k &= kh - 1 \\ I &= \frac{2h}{45} \sum_{k=0}^{m-1} [(7a(x_{4k}) + 32a(x_{4k+1}) + 12a(x_{4k+2}) + 32a(x_{4k+3}) + 7a(x_{4k+4})] \end{aligned}$$

The coefficients $a_{m,n}$ are communicated to neighboring patches in the $\bar{\alpha}$ region at both the same level l to compute $\phi^{l,loc,i}$ as well as to the coarser level where it only evaluated at the coarse sample node points.

Once the projection $P(f)$ is computed then the Hockney Algorithm is executed again.

Bibliography

[1] M. Adams, `HPGMG 1.0`*: A benchmark for ranking high performance computing systems*, (2014).

[2] R. C. Agarwal, F. G. Gustavson, and M. Zubair, *An efficient parallel algorithm for the 3-d fft nas parallel benchmark*, in Scalable High-Performance Computing Conference, 1994., Proceedings of the, IEEE, 1994, pp. 129–133.

[3] A. S. Almgren, T. Buttke, and P. Colella, *A fast adaptive vortex method in 3 dimensions*, J. Comput. Phys., 113 (1994), pp. 177–200.

[4] M. Anderson, G. Ballard, J. Demmel, and K. Keutzer, *Communication-avoiding qr decomposition for gpus*, in Parallel & Distributed Processing Symposium (IPDPS), 2011 IEEE International, IEEE, 2011, pp. 48–58.

[5] O. Ayala and L.-P. Wang, *Parallel implementation and scalability analysis of 3d fast fourier transform using 2d domain decomposition*, Parallel Computing, 39 (2013), pp. 58–77.

[6] J. Bachan, S. Baden, D. Bonachea, P. Hargrove, S. Hofmeyr, K. Ibrahim, M. Jacquelin, A. Kamil, B. Lelbach, and B. van Straalen, *Upc++ specification v1. 0, draft 6*, (2018).

[7] J. Bachan, S. Baden, D. Bonachea, P. Hargrove, S. Hofmeyr, K. Ibrahim, M. Jacquelin, A. Kamil, and B. van Straalen, *Upc++ programmers guide, v1. 0-2017.9*, (2017).

[8] S. B. Baden and N. P. Chrisochoides, *Structured adaptive mesh refinement (SAMR) grid methods*, vol. 117, Springer Science & Business Media, 2000.

[9] J. Baert, A. Lagae, and P. Dutré, *Out-of-core construction of sparse voxel octrees*, in Proceedings of the 5th high-performance graphics conference, ACM, 2013, pp. 27–32.

[10] G. Ballard, E. Carson, J. Demmel, M. Hoemmen, N. Knight, and O. Schwartz, *Communication lower bounds and optimal algorithms for numerical linear algebra*, Acta Numerica, 23 (2014), pp. 1–155.

[11] G. Ballard, J. Demmel, O. Holtz, and O. Schwartz, *Minimizing communication in numerical linear algebra*, SIAM Journal on Matrix Analysis and Applications, 32 (2011), pp. 866–901.

[12] J. Barnes and P. Hut, *A hierarchical o (n log n) force-calculation algorithm*, nature, 324 (1986), p. 446.

[13] L. Barroso, M. Marty, D. Patterson, and P. Ranganathan, *Attack of the killer microseconds*, Communications of the ACM, 60 (2017), pp. 48–54.

[14] P. Basu, A. Venkat, M. Hall, S. Williams, B. Van Straalen, and L. Oliker, *Compiler generation and autotuning of communication-avoiding operators for geometric multigrid*, tech. rep., Lawrence Berkeley National Lab.(LBNL), Berkeley, CA (United States), 2014.

[15] C. Bell, D. Bonachea, R. Nishtala, and K. Yelick, *Optimizing bandwidth limited problems using one-sided communication and overlap*, in Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International, IEEE, 2006, pp. 10–pp.

[16] W. Bhimji, D. Bard, M. Romanus, D. Paul, A. Ovsyannikov, B. Friesen, M. Bryson, J. Correa, G. K. Lockwood, V. Tsulaia, et al., *Accelerating science with the nersc burst buffer early user program*, (2016).

[17] A. Brandt and O. E. Livne, *Multigrid techniques: 1984 guide with applications to fluid dynamics*, vol. 67, SIAM, 2011.

[18] W. L. Briggs, *A Multigrid Tutorial*, SIAM, 1987.

[19] O. Buneman, *Analytic inversion of the five-point poisson operator*, Journal of Computational Physics, 8 (1971), pp. 500–505.

[20] A. Chan, P. Balaji, W. Gropp, and R. Thakur, *Communication analysis of parallel 3d fft for flat cartesian meshes on large blue gene systems*, in High Performance Computing-HiPC 2008, Springer, 2008, pp. 350–364.

[21] A. J. Chorin, *A numerical method for solving incompressible viscous flow problems*, Journal of computational physics, 2 (1967), pp. 12–26.

[22] ——, *A numerical method for solving incompressible viscous flow problems*, Journal of Computational Physics, 135 (1997), pp. 118–125.

[23] P. Colella, J. Bell, N. Keen, T. Ligocki, M. Lijewski, and B. Van Straalen, *Performance and scaling of locally-structured grid methods for partial differential equations*, in Journal of Physics: Conference Series, vol. 78, IOP Publishing, 2007, p. 012013.

[24] P. Colella, D. Graves, T. Ligocki, D. Martin, D. Modiano, D. Serafini, and B. Van Straalen, *Chombo software package for AMR applications*. `https://commons.lbl.gov/display/chombo/Chombo+Download+Page`.

[25] L. Collatz, *The numerical treatment of differential equations*, vol. 60, Springer Science & Business Media, 2012.

[26] J. W. Cooley and J. W. Tukey, *An algorithm for the machine calculation of complex fourier series*, Mathematics of computation, 19 (1965), pp. 297–301.

[27] K. Czechowski, C. Battaglino, C. McClanahan, K. Iyer, P.-K. Yeung, and R. Vuduc, *On the communication complexity of 3d ffts and its implications for exascale*, in Proceedings of the 26th ACM international conference on Supercomputing, ACM, 2012, pp. 205–214.

[28] J. Demmel, *Communication-avoiding algorithms for linear algebra and beyond.*, in IPDPS, 2013, p. 585.

[29] J. Demmel, M. Hoemmen, M. Mohiyuddin, and K. Yelick, *Avoiding communication in sparse matrix computations*, in Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on, IEEE, 2008, pp. 1–12.

[30] R. H. Dennard, F. H. Gaensslen, V. L. Rideout, E. Bassous, and A. R. LeBlanc, *Design of ion-implanted mosfet's with very small physical dimensions*, IEEE Journal of Solid-State Circuits, 9 (1974), pp. 256–268.

[31] A. Dubey, A. Almgren, J. Bell, M. Berzins, S. Brandt, G. Bryan, P. Colella, D. Graves, M. Lijewski, F. Löffler, et al., *A survey of high level frameworks in block-structured adaptive mesh refinement packages*, Journal of Parallel and Distributed Computing, 74 (2014), pp. 3217–3227.

[32] H. Esmaeilzadeh, E. Blem, R. S. Amant, K. Sankaralingam, and D. Burger, *Dark silicon and the end of multicore scaling*, in Computer Architecture (ISCA), 2011 38th Annual International Symposium on, IEEE, 2011, pp. 365–376.

[33] F. Ethridge and L. Greengard, *A new fast-multipole accelerated Poisson solver in two dimensions*, SIAM Journal Sci. Comput., 23 (2001), pp. 741–760.

[34] F. Franchetti and M. Puschel, *Generating high performance pruned fft implementations*, in Acoustics, Speech and Signal Processing, 2009. ICASSP 2009. IEEE International Conference on, IEEE, 2009, pp. 549–552.

[35] M. Frigo and S. G. Johnson, *FFTW: An adaptive software architecture for the FFT*, in ICASSP Conference Proceedings, vol. 3, ICASSP, 1998, pp. 1381–1384.

[36] M. Frigo and S. G. Johnson, *The design and implementation of FFTW3*, Proceedings of the IEEE, 93 (2005), pp. 216–231. Special issue on "Program Generation, Optimization, and Platform Adaptation".

[37] S. H. Fuller and L. I. Millett, *The Future of Computing Performance: Game Over or Next Level?*, The National Academies Press, 2011.

[38] L. Greengard and J.-Y. Lee, *A direct adaptive Poisson solver of arbitrary order accuracy*, J. Comput. Phys., 125 (1996), pp. 415–424.

[39] L. Greengard and V. Rokhlin, *A fast algorithm for particle simulations*, J. Comput. Phys., 73 (1987), pp. 325–348.

[40] D. Hackenberg, D. Molka, and W. E. Nagel, *Comparing cache architectures and coherency protocols on x86-64 multicore smp systems*, in Microarchitecture, 2009. MICRO-42. 42nd Annual IEEE/ACM International Symposium on, IEEE, 2009, pp. 413–422.

[41] R. W. Hockney, *The potential calculation and some applications*, Methods in Computational Physics, 9 (1970), pp. 135–211.

[42] R. W. Hockney and J. W. Eastwood, *Computer Simulation Using Particles*, McGraw-Hill, 1981.

[43] B. Hoefflinger, *Itrs: The international technology roadmap for semiconductors*, in Chips 2020, Springer, 2011, pp. 161–174.

[44] H. H. Hum and J. R. Goodman, *Forward state for use in cache coherency in a multiprocessor system*, July 26 2005. US Patent 6,922,756.

[45] J. Johnson and X. Xu, *Generating symmetric DFTs and equivariant FFT algorithms*, in Proceedings of the 2007 international symposium on Symbolic and algebraic computation, ISSAC '07, New York, NY, USA, 2007, ACM, pp. 195–202.

[46] C. Kavouklis and P. Colella, *Method of local corrections with higher-order moments*, too appear, (2014).

[47] C. Kavouklis and P. Colella, *Computation of volume potentials on structured grids using the method of local corrections*, arXiv preprint arXiv:1702.08111, (2017).

[48] I. Lashuk, A. Chandramowlishwaran, H. Langston, T.-A. Nguyen, R. Sampath, A. Shringarpure, R. Vuduc, L. Ying, D. Zorin, and G. Biros, *A massively parallel adaptive fast multipole method on heterogeneous architectures*, Commun. ACM, 55 (2012), pp. 101–109.

[49] X. Leroy, D. Doligez, A. Frisch, J. Garrigue, D. Rémy, and J. Vouillon, *The ocaml system release 4.02*, Institut National de Recherche en Informatique et en Automatique, 54 (2014).

[50] Y. J. Lo, S. Williams, B. Van Straalen, T. J. Ligocki, M. J. Cordery, N. J. Wright, M. W. Hall, and L. Oliker, *Roofline model toolkit: A practical tool for architectural and program analysis*, in International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems, Springer, 2014, pp. 129–148.

[51] D. F. Martin and P. Colella, *A cell-centered adaptive projection method for the incompressible euler equations*, Journal of computational Physics, 163 (2000), pp. 271–312.

[52] A. Mayo, *The fast solution of poisson's and the biharmonic equations on irregular regions*, SIAM Journal on Numerical Analysis, 21 (1984), pp. 285–299.

[53] P. McCorquodale, P. Colella, G. Balls, and S. Baden, *A local corrections algorithm for solving Poisson's equation in three dimensions*, Communications in Applied Mathematics and Computational Science, 2 (2007), pp. 57–81.

[54] D. Molka, D. Hackenberg, and R. Schöne, *Main memory and cache performance of intel sandy bridge and amd bulldozer*, in Proceedings of the workshop on Memory Systems Performance and Correctness, ACM, 2014, p. 4.

[55] G. M. Morton, *A computer oriented geodetic data base and a new technique in file sequencing*, (1966).

[56] MPI Forum, *Message Passing Interface (MPI)* . Forum Home Page `http://www.mpi-forum.org/` (Dec. 2009).

[57] A. OpenMP, *Openmp application program interface version 4.0*, 2013.

[58] A. Ovsyannikov, M. Romanus, B. Van Straalen, G. H. Weber, and D. Trebotich, *Scientific workflows at datawarp-speed: accelerated data-intensive science using nersc's burst buffer*, in Parallel Data Storage and data Intensive Scalable Computing Systems (PDSW-DISCS), 2016 1st Joint International Workshop on, IEEE, 2016, pp. 1–6.

[59] J. Pinkerton, *The case for rdma*, RDMA Consortium, May, 29 (2002), p. 27.

[60] C. A. Rendleman, V. E. Beckner, M. Lijewski, W. Crutchfield, and J. B. Bell, *Parallelization of structured, hierarchical adaptive mesh refinement algorithms*, Computing and Visualization in Science, 3 (2000), pp. 147–157.

[61] W. F. Spotz and G. F. Carey, *A high-order compact formulation for the 3d poisson equation*, Numerical Methods for Partial Differential Equations: An International Journal, 12 (1996), pp. 235–243.

[62] The HDF Group, *Hierarchical Data Format, version 5*, 1997-NNNN.

[63] U. Trottenberg, C. W. Oosterlee, and A. Schuller, *Multigrid*, Access Online via Elsevier, 2000.

[64] B. Van Straalen, P. Colella, D. T. Graves, and N. Keen, *Petascale block-structured amr applications without distributed meta-data*, in Euro-Par 2011 Parallel Processing, Springer, 2011, pp. 377–386.

[65] S. Williams, D. Kalamkar, A. Singh, A. M. Deshpande, B. V. Straalen, M. Smelyanskiy, A. Almgren, P. Dubey, J. Shalf, and L. Oliker, *Implementation and optimization of minigmg-a compact geometric multigrid benchmark*, (2012).

[66] S. Williams, D. D. Kalamkar, A. Singh, A. M. Deshpande, B. Van Straalen, M. Smelyanskiy, A. Almgren, P. Dubey, J. Shalf, and L. Oliker, *Optimization of geometric multigrid for emerging multi-and manycore processors*, in Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, IEEE Computer Society Press, 2012, p. 96.

[67] S. Williams, A. Waterman, and D. Patterson, *Roofline: an insightful visual performance model for multicore architectures*, Commun. ACM, 52 (2009), pp. 65–76.

[68] L. Ying, G. Biros, and D. Zorin, *A kernel-independent adaptive fast multipole algorithm in two and three dimensions*, Journal of Computational Physics, 196 (2004), pp. 591–626.

[69] T. Zhao, M. Hall, P. Basu, S. Williams, and H. Johansen, *Simd code generation for stencils on brick decompositions*, in Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, ACM, 2018, pp. 423–424.

[70] Y. Zheng, A. Kamil, M. B. Driscoll, H. Shan, and K. Yelick, *Upc++: a pgas extension for c++*, in Parallel and Distributed Processing Symposium, 2014 IEEE 28th International, IEEE, 2014, pp. 1105–1114.