FFT, FMM, OR MULTIGRID? A COMPARATIVE STUDY OF STATE-OF-THE-ART POISSON SOLVERS

AMIR GHOLOMI∗, DHAIRYA MALHOTRA∗, HARI SUNDAR∗, AND GEORGE BIROS∗

Abstract. We discuss the fast solution of the Poisson problem on a unit cube. We benchmark the performance of the most scalable methods for the Poisson problem: the Fast Fourier Transform (FFT), the Fast Multipole Method (FMM), the geometric multigrid (GMG) and algebraic multigrid (AMG). The GMG and FMM are high-order schemes for Poisson problems developed in our group. The AMG code is the ML Trilinos library and the FFT is the P3DFFT library. We examine and report results for weak scaling, strong scaling, and time to solution for uniform and highly refined grids. We present results on the Stampede system at the Texas Advanced Computing Center and on the Titan system at the Oak Ridge National Laboratory. In our largest test case, we solved a problem with 600 billion unknowns on 229,379 cores of Titan. Overall all methods scale quite well to these problem sizes. We have tested all of the methods with different source distributions. FFT is the method of choice for smooth source functions that can be resolved with a uniform mesh. However, it loses its speed in the presence of highly localized features in the source function. FMM and GMG considerably outperform FFT for those cases.

Key words. Poisson Solvers, Fast Fourier Transform, Fast Multipole Method, Multigrid, Parallel Computing, Exascale algorithms, Co-Design

AMS subject classifications. 17B63, 65T50, 65T40, 78M16, 65N55, 65Y05

1. Introduction. Consider the following problem: given $f$, a smooth and periodic function in the unit cube, find $u$ which is also smooth and periodic in the unit cube such that

$$(1.1) \quad -\Delta u = f,$$

where $\Delta$ is the Laplace operator. This is also known as the constant-coefficient Poisson problem. It encapsulates many of the difficulties in solving elliptic partial differential equations (PDEs). Algorithms for solving this problem, also known as “Poisson solvers” find applications in astrophysics, chemistry, mechanics, electromagnetics, statistics, and image processing, to name a few. Vendors like Intel and NVIDIA provide Poisson solvers in their math libraries. Examples of scientific computing libraries that provide Poisson solvers include PETSc, Trilinos, deal.II, and MATLAB. In other words, the significance of developing efficient Poisson solvers cannot be overstated.

Poisson solvers must scale to leadership architectures and trillions of unknowns. Therefore, direct matrix factorizations cannot be used. Example of methods that scale well are the FFT (based on spectral discretizations)\(^1\), the Fast Multipole Method (based discretize the integral equation reformulation of (1.1), and multigrid methods (for stencil-based discretizations). Other scalable methods include domain decomposition and wavelet transforms, which will not discuss here. Despite the existence of many different Poisson solvers there has been little work in directly benchmarking the computational efficiency of these methods. Such benchmarking is typical in other scientific computing areas (e.g., sorting, matrix computations, graph partitioning).

\(^{\ast}\)Institute for Computational Engineering and Sciences, The University of Texas at Austin, Austin, TX 78712. (i.amirgh@gmail.com, dhairya.malhotra88@gmail.com, hsundar@gmail.com, gbiros@acm.org).

\(^{1}\)FFT, can be used also to diagonalize and invert stencil discretizations on uniform grids. We are not discussing this case here.
Contributions and methodology. In this paper we benchmark four state-of-the-art algorithms and implementations, two from our group and two from different groups: The first solver is a 2D-decomposed parallel FFT using the P3DFFT library developed by Dmitry Pekurovsky [21]. P3DFFT can be used only for uniform grid discretizations. P3DFFT uses MPI and OpenMP.

The second solver is the ML algebraic multigrid solver which is part of the Trilinos library developed and maintained by the U.S. Department of Energy [13, 14]. In our runs we use ML with MPI.

The third solver is an in-house Geometric Multigrid scheme that uses continuous Galerkin discretizations on octree meshes. The low-order version of the code appeared in [26], but the high-order results we report here are new, as well as the outline of the algorithm. The library uses MPI and more details about the new algorithm will be presented in §2.

The fourth code is PvFMM, also an in-house novel parallel volume FMM that supports continuous as well as particle sources ($f$). It uses octree discretization using Chebyshev polynomials at each leaf node to represent $f$. PvFMM uses MPI and OpenMP and optionally the Intel’s Phi architecture [18].

We compare these four methods on two different architectures, Stampede and Titan, and we address two main questions:

- What are the constants in the complexity estimates? We perform weak and strong scaling studies to directly compare the complexity estimates on specific architectures using the same problem size. We have scaled our runs up to 229,376 cores on Titan for problems with up to 600 billion unknowns. Our goal here is not to fit a detailed performance model to the runs but rather to identify whether there are order-of-magnitude size differences in the performance between these methods.

- Which method is faster? For an application user what matters the most is the wall-clock time to solution. That is, given $f$, we would like to evaluate $u$ (typically at a given number of points) to a specified accuracy. We consider two main cases, highly oscillatory fields, for which a regular grid is necessary, and highly localized fields for which refined meshes are expected to be more effective.

To our knowledge, such a benchmark at these problem sizes and number of cores is the first of its kind. We view it as a companion to existing theoretical complexity analysis. In addition to work complexity given a problem size, we consider the issue of work complexity given a target accuracy using both uniform and non-uniform grids—the “right” problem size is not known a priori. One reason such a study has not taken place is that the underlying technologies have not been available at this scale. Indeed, we are not aware of any other distributed-memory FMM codes that allow $f$ to be an arbitrary function (most existing codes only support sums of delta functions, also known as point-FMMs methods). Also, the only other scalable, high-order, multigrid scheme we know is that of Paul Fischer’s group [17]. Both of our GMG and FMM codes support arbitrary order discretizations. In summary, we test weak and strong scalability of all of these methods and report time to solution, and setup time for different test cases.

Qualitatively, the main conclusions of our study can be summarized as follows: FFT is the method of choice for uniform discretizations even at large-core count problems. FMM and MG are the methods of choice in the presence of strongly-localized features. AMG scales well, but it is significantly slower, especially when
including setup costs. It is important to note that low-order discretizations (e.g. the 7-point Laplacian) end up being 1000× or more slower than high-order schemes and the FFT. For the class of problems we consider here low-order methods are simply not competitive.

Related work. There is rich literature discussing the accuracy and scalability of FFT, FMM and multigrid but to our knowledge, little work has been done on directly comparing the efficiency of these schemes. In [12], theoretical complexity estimates for the scaling of FFT and multigrid (for uniform grids) are provided and their implications towards the design of exascale architectures is discussed. In [6], a similar study is carried for the FFT, along with experimental results (also using FFT) on both CPU and hybrid systems. A nice performance model is introduced that accounts for both intra-node and inter-node communication costs. In [1], the authors consider complexity estimates that account for low-level details hardware details and consider the viability of different applications including FFT, and matrix vector multiplications, and molecular dynamics simulations. In [11], the authors discuss the scalability of algebraic multigrid (on uniform grids), provide performance models, and conduct an experimental scalability study on up to 65,536 cores. A perspective on scalability is given in [30]. In [32], the authors discuss the scalability of a point FMM code to exascale architectures and provide scalability results up to 32,768 cores and 40 billion unknowns. In our group we have worked on scalable geometric multigrid methods and their comparison to algebraic multigrid schemes [26] (but only for low-order discretizations), as well as parallel FMM schemes based on the kernel-independent variant of the FMM kernels [16, 31]. All of these studies are critical in understanding the scalability of the schemes and helping co-design the next architectures. We consider our study as a companion to these works as it provides experimental data that can be further analyzed using performance models. Also, with the exception of the work within our group, others have only considered uniformly refined grids (and low-order discretizations for the multigrid). Here we consider all cases: uniform and refined grids, low and high-order discretizations, and four major algorithms.

Finally, let us mention some select fast Poisson solvers. Other scalable approaches include hybrid domain decomposition methods [17]. A very efficient Poisson solver is based on a non-iterative domain decomposition method [19] using a low-order approximation scheme. In [15], that solver was compared with a high order volume FMM. The FMM solver required $4 \times -100 \times$ fewer unknowns. Other works based on FFTs, tree codes and multigrid that are highly scalable (albeit for low-order, or point FMM only) include [8, 20, 22, 23, 25].

Limitations. Our study is limited to problems with constant coefficients on the unit cube. AMG is the only method that is directly applicable to general geometries. GMG methods also can handle certain types of complex geometries. FMM can be used for constant-coefficient problems on arbitrary geometries. In our tests we only use periodic boundary conditions but the results apply to Neumann and Dirichlet problems on the unit cube. Also other methods like domain decomposition or hybrids like particle-in-cell methods are not discussed. One disadvantage of high-order methods is that when used with explicit time stepping they can lead to extremely small time steps. Finally, FFTs, FMM, and Multigrid can be also discussed in other contexts (e.g., Ewald sums, particle-in-cell methods, signal analysis).

Outline of the paper. In section §2, we summarize the basic facts about FFT, Multigrid, and FMM. In §3, we summarize the experimental setup, platforms, and the choice of the right hand sides. In §4, we present and discuss the results of our
experiments.

**Notation:** We use $p$ to denote the number of cores (this doesn’t include the cores of Phi), $q$ to denote the order of polynomial approximation for $f$ and $u$, $m$ the FMM approximation order for the far field, and $N$ the total number of unknowns.

2. Methods. Here we describe the basic algorithmic components of each method, and their overall complexity (setup and solve) and the solve complexity ($T_{FMM}, T_{FFT}, T_{GMG}$). In all of our results we assume that $N \gg p$ and the complexity estimates are stated for an uncongested hypercube topology and uniform grids.

2.1. The Fast Fourier Transform. The Fast Fourier Transform is an algorithm for computing the Discrete Fourier Transform (DFT) of a signal in $O(N \log N)$. Sequential implementations of FFT are straightforward to implement and several efficient implementations exist, e.g. FFTW [10]. In addition, several FFT algorithms have been proposed for distributed machines (for example see [9, 27, 28]). For computing 3D FFTs, the key challenge is in dividing the data across the processes. Most FFT algorithms partition the data to $p$ slabs or slices, each containing $N/p$ samples. Such an approach however limits the maximum number of process to the number of slices in the input data (basically $p$ is at most limited to $O(N^{1/3})$).

A recent distributed FFT library, P3DFFT library improves this bound to $O(N^{2/3})$ by using a 2D pencil decomposition [21]. Consider a 3D function of size $N = N_x \times N_y \times N_z$ whose FFT we wish to compute on $p$ processes. P3DFFT maps the processes into a 2D grid of size $p_x \times p_y = p$, each having a pencil of size $N_x/p_x \times N_y/p_y \times N_z$ (or an equivalent partition). In the limit of $p = N_x \times N_y = N^{2/3}$ processes, each process will get a one dimensional ‘pencil’ of the data (of length $N_z$). With the pencil decomposition, each process has all of the data in one direction (e.g. $z$ direction), and partial data of the other two directions ($x$ and $y$). Consequently, the 1D FFT along the $z$ direction can be performed independent of other processes, using a serial (multi-threaded) FFT algorithm.

Following the local 1D FFT (along the $z$-direction), the process communicates its data using MPI_Alltoallv to other processes in the same row ($p_x$) to form the transpose. After it receives the transpose data from other processes, it computes the 1D FFT along the next direction. This process is once again repeated in the column direction ($p_y$).

The communication cost of P3DFFT [21] is given by $O(N \sigma(p))$, where $\sigma(p)$ is the bisection bandwidth of the network; for a hypercube it is $p/2$. The total execution time of one 3D FFT can be approximated by

$$T_{FFT} = O\left(\frac{N \log N}{p}\right) + O\left(\frac{N}{p}\right).$$

To solve the Poisson problem one needs to compute FFT of $f$ in (1.1), scale it by the corresponding diagonal form of the Laplace operator (using a Hadamard product), and compute the inverse Fourier transform of the result. The wave number multiplication has a complexity of $O(N/p)$. Therefore, the overall time to solve the Poisson problem with FFT will be the same order as $T_{FFT}$.

The choice of network size will affect the performance of P3DFFT library, and it is suggested that $p_1 \ll p_2$ to achieve best performance [21]. This behavior was observed in our experiments on Titan which has a torus network, but was negligible on Stampede which has a fat-tree network. The timings that we report for the runs on Titan, correspond to the best empirically chosen network grid size.
We must also mention that we compiled the library with ‘FFTW_MEASURE’ planner flag. This flag is used to tune the library to the machine used. The choice of the planner flag will greatly affect the setup time. However, one should not that this is an initial cost that is not important when one is solving the problem across nonlinear iterations or in a time-stepping fashion. For other cases that the setup time is important, one can use a less optimal planner.

2.2. The Fast Multipole Method. The FMM was originally [5] developed to speedup the solution of N-body problems by reducing the complexity from $O(N^2)$ to $O(N)$. Solving the Poisson’s equation by computing volume potential is very similar to an N-body problem, except that the summation over source points is replaced by an integral over the continuous source density. The method described in [15], discusses the modifications required for volume FMM and this is the approach used in our implementation. We further optimized this method, added support for Intel Phi and extended it to a distributed memory method based on our hypercube communication algorithm [16].

We evaluate solution to (1.1) as a convolution of the source density function $f$ with $K(x) = -\frac{(4\pi|x|^{-1})}{(free ~space ~Green’s ~function ~for ~Laplace ~equation)}$. We accelerate computation of this convolution using FMM. The basic idea behind FMM is to construct a hierarchical decomposition of the computational domain using an octree. Then, the solution at each point $x$ can be evaluated by summing over contributions from all octants in the octree. This summation is split into near and far interactions:

$$u(x) = \sum_{B \in \text{Near}(x)} \int_{B} K(x - y) f(y) + \sum_{B \in \text{Far}(x)} \int_{B} K(x - y) f(y)$$

The near interactions (from $B \in \text{Near}(x)$) are computed through direct integration. The far interactions (from $B \in \text{Far}(x)$) are low-rank and can be approximated. In the following sections, we describe the tree construction and the different interactions in more detail.

2.2.1. Octree Construction. We partition the computational domain using an octree data structure. For each leaf octant $B$, we approximate the source density $f$ by Chebyshev polynomials of degree $q$

$$f(y) = \sum_{i,j,k \geq 0}^{i+j+k \leq q} \alpha_{i,j,k} T_i(y_1)T_j(y_2)T_k(y_3)$$

where, $T_i$ is the Chebyshev polynomial of degree $i$ and $y$ is a point in the octant $B$. The absolute sum of the highest order coefficients is used as an estimate of the truncation error. This error estimate is used to refine adaptively until a specified error tolerance is achieved.

We also apply 2:1-balance constraint on our octree, i.e. we constrain the difference in depth of adjacent leaf octants to be at most one. To do this, we need to further subdivide some octants and is called 2:1-balance refinement.

2.2.2. Far Interactions. We define a source and a target octant to be well separated (or far) if they are at the same depth in the octree and are not adjacent. To compute far interactions we use two building blocks: multipole expansions and local expansions. The multipole expansion approximates the potential of an octant far away from it. The local expansion approximates the potential within an octant.
due to sources far away from it. The interactions are then approximated by computing a multipole expansion (source-to-multipole, multipole-to-multipole) for the source octant, multipole-to-local (V-list) translation and then evaluating the local expansion (local-to-local, local-to-target) at the target octant. We use the kernel-independent variant of FMM in our implementation. The form of the multipole and local expansions and the V-list translation operator for this variant are discussed in detail in [31].

2.2.3. Near Interactions. The near interactions (source-to-target or U-list), between pairs of adjacent source and target octants, are computed using exact integration. Since these are singular and near-singular integrals (because the kernel has a singularity), it is not feasible to do this on the fly. We have to precompute integrals over Chebyshev basis functions, then the integral can be represented as a linear transformation:

\[ u(x) = \int B K(x - y) f(y) = \int B K(x - y) \sum_{i,j,k} \alpha_{i,j,k} T_{i,j,k}(y) = \sum_{i,j,k} \alpha_{i,j,k} I_x \]

We precompute these integrals for each target point \( x \), and for each interaction direction then construct interaction matrices. These interactions can then be evaluated through DGEMVs. To limit the number of possible interactions directions and make this precomputation possible, we need the 2:1-balance constraint (discussed above).

\( \text{(a) Upward Pass} \)

\( \text{(b) Downward Pass} \)

**Fig. 1:** (a) Upward Pass: constructing multipole expansions. (b) Downward Pass: constructing local expansions, evaluating near interactions.

2.2.4. Summary of Volume FMM. The overall algorithm for volume FMM can be summarized as follows:

- **Tree Construction:** Construct a piecewise Chebyshev approximation of the source density using octree based domain decomposition and perform 2:1-balance refinement.
- **Upward Pass:** For all leaf nodes apply source-to-multipole (S2M) translation to construct the multipole expansion from the Chebyshev approximation. For all non-leaf nodes apply multipole-to-multipole (M2M) translations in bottom-up order, to construct multipole expansion from that of its children, as shown in Fig 1(a).
- **Communication:** In the distributed memory implementation, we communicate the source density and the multipole expansions for ghost octants.
- **Downward Pass:** For all octree nodes, apply V-list and source-to-local (X-list) translations to construct the local expansion of each octant. In top-down
order apply the local-to-local (L2L) translation to all nodes (Fig 1(b)). For all leaf octants, apply local-to-target (L2T), multipole-to-target (W-list) and source-to-target (U-list) translations to construct the final target potential as piecewise Chebyshev interpolation.

In our results, we report tree construction as the setup phase. The upward-pass, communication and downward-pass together constitute the evaluation phase.

2.2.5. Parallel Fast Multipole Method. Here we list the important features of the intra-node parallelism (accelerators, multithreading and vectorization) and distributed memory parallelism. A detailed discussion of these optimizations is beyond the scope of this paper. We refer the interested readers to [18] for a detailed discussion of these concepts.

Asynchronous Execution on Intel Phi: We compute U,W,X-list interactions on Intel Phi and the remaining interactions (V-list, D2D and D2T) on the CPU. All computations and memory transfers between Phi and the host are asynchronous and overlapped with computation on CPU.

U,W,X-List Optimizations: We group similar interactions, those with the same interaction matrix or related by a spatial symmetry relation into a single matrix-matrix product, evaluated efficiently through DGEMM. This is crucial for execution on Phi, since the Phi requires the matrices to be large to achieve good performance.

V-List Optimizations: The V-list interactions involve computation of the Hadamard products, which has low computational intensity and is therefore bandwidth bound. We rearrange data and use spatial locality of V-list interactions to optimize cache utilization. This along with use of AVX and SSE vector intrinsics and OpenMP allowed us to achieve over 50% of peak performance for this operation on the Intel Sandy Bridge architecture.

Distributed 2:1 Balance Refinement: We developed a new distributed memory algorithm for 2:1 balance refinement, which is more robust for highly non-uniform octrees than our earlier implementation.

Distributed Memory Parallelism: We use Morton ordering to partition octants across processors during the tree construction. In the FMM evaluation, after the upward pass, we need to construct the local essential tree through a reduce-broadcast communication operation. For this, we use the hypercube communication scheme of [16].

2.2.6. Complexity. The cost of FMM evaluation is given by the number of interactions between the octree nodes weighted by the cost of each translation. The cost of each interaction depends on the multipole order $m$ and the order of Chebyshev polynomials $q$. Let $N_{oct}$ be the local octree nodes, $N_{leaf}$ the number of local leaf nodes. Also, let $N_U$, $N_V$, $N_W$, $N_X (= N_W)$ denote the number of interactions of each type U,V,W and X-list respectively. The overall cost is summarized in Table 1.

<table>
<thead>
<tr>
<th>Interaction Type</th>
<th>Computational Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>S2M, L2T</td>
<td>$O(N_{leaf} \times q^3 \times m^2)$</td>
</tr>
<tr>
<td>M2M, L2L</td>
<td>$O(N_{oct} \times m^2 \times m^2)$</td>
</tr>
<tr>
<td>W-list, X-list</td>
<td>$O(N_W \times m^2 \times q^3)$</td>
</tr>
<tr>
<td>U-list</td>
<td>$O(N_U \times q^3 \times q^3)$</td>
</tr>
<tr>
<td>V-list</td>
<td>$O(N_V \times m^3 + N_{oct} \times m^3 \log m)$</td>
</tr>
</tbody>
</table>

Table 1: Computational cost for each interaction type.
The communication cost for the hypercube communication scheme is discussed in detail in [16]. For an uncongested network, that work provides a worst case complexity which scales as \( O(N_s(q^3 + m^2)\sqrt{p}) \), where \( N_s \) is the maximum number of shared octants per processor. However, assuming that the messages are evenly distributed across processors in every stage of the hypercube communication process, we get a cost of \( O(N_s(q^3 + m^2)\log p) \). In our experiments with uniform octrees, the observed complexity agrees with this estimate. Also, since shared octants are near the boundary of the processor domains, we have \( N_s \sim \left(\frac{N_{\text{oct}}}{p}\right)^{2/3} \), where \( N_{\text{oct}} \) is total number of octants.

In the uniform octree case, there are \( N_{\text{oct}} = N/q^3 \) total octants, \( N_U = 27N_{\text{oct}} \) and \( N_V = 189N_{\text{oct}} \). Due to the large constant factors, the cost of U-list and V-list interactions dominate over other interactions and the overall cost is:

\[
T_{\text{FMM}} = O\left(q^3\frac{N}{p}\right) + O\left(\frac{m^3 N}{q^3 p}\right) + O\left(\left(\frac{N}{p}\right)^{2/3} q \log p\right).
\]

In our results, we also report the time for setup (tree construction and 2:1 balance refinement). For tree construction, the cost of Chebyshev approximation for \( O(N/q^3) \) octants, distributed across \( p \) processors is \( O(qN/p) \). During the adaptive refinement, we may also need to repeatedly redistribute octants across processors. This communication cost is data dependent and is difficult to analyze. For the 2:1 balance refinement, the cost is \( O\left(qN/p + (N \log N)/(pq^3)\right) \) assuming a hypercube interconnect.

2.3. Geometric and Algebraic Multigrid. Multigrid is one of the most effective solvers for elliptic operators. It is algorithmically optimal and easy to implement and parallelize for uniform grids. Multigrid comprises of two complimentary stages: smoothing and coarse grid correction. Smoothing involves the application of a smoother to reduce (oscillatory) high-frequency errors. Coarse-grid correction involves transferring information to a coarser grid through restriction, solving a coarse-grid system of equations, and then transferring the solution back to the original grid through prolongation (interpolation). The coarse-grid correction eliminates (smooth) low-frequency errors. This approach can be applied recursively to obtain a multilevel system consisting of progressively coarser meshes. The multigrid method for the discretization of (1.1) \( A_h u_h = f_h \) amounts to the recursive application of the well-known V-cycle multigrid scheme (Algorithm 2.1). Here, \( S \) is the smoother and \( k \) denotes the multigrid level. The solve at the coarsest level \((k = 0)\), is done using a direct solver.

Multigrid methods can be classified into two categories, geometric and algebraic. The primary difference is that GMG methods rely on the underlying mesh connectivity
for constructing coarser multigrid levels, whereas AMG methods are mesh-agnostic and work directly on the fine-grid matrix. Our GMG code is an extension of our previous work [26] to support high-order discretizations. For the AMG comparison, we use ML [13] from the Trilinos Project [14]. ML implements smoothed aggregation, a variant of AMG, and has shown excellent robustness and scalability. All experiments reported in this paper for multigrid (AMG and GMG) were performed using a single multigrid V-cycle as a preconditioner for the conjugate gradient (CG) method [29].

For the AMG and GMG scalability experiments, we report times (in seconds) for the following stages:

- **Setup**: setting up the multigrid hierarchy and computing the diagonal of the operator for the Jacobi smoother,
- **Eval**: applying the smoother and in the coarse grid solve, and
- **Comm**: performing restriction and prolongation.
- **Solve**: Eval + Comm.

In the rest of this section we provide additional details of our GMG implementation, especially where it differs from [26].

### 2.3.1. Meshing & Discretization.

Our parallel geometric multigrid framework is based on hexahedral meshes derived from adaptive octrees [3, 24, 26]. As the multigrid hierarchy is independent of the discretization order, the construction of the grid hierarchy is identical to [26].

We use high-order discretizations based on Legendre-Gauss-Lobatto (LGL) nodal basis functions for polynomial orders $1 \leq q \leq 16$. For tensorized nodal basis functions on hexahedral meshes, the application of elemental matrices to vectors can be implemented efficiently by exploiting the tensor structure of the basis functions, as is common for spectral elements, e.g., [7]. This results in a computational complexity of $O(q^4)$ for the element MatVec instead of $O(q^6)$ if the element matrices were assembled. Globally, across $p$ processors, using tensor products the work complexity for a MatVec is $O(Nq/p)$, requiring $O(Nq^3)$ storage\(^2\).

### 2.3.2. Smoother.

We use damped Jacobi smoothing with $\omega = \frac{2}{3}$ for all runs reported in this paper. Although the performance of the Jacobi smoother deteriorates rapidly at $q > 4$ for variable coefficient problems, it performs well for the constant coefficient Poisson problem. For variable coefficient problems high-order Multigrid, Chebyshev-accelerated Jacobi smoother [2] provides good multigrid convergence up to $q = 16$. The cost of applying the Chebyshev-accelerated Jacobi smoother is similar to the Jacobi smoother, although it does require estimation of the maximum eigenvalues of the system matrix, which has to be done during the setup.

### 2.3.3. Restriction & Prolongation Operators.

Similar to [26], the prolongation operator can be represented as a matrix-vector product with the input vector as the coarse grid nodal values and the output as the fine grid nodal values [24]. The matrix entries are the coarse grid shape functions evaluated at the fine-grid vertices, $p_i$, $P(i,j) = \phi_j^{k-1}(p_i)$. Here $\phi_j^k$ is the $j$th nodal basis on the $k$th grid. The main difference with respect to [26] is the use for the high-order basis for prolongation.

The restriction operator is the adjoint of the prolongation operator with respect to the mass-weighted inner products. This only requires the application of the transpose of the prolongation operator to vectors, see for instance [24].

\(^2\)as opposed to $O(Nq^3/p)$ work and $O(Nq^3)$ storage
2.3.4. Complexity. For a $q$-order discretization with $N$ unknowns, the number of elements in the mesh is $O(N/q^3)$. Given $p$ processes, the time complexity of building the multigrid hierarchy is $O(N/(pq^3) + \log p)$. The $O(N/(pq^3))$ corresponds to the coarsening and meshing operations, which are linear in the number of elements $(N/(pq^3))$. The second term accounts for the creation of the log $p$ multigrid levels. The complexity of enforcing 2:1-balance is $O(N/(pq^3) \log N/(pq^3))$ [4]. The cost of a MatVec is $O(Nq/p)$. Assuming a uniform grid, we can estimate the communication costs as well: Each coarsening, balancing, and partition-correction call requires additional $O(\log p)$ time to MPI_Allgather the local element count (one long integer). For partitioning and transferring, all elements of a process can potentially be communicated to $O(1)$ processes. These transfers are implemented using non-blocking point-to-point communications, therefore the communication complexity per process is $O(N/p)$. Thus the complexity for the GMG solve (without setup) is

$$T_{\text{GMG}} = O\left(\frac{Nq}{p}\right) + O(\log p).$$

2.3.5. Caveats. It is important to point out that although our GMG code is capable of handling complex geometries, the version used in this comparison is optimized for a unit cube domain with periodic boundary conditions. The setup phase, specifically the computation of the diagonal will be significantly more expensive for other domains. In addition, the smoother will also be more expensive due the need to compute the Jacobian of the mapping (this can be traded for pre-computation and additional storage) from the reference element to each element.

3. Experimental setup. In this section we give details on the experimental setup we used to test the methods.

Hardware. The hardware employed for the runtime experiments carried out is the Stampede system at TACC and Titan at ORNL. Stampede entered production in January 2013 and is a high-performance Linux cluster consisting of 6,400 compute nodes, each with dual, eight-core processors for a total of 102,400 CPU-cores. The dual-CPUs in each host are Intel Xeon E5 (Sandy Bridge) processors running at 2.7GHz with 2GB/core of memory and a three-level cache. The nodes also feature the new Intel Xeon Phi coprocessors. Stampede has a 56GB/s FDR Mellanox InfiniBand network connected in a fat tree configuration which carries all high-speed traffic (including both MPI and parallel file-system data). Titan is a Cray XK7 with a total of 18,688 nodes consisting of a single 16-core AMD Opteron 6200 series processor, for a total of 299,008 cores. Each node has 32GB of memory. It is also equipped with a Gemini interconnect and 600 terabytes of memory across all nodes.

Test Cases. To test the solvers we consider two synthetic cases for $u$. The first test case is an oscillatory field given by $u(x_1, x_2, x_3) = \sin(2\pi k x_1) \sin(2\pi k x_2) \sin(2\pi k x_3)$. This field is a proxy for problems that require uniform resolutions. FFT can resolve such a field to machine precision using $2^k$ points per dimension. First-order methods (for example the 7-point Laplacian or linear finite elements) require roughly $4k$ points/per digit of accuracy. For two digits of accuracy in 3D, first-order methods require $64 \times$ more unknowns than FFT, and for seven digits of accuracy require $2048^3k$ more unknowns than FFT. Higher-order methods, reduce the work significantly, but FFT remains optimal.

The second test case is $u = \exp(- (r/R)^a)$, which approximates a piecewise constant function supported at the center of a sphere of radius $R$; and $r$ is the distance
from the point we evaluate \( u \) to the center of the sphere. As \( \alpha \) increases \( u \) develops very sharp gradients around \( r = R \) and is almost constant everywhere else. Roughly speaking, every time we double \( \alpha \) we increase the spectrum of \( u \) by a factor of two and therefore in 3D the number of unknowns for FFT increases by a factor of eight. At the limit as \( u \) becomes discontinuous FFT loses resolution. On the other hand an adaptive method only refines around the area of discontinuity and can resolve the solution for large \( \alpha \). Figure 2 shows an example of an adaptive mesh used by FMM. As a result, it is expected that FMM or GMG/AMG be the optimal method for this test case.

![Adaptive mesh structure for test case 2](image)

**Fig. 2:** Adaptive mesh structure for test case 2, the green cube (top quadrant) is the adaptive mesh of FMM-6 and the blue cube (lower quadrant) is that of FMM-14. FFT can only use a uniform mesh.

For both test cases, the source function \( f \) is computed analytically from \( f = -\Delta u \). This source function is then given to each of the four methods, to compute \( u_N \) numerically. One can then compute pointwise \( \ell_\infty \) error norm by comparing the exact and numerical solutions. These two test cases are used to measure the time-to-solution performance of the four schemes.

**Software.** In all cases, we run our codes with no other tuning other than setting the target accuracy. For the FMM we have two settings, high accuracy \((q = 14, m = 9)\) and low accuracy \( q = 6 \) and \( m = 3 \). For FMM the adaptivity criterion is based on the decay of the tail of the Chebyshev expansion of \( f \) at every leaf node. For GMG and AMG the adaptivity is based on the gradient of \( f \). For the multigrid case the smoothing is done using a pointwise Jacobi scheme with two pre-smoothing steps, and one post-smoothing step. The algebraic residual was driven (using CG) to machine precision in all runs. The coarse solves were done using sparse LU factorizations.

**4. Results.** We are evaluating our geometric multigrid (GMG) with \( q = 1, 4, 8, 10, 14 \), the fast multipole method with CPUs (FMM), the hybrid CPU+Phi (FMMPHI) with \( q = 6, 14 \), and finally the (FFT) using the P3DFFT library. As a reference we also perform some runs with Trilinos’s ML algebraic multigrid (AMG)
Fig. 3: Strong scaling results on Stampede. We report overall evaluation time in seconds for a problem with $N = 1024^3$. Representative overall efficiencies are as follows: FFT (60%), FMM-14 (64%), AMG-1 (33%), GMG-4 (67%), FMMPHI-14 (57%). FFT is the fastest method. We can observe orders of magnitude differences between the different variants. Although, we don’t report the results in detail here, we also tested AMG-4 and it was two times slower than AMG-1.

library. When using AMG, we are using matrices that are generated by our library using first and fourth order polynomials. AMG is mostly used as a reference point to get a sense of how our solvers and P3DFFT perform.

We use the following notation to denote the different variants of the methods we are testing: “X-$q$” indicates method “X” ran with $q$-th order polynomials. For example “GMG-4” indicates geometric multigrid with fourth order polynomials; “AMG-1”, indicates algebraic multigrid with linear polynomials.

We have performed multiple tests to benchmark these methods. First of all, we have tested weak/strong scalability on Stampede and Titan. This is an important test, since if a method fails to scale after a certain limit, it would not be efficient for large problem sizes. To clarify, we must mention that this test is not related to the test cases defined before, since we are testing all of the methods with a prescribed number of unknowns and we are not concerned with accuracy. Other than this test, what is important is time to solution for each method for a given accuracy. From a practical point of view, the desired method, is the one with a good scalability and the least time to solution. We report timing breakdown (i.e. wall-clock time of “Setup”, “Comm” and “Solve” times) for all of the methods for the two test cases defined in §3. The “Setup” that involves costs that incur while building data-structures whenever a change in the number of unknowns or their spatial distribution takes place; “Solve” is the time required to evaluate the solution $u$ whenever a new right-hand side $f$ is specified (without changing the resolution); and “Comm” is the distributed memory communication costs during the “Solve” phase. For AMG library we do not report communication costs, as the library does not provide such an option (as of ML 5.0).
As mentioned in the introduction more detailed analysis of the three algorithms (along with discussion on single-core performance) can be found in [18, 21, 26].

**Strong scaling analysis on Stampede, uniform grid.** To demonstrate the overhead and performance characteristics of the methods, we consider the strong scaling for a problem with one billion unknowns ($10^9$). We report “Solve” time in Figure 3; and the “Setup” and “Comm” times in Table 2. We report “Solve” time separately since in many cases setup time is amortized across nonlinear iterations or time-stepping. As one can see, FFT is clearly the fastest method, although FMM is not far behind. Here AMG-1, despite having less computation per unknown is somewhat slower. All the methods scale quite well. Also note that GMG-1 is slower than GMG-4, due to the lower compute intensity.

The FFT setup times are quite high, but in most cases these are done only once during the course of a calculation, since it is unusual to change resolution online when using FFTs.

**Weak scaling on Stampede.** Again, to illustrate the constant factors in the complexity estimates for the different methods, we consider a weak scaling test in which we keep the number of unknowns per core fixed to roughly one million and we increase $N$ and $p$, keeping their ratio fixed Figure 4. The observed end-to-end efficiencies for the different methods are as follows: FFT(49%), FMM-14(86%), FMMPHI-14(%82), GMG-1(96%), GMG-4(86%). FFT does not scale as well as other methods but it is still quite faster. The FMMPHI scales quite well and is almost as fast as

<table>
<thead>
<tr>
<th>$p$</th>
<th>phase</th>
<th>AMG-1</th>
<th>GMG-1</th>
<th>GMG-4</th>
<th>FMM-14</th>
<th>FMMPHI-14</th>
<th>FFT</th>
</tr>
</thead>
<tbody>
<tr>
<td>1,024</td>
<td>Setup</td>
<td>5.4</td>
<td>1.5</td>
<td>0.3</td>
<td>5.2</td>
<td>5.2</td>
<td>30.7</td>
</tr>
<tr>
<td></td>
<td>Comm</td>
<td>-</td>
<td>2.8</td>
<td>2.9</td>
<td>0.2</td>
<td>0.2</td>
<td>0.2</td>
</tr>
<tr>
<td></td>
<td>Solve</td>
<td>15</td>
<td>26.2</td>
<td>11.8</td>
<td>5.0</td>
<td>3.0</td>
<td>1.4</td>
</tr>
<tr>
<td>4,096</td>
<td>Setup</td>
<td>4.1</td>
<td>0.7</td>
<td>0.3</td>
<td>3.9</td>
<td>3.9</td>
<td>15.5</td>
</tr>
<tr>
<td></td>
<td>Comm</td>
<td>-</td>
<td>0.9</td>
<td>0.9</td>
<td>0.1</td>
<td>0.9</td>
<td>0.1</td>
</tr>
<tr>
<td></td>
<td>Solve</td>
<td>4.7</td>
<td>6.3</td>
<td>2.5</td>
<td>1.5</td>
<td>0.8</td>
<td>0.4</td>
</tr>
<tr>
<td>16,384</td>
<td>Setup</td>
<td>15</td>
<td>0.5</td>
<td>0.4</td>
<td>5.2</td>
<td>5.2</td>
<td>3.9</td>
</tr>
<tr>
<td></td>
<td>Comm</td>
<td>-</td>
<td>0.9</td>
<td>0.8</td>
<td>0.1</td>
<td>0.1</td>
<td>0.04</td>
</tr>
<tr>
<td></td>
<td>Solve</td>
<td>1.9</td>
<td>1.6</td>
<td>0.6</td>
<td>0.5</td>
<td>0.3</td>
<td>0.1</td>
</tr>
</tbody>
</table>

Table 2: Strong scaling on Stampede: In this table we give representative break-downs of the timings (in seconds) for the smooth source distribution with $N = 10^9$. GMG has insignificant setup costs. AMG has a higher cost as it needs to build the multigrid hierarchy algebraically using graph coarsening. FFT has large setup costs due to tuning to the particular topology. Finally, FMM has setup costs that are in the order of the solve time; these costs are related to building the near and far lists.

<table>
<thead>
<tr>
<th>$N/p$</th>
<th>phase</th>
<th>AMG-1</th>
<th>GMG-1</th>
<th>GMG-4</th>
<th>FMM-14</th>
<th>FMMPHI-14</th>
<th>FFT</th>
</tr>
</thead>
<tbody>
<tr>
<td>512$^3$/128</td>
<td>Setup</td>
<td>4.9</td>
<td>1.7</td>
<td>0.1</td>
<td>4.1</td>
<td>4.1</td>
<td>28.3</td>
</tr>
<tr>
<td></td>
<td>Comm</td>
<td>-</td>
<td>2.0</td>
<td>2.1</td>
<td>0.2</td>
<td>0.2</td>
<td>0.2</td>
</tr>
<tr>
<td></td>
<td>Solve</td>
<td>14.9</td>
<td>32.9</td>
<td>15.8</td>
<td>4.9</td>
<td>2.8</td>
<td>1.1</td>
</tr>
<tr>
<td>1024$^3$/1,024</td>
<td>Setup</td>
<td>5.4</td>
<td>1.5</td>
<td>0.3</td>
<td>4.5</td>
<td>4.5</td>
<td>30.8</td>
</tr>
<tr>
<td></td>
<td>Comm</td>
<td>-</td>
<td>2.8</td>
<td>3.0</td>
<td>0.2</td>
<td>0.2</td>
<td>0.2</td>
</tr>
<tr>
<td></td>
<td>Solve</td>
<td>14.9</td>
<td>32.2</td>
<td>16.9</td>
<td>3.1</td>
<td>3.0</td>
<td>1.1</td>
</tr>
<tr>
<td>2048$^3$/8,192</td>
<td>Setup</td>
<td>6.5</td>
<td>1.7</td>
<td>0.5</td>
<td>5.6</td>
<td>5.6</td>
<td>36.1</td>
</tr>
<tr>
<td></td>
<td>Comm</td>
<td>-</td>
<td>4.3</td>
<td>4.6</td>
<td>0.3</td>
<td>0.3</td>
<td>0.4</td>
</tr>
<tr>
<td></td>
<td>Solve</td>
<td>15.2</td>
<td>35.7</td>
<td>17.9</td>
<td>2.3</td>
<td>2.1</td>
<td>1.4</td>
</tr>
<tr>
<td>4096$^3$/65,536</td>
<td>Setup</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>Comm</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>Solve</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

Table 3: Weak scaling on Stampede: In this table we give representative break-downs of the weak scaling experiments performed on Stampede. Setup time, communication time, and ‘solve’ time are reported for each method.
FFT. It is worth noting that although GMG-4 does not scale as well as GMG-1, but it is about two times faster. This is due to the higher order scheme used in GMG-4.

Strong and weak scaling on Titan. On Titan we have performed weak scaling with FFT, and FMM-14. (If accepted, the final version we will also have results for GMG and AMG. The low-order versions for these methods have already scaled to 260K cores on Jaguar [26].) The results are summarized in Table 4 for a problem with 550 billion unknowns ($N = 8192^3$) for the FFT and 600 billion unknowns for the FMM. (Due to different discretization scheme of FMM, it is not possible to have exactly the same number of unknowns.)

For the weak scaling here we just report the “Solve” times for FFT and FMM. We observe that the FMM is almost $5 \times$ slower than the FFT. This occurs mainly
Table 5: Weak scaling on Titan. We report wall-clock time in seconds for the FFT and FMM with 14th order polynomials. The largest problem size corresponds to half a trillion unknowns. FFT is significantly faster than FMM.

Table 6: Time-to-solution for test case 1 (oscillatory field) performed on Stampede. Here we report the number of cores $p$, the effective resolution (wavenumber) $k$, the relative discrete infinity norm $\ell_\infty$ of the error in $u$, “Solve” time “$T$” and the uniform refinement level “$L$”, required to achieve single-precision accuracy or better. For the FFT we don’t report errors since it resolves the solution to machine accuracy. FFT dramatically outperforms FMM and GMG, since it can resolve the problem with considerably fewer unknowns. FMM requires $8^L q^3/6$ (and GMG $8^L q^3$) or roughly 9 billion unknowns to resolve a $u$ with $k = 128$ to single precision; FFT can resolve $k = 256$ to machine precision with nearly $70 \times$ fewer points. Also note that the target precision doesn’t change the conclusions, it just determines the overall problem size. Also FMMPHI-14 is roughly twice faster; bit still it cannot compete with the CPU-only FFT.

because FFT is memory bound and less sensitive to lower peak performance (which is the case for Titan CPUs compared to Stampede CPUs) so FMM runs slower on Titan whereas FFT is not affected as much.

Time to solution weak scaling. So as we saw in the previous paragraphs, one can test these solvers using the same number of unknowns across solvers, where significant differences was observed. However, what matters most is time to solution. Once this is factored in, the differences between the solvers become more pronounced.

So let’s start by looking at test case 1 (the oscillatory synthetic problem), a problem that requires uniform refinement. We report the “Solve” time to reach single precision accuracy in Table 6. From this first experiment, it is clear that FFT is extremely efficient for these problem types. This remains true even for smaller desired accuracies. For example, for an 8,192-core run, if we retain the same grid and we change $k$ from 128 to 256, the errors for FMM and GMG drop to three digits of accuracy whereas FFT can still resolve it to machine precision. Let us emphasize that all this is well known. We just want to highlight that this is the case even if one uses very high-order FMM or Galerkin methods.

But what about solutions with localized features as in test case 2? We report the “Solve” time to solve the nearly discontinuous $u$ in Table 7. Now the story differs dramatically. FFT starts well, but cannot resolve the length-scales of the solution and
Table 7: Time-to-solution for test case 2 (discontinuous field) performed on Stampede. Here $\alpha$ is a measure of the difficulty of resolving $u$. The higher the $\alpha$ value the sharper the derivatives. We report “Solve” time $T$ and the required number of unknowns $N$ to achieve single precision accuracy or higher. In all of these runs and for all three methods the relative error $\ell_{\infty}$ is less than $4E^{-7}$. We used GMG-10, because in our test it was the fastest variant of GMG across GMG-4, GMG-8, and GMG-14.

<table>
<thead>
<tr>
<th>$p$</th>
<th>$\alpha$</th>
<th>GMG-10</th>
<th>FMM-14</th>
<th>FMMPHI-14</th>
<th>FFT</th>
</tr>
</thead>
<tbody>
<tr>
<td>32</td>
<td>10</td>
<td>8.6E+7</td>
<td>0.5</td>
<td>3E+6</td>
<td>0.4</td>
</tr>
<tr>
<td>512</td>
<td>40</td>
<td>8.8E+8</td>
<td>0.7</td>
<td>6E+7</td>
<td>0.3</td>
</tr>
<tr>
<td>8,192</td>
<td>160</td>
<td>9.7E+9</td>
<td>0.8</td>
<td>1E+9</td>
<td>0.5</td>
</tr>
<tr>
<td>32,768</td>
<td>320</td>
<td>10.8E+9</td>
<td>1.8</td>
<td>4E+9</td>
<td>0.9</td>
</tr>
<tr>
<td>65,536</td>
<td>640</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td>100,800</td>
<td>640</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>--</td>
</tr>
</tbody>
</table>

the uniform grid size shoots up; for the largest problem size FFT requires 100× more unknowns than those required by FMM and GMG. Furthermore, FMM outperforms multigrid significantly. Understanding this requires a lengthier analysis. In passing, let us mention that the adaptive multigrid is quite efficiently implemented but cannot take advantage of dense DGEMMs as efficiently as FMM due to the nature of the algorithm. As a result its floating point performance is not as good as that of FMM. Finally, we would like to remark that both GMG and FMM over-refine to maintain 2:1 balancing. With the current technologies this is necessary to ensure good performance.

5. Conclusions. In a nutshell the conclusions are the following:

- Although FMM and in particular FMMPHI perform comparably with FFT for the same number of unknowns (on Stampede), this is somewhat misleading. For smooth target functions, FFT’s resolution power per unknown is much higher than any piecewise polynomial-based interpolation. For this reason, one should note that it is not possible to compare these methods using exactly the same $N$.

- In our tests, even when we use the same $N$ to compare methods, FFT outperforms the other methods for uniform grids. As expected the communication cost of FFT does not scale as favorably. However, FFT requires far fewer unknowns for functions that are within its resolution, and it also has the smallest complexity estimate. So even if the communication scaling is suboptimal, overall the method is much faster for such cases.

- For fields that have local features or discontinuities FFT loses its advantage. Adaptively refined algorithms like GMG and FMM can resolve such fields more effectively since they require orders of magnitude fewer unknowns than FFT.

- Both FMM and GMG methods scale quite well since they are matrix free and both use octree-based hierarchical space decomposition methods. The structure of the calculation however is different, with the FMM offering significant opportunities for task parallelism and compute intensity. On the other hand, the setup of FMM is more expensive.

- A state of the art algebraic solver is orders of magnitude slower than the fastest variants. Of course, AMG is much more general than FFT, FMM, or GMG as it can deal with arbitrary geometries, variable and anisotropic coefficients, and more general boundary conditions. However, many important applications require fast Poisson solvers on a cube with boundary conditions
that FFT, FMM, or GMG can handle. Therefore, using a method designed for more general situations can incur high and unnecessary computational costs.

- All three methods can be further improved: by integrating tightly with accelerators, further improving single-node performance, and trying improve the per leaf high-order calculations (for FMM and GMG).
- A robust black-box solver should be a hybrid method for cases when the true solution is a superposition of a highly oscillatory field and a highly localized field. This is specially true for problems with more complex geometries and boundary conditions.
- It would be tempting to use the expressions for $T_{FMM}$, $T_{FFT}$, and $T_{GMG}$ of §2 with $N$ fixed to try to quantify the effect of the order $q$ on the communication and computation costs and how it compares between the methods. However $N$ does depend on $q$ and the relation is non-trivial.

**Acknowledgment.** The authors would like to thank Omar Ghattas, Leslie Greengard, and Georg Stadler for useful discussions on this topic. Also, we are grateful to Judy Hill of ORNL for her help with the experiments on Titan. Support for this work was provided by: the U.S. Air Force Office of Scientific Research (AFOSR) Computational Mathematics program under award number FA9550-09-1-0608; the U.S. Department of Energy Office of Science (DOE-SC), Advanced Scientific Computing Research (ASCR), DE-FG02-09ER25914, and the U.S. National Science Foundation (NSF) Cyber-enabled Discovery and Innovation (CDI) program under awards CMS-1028889 and OPP-0941678, and the PetaApps program under award OCI-0749334. Computing time on the Cray XK7 Titan was provided by the Oak Ridge Leadership Computing Facility at Oak Ridge National Laboratory, which is supported by the Office of Science of the Department of Energy under Contract DE-AC05-00OR22725. Computing time on the Texas Advanced Computing Centers Stampede supercomputer was provided by an allocation from TACC and the NSF.

**REFERENCES**


