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

AMIR GHOLAMI*, 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 novel parallel schemes using high-order approximation for Poisson problems developed in our group. The FFT code is from P3DFFT library and AMG code from ML Trilinos 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. Our results show that FFT is the method of choice for smooth source functions that can be resolved with a uniform mesh. However, it loses its performance 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. The need for large scale parallel solvers for elliptic partial differential equations (PDES) pervades across a spectrum of problems with resolution requirements that cannot be accommodated on current systems. Several research groups are working on technologies that scale to trillions of unknowns and billions of cores. To illustrate some of the issues in scaling such solvers and to provide a (non-exhaustive) snapshot of current technologies, we conduct an experimental study of solving a simple model elliptic PDE and compare several state of the art methodologies.

To enable the use of all the different codes, we restrict our attention to the following model problem: given f, a smooth and periodic function in the unit cube, we wish to find u (also smooth and periodic in the unit cube) such that

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

where Δ 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 [1], Trilinos [20], deal.II [2], and MATLAB.

Poisson solvers must scale to trillions of unknowns. Example of methods that scale well are the FFT (based on spectral discretizations)[1], the Fast Multipole Method

---

*Institute for Computational Engineering and Sciences, The University of Texas at Austin, Austin, TX 78712. (i.amirgh@gmail.com, dhairya.malhotra@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.

(based on discretizing 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 be discussed 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 quite typical in other scientific computing areas (e.g., sorting, matrix computations, and graph partitioning).

Methodology and contributions. 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 [28]. 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 [18, 20]. 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 [33], but the scalability high-order results we report here are new, as well as the outline of the algorithm. The sequential algorithm is described in [35]. 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 [25].

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

- How does the cost per unknown compares for the different methods? This is related to the constant in the complexity estimates which are functions of the problem size, the number of processors, the approximation order, and of course the implementation. 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 differences in the performance between these methods.
- Which method is faster? 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 adaptively 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 [24]. 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 results of our study for solving (1.1) 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. Low-order discretizations (e.g. the 7-point Laplacian) end up being $1000\times$ 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 [17], 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 [10], a similar study is carried for the FFT, along with experimental results on both CPU and hybrid systems. An interesting performance model is introduced that accounts for both intra-node and inter-node communication costs. In [3], the authors consider complexity estimates that account for low-level hardware details and consider the viability of different applications including FFT, and matrix vector multiplications, and molecular dynamics simulations. In [16], 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 [40]. In [42], 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 [33] (but only for low-order discretizations), as well as parallel FMM schemes based on the kernel-independent variant of the FMM kernels [22, 41]. 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, except for the work in 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.

Other scalable approaches to solving the Poisson problem include hybrid domain decomposition methods [24]. A very efficient Poisson solver is based on a non-iterative domain decomposition method [26] using a low-order approximation scheme. In [21], 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 [12,27,29,30,32].

Limitations. Our study is limited to problems with constant coefficients on the unit cube and with periodic boundary conditions. AMG is the only method that is directly applicable to general geometries and problems with complex coefficients. GMG methods also can handle certain types of complex geometries. FMM can be used for constant-coefficient problems on arbitrary geometries but can also be extended to variable coefficients using volume integral equations. Complex geometries are also

possible but the technology for such problems is not as developed and robust as algebraic multigrid. In our tests we only use periodic boundary conditions but the results apply to Neumann and Dirichlet problems on the unit cube. For our high-order geometric multigrid we use a smoother that heuristically works well but we do not have theory that shows that we have made the best possible choice (see §2.3).

Comparing highly specialized codes with much general purpose codes (like the AMG solvers we use here) is problematic but, we think, informative as it provides some data for future developments of software that must be as efficient as possible if it is to be used in the million and billion-core systems. As we mentioned other methods like domain decomposition or hybrids like particle-in-cell methods are not discussed. One salient disadvantage of high-order methods is that when used with explicit time stepping schemes (for example, due to coupling to a transport equation) 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), which we do not do here.

Outline of the paper. In section §2, we summarize complexity estimates for FFT, Multigrid, and FMM. For FMM and geometric multigrid we provide some more detail since some components of the underlying algorithms are new. 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 ($T_{\mathrm{FMM}}, T_{\mathrm{FFT}}, T_{\mathrm{GMG}}$). In all of our results we assume that $N/p \gg 1$ 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 $\mathcal{O}(N \log N)$. Sequential implementations of FFT are straightforward to implement and several efficient implementations exist, e.g. FFTW [15]. In addition, several FFT algorithms have been proposed for distributed machines (for example see [14, 36, 37]). 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 $\mathcal{O}(N^{1/3})$).

A recent distributed FFT library, P3DFFT library improves this bound to $\mathcal{O}(N^{2/3})$ by using a 2D *pencil decomposition* [28]. 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 block (or *"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 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_Alltoall` 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 [28] is given by $\mathcal{O}(\frac{N}{\sigma(p)})$, 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

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

To solve the Poisson problem one needs to compute FFT of f in (1.1), scale it by the corresponding inverse 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 $\mathcal{O}(N/p)$. 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 $p_x \ll p_y$ to achieve the best performance [28]. 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 the `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 note that this is an initial cost that is not important when one is solving the problem for many different right hand sides. For other cases where the setup time is important, one can use a less optimal planner.

2.2. The Fast Multipole Method. The FMM was originally developed to speedup the solution of particle N-body problems by reducing the complexity from $\mathcal{O}(N^2)$ to $\mathcal{O}(N)$ [8,9]. Solving the Poisson's equation by computing volume potential is similar to an N-body problem, except that the summation over source points is replaced by an integral over the continuous source density. The work [13, 21] shows how one needs to modify the particle-FMM to obtain a volume FMM and this is the approach used in our implementation. Our contribution is that we have focused on the efficient implementation of this algorithm by considering efficient blocking and vectorization strategies for per core performance. We further optimized this method by adding support for the Intel Phi accelerator and extended it to a distributed memory method based on our hypercube communication algorithm [22]. Below we summarize the main components of our algorithm, which is described in full detail in [25].

We evaluate the solution to (1.1) as a convolution of the source density function f with $K(x) = -(4\pi|x|)^{-1}$ (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 order q

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

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, that is 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 [34].

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.[2] 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 [41].

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).

---

[2]We refer to multipole expansions and we use the term *"multipole order"* m to denote the accuracy of the far field approximation, following the analytic FMM. However, in our implementation we use the kernel independent FMM and m refers to the square root of the number of equivalent points. We used the terminology from the original FMM for the readers that are not familiar with the kernel independent FMM [41].
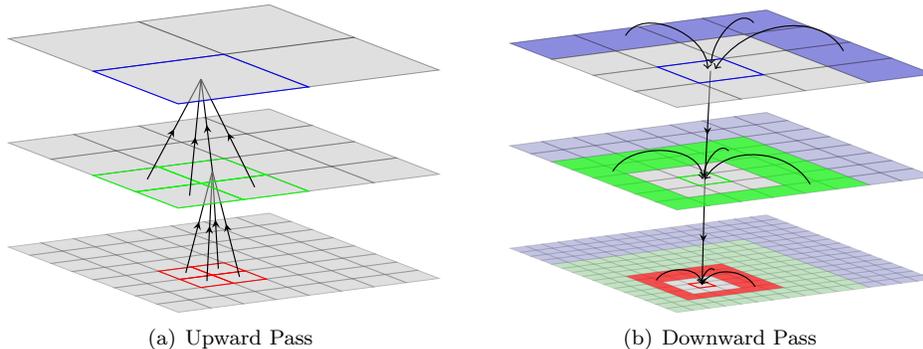
(a) Upward Pass   (b) Downward Pass

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 [25] 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, L2L and L2T) 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 [22].

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.

Interaction Type	Computational Cost
S2M, L2T	$\mathcal{O}(N_{leaf} \times q^3 \times m^2)$
M2M, L2L	$\mathcal{O}(N_{oct} \times m^2 \times m^2)$
W-list, X-list	$\mathcal{O}(N_W \times m^2 \times q^3)$
U-list	$\mathcal{O}(N_U \times q^3 \times q^3)$
V-list	$\mathcal{O}(N_V \times m^3 + N_{oct} \times m^3 \log m)$

Table 1: *Computational cost for each interaction type.*

The communication cost for the hypercube communication scheme is discussed in detail in [22]. For an uncongested network, that work provides a worst case complexity which scales as $\mathcal{O}(N_s(q^3 + m^2)\sqrt{p})$, 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 $\mathcal{O}(N_s(q^3 + m^2) \log p)$. 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 $N_s \sim (N_{oct}/p)^{2/3}$, where N_{oct} is total number of octants.

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

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

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 $\mathcal{O}(N/q^3)$ octants, distributed across p processors is $\mathcal{O}(qN/p)$. 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 $\mathcal{O}\left(qN/p + (N \log N)/(pq^3)\right)$ assuming a hypercube interconnect.

2.3. Geometric and Algebraic Multigrid. Multigrid [4, 19, 39] is one of the most effective solvers for elliptic operators. It is algorithmically optimal and easy to implement and parallelize for uniform grids. Multigrid consists of two complimentary stages: *smoothing* and *coarse grid correction*. Smoothing involves the application of a (typically stationary) iterative solver 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,

---

Algorithm 2.1 Multigrid V-Cycle

Pre-smooth:	u_k	$\leftarrow$	$S_k(u_k, f_k, A_k)$
Compute Residual:	r_k	$\leftarrow$	$f_k - A_k u_k$
Restrict:	r_{k-1}	$\leftarrow$	$R_k r_k$
Recurse:	e_{k-1}	$\leftarrow$	$A_{k-1}^{-1} r_{k-1}$
Correct:	u_k	$\leftarrow$	$u_k + P_k e_{k-1}$
Post-smooth:	u_k	$\leftarrow$	$S_k(u_k, f_k, A_k)$

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 [33] to support high-order discretizations. For the AMG comparison, we use *ML* [18] from the *Trilinos* Project [20]. *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 [38].

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 [33].

2.3.1. Meshing & Discretization. Our parallel geometric multigrid framework is based on hexahedral meshes derived from adaptive octrees [6, 31, 33]. As the multigrid hierarchy is independent of the discretization order, the construction of the grid hierarchy is identical to [33].

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., [11]. This results in a computational complexity of $\mathcal{O}(q^4)$ for the element MatVec instead of $\mathcal{O}(q^6)$ if the element matrices were assembled. Globally, across p processors, using tensor products the work complexity for a MatVec is $\mathcal{O}(Nq/p)$, requiring $\mathcal{O}(N)$ storage[3].

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 [5] 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 [33], 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 [31]. 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 ϕ_j^k is the jth nodal basis on the kth grid. The main difference with respect to [33] 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 [31].

2.3.4. Complexity. For a q-order discretization with N unknowns, the number of elements in the mesh is $\mathcal{O}(N/q^3)$. Given p processes, the time complexity of building the multigrid hierarchy is $\mathcal{O}(N/(pq^3) + \log p)$. The $\mathcal{O}(N/(pq^3))$ 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 $\mathcal{O}(N/(pq^3) \log N/(pq^3))$ [7]. The cost of a MatVec is $\mathcal{O}(Nq/p)$. Assuming a uniform grid, we can estimate the communication costs as well: Each coarsening, balancing, and partition-correction call requires additional $\mathcal{O}(\log p)$ 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 $\mathcal{O}(1)$ processes. These transfers are implemented using non-blocking point-to-point communications, therefore the communication complexity per process is $\mathcal{O}(N/p)$. Thus the complexity for the GMG solve (without setup) is

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

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

---

[3]as opposed to $\mathcal{O}(Nq^3/p)$ work and $\mathcal{O}(Nq^3)$ storage

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 Infini-Band 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.
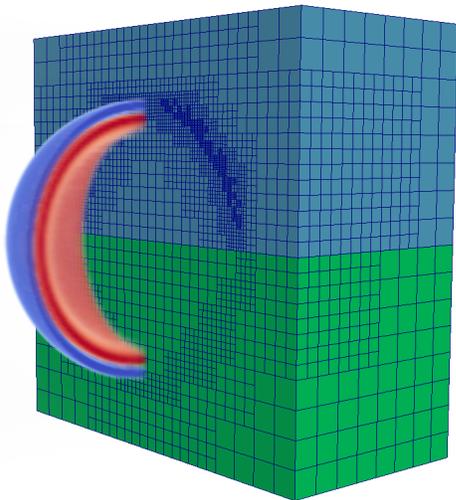

Fig. 2: *Adaptive mesh structure for the forcing term, f, in test case 2. The green cube (top quadrant) is the adaptive mesh using 6th order elements and the blue cube (lower quadrant) is that of using 14th order elements. The number of unknowns is significantly reduced.*

Test Cases. To test the time-to-solution we consider two synthetic cases for u, one that requires uniform discretization and one that requires nonuniform discretization.

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 requires uniform discretization. FFT can resolve u to machine precision using $2k$ points per dimension. First-order methods (for example the 7-point Laplacian or linear finite elements) require roughly $4k$ points per digit of accuracy. Roughly speaking, for two digits of accuracy in 3D, first-order methods require 64× more unknowns than FFT, and for seven digits of accuracy require $2048^3 k$ more unknowns than FFT. Higher-order methods, reduce the number of unknowns significantly but are still suboptimal compared to FFT for

11

this particular u.

The second test case is $u = \exp(-(r/R)^\alpha)$, 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 α increases u develops very sharp gradients around $r = R$ and is almost constant everywhere else. Roughly speaking, every time we double α 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 α. 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.

For both time-to-solution test cases, the source function is set to the closed form Laplacian of the exact solution, that is $f = -\Delta u$. This f is then used to compute the discretized solution u_N. We report $\|u - u_N\|_{\ell_\infty}$, where we abuse the notation and use u for the exact solution evaluated at the discretization points.

Software parameters. For the FMM we have two settings, high accuracy ($q = 14$, $m = 10^2$) and low accuracy $q = 6$ and $m = 4^2$. 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 is driven (using CG) to machine precision in all runs. The coarse solves were done using sparse parallel LU factorizations (SuperLU [23]).

4. Results. We are evaluating our geometric multigrid (**GMG**) with $q = 1, 4, 8, 10, 14$, our fast multipole method with CPUs (**FMM**) and 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**) library. When using AMG, we are using matrices that are generated by our library using first and fourth order polynomials. AMG is used as a reference point to show the difference between using a state of the art general-purpose solver over a more problem-specific solver.

FMMPHI is the only solver that uses hardware acceleration (Phi). All other solvers use only x86 CPUs.

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.

Our first test is a weak and strong scaling test on Stampede and Titan to assess the cost per unknown as a function of the method and the problem size without regard to the accuracy. The strong scaling results are reported in Tables 2 (Stampede) and 4 (Titan) and the weak scaling results are reported in Tables 3 (Stampede) and 5 (Titan). All these results where done using uniform discretization so that we could compare with FFT. FMM and GMG have overhead costs since they are not optimized specifically for regular grids.

The second set of experiments examine the time to solution, which are of course the ones that fully characterize the performance of these solvers. The results for the oscillatory u are reported in Table 6 and the results for the non-uniform u are reported in Table 7.
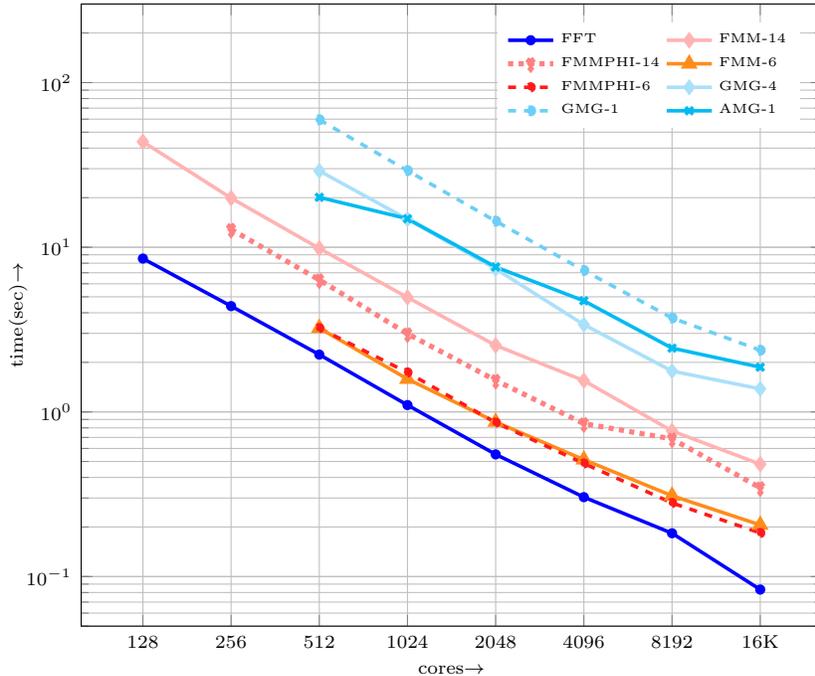
Fig. 3: *Here we report strong scaling results on Stampede that capture the cost per unknown for the different methods. These are only the "Solve" times compared to Table 2. For the FFT and FMM this involves just a single evaluation. For the GMG and AMG methods it involves several multigrid sweeps to bring the relative algebraic residual to zero (up to double precision). All times are in seconds for a problem with $N = 1024^3$. The parallel efficiency for the different methods is: FFT (60%), FMM-14 (64%), AMG-1 (33%), GMG-4 (67%), FMMPHI-14 (57%). In absolute terms FFT has always the lowest constant per unknown. We can observe orders of magnitude differences between the different variants. Since FMM involves just an evaluation lower order has a smaller constant than higher. Notice also that for our implementations, FMM with high-order is significantly faster than GMG low order. Although, we don't report the results in detail here, we also tested AMG-4 and it was 2× slower than AMG-1.*

We also report wall-clock times of "*Setup*", "*Comm*" and "*Solve*". The "*Setup*" involves costs that incur while building data-structures whenever a change in the number of unknowns or their spatial distribution takes place.For the GMG, AMG, building the grid hierarchy, 2:1 balancing it, computing coefficients for the Chebyshev smoother, and setting-up the sparse direct solver for the coarse solve. For the FMM, it involves building the tree, loading the precomputed near-interaction and translation operators and 2:1 balancing. For right-hand sides that require the same number of unknowns but different trees due to different distribution of spatial features, the setup has to be called again. For the FFT the setup phase involves parameter tuning to optimize CPU performance. "*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 we could not find an easy way to measure it (as of version ML 5.0). As we mentioned in the introduction more detailed analysis of the three algorithms (along with discussion on single-core performance) can be found in [25, 28, 33].

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

Table 2: *Here we report strong scaling results on Stampede. We report representative breakdowns of the timings (in seconds) for uniform grids with $N = 1024^3$. 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 careful tuning to the particular topology. Finally, FMM has setup costs that are in the order of the solve time; these costs are related tree construction loading precomputed tables, 2:1 balancing, and constructing the interaction lists.*

N/p	phase	AMG-1	GMG-1	GMG-4	FMM-14	FMMPHI-14	FFT
$512^3/128$	Setup	4.9	1.7	0.1	4.1	4.1	28.3
	Comm	–	2.0	2.1	0.2	0.2	0.2
	Solve	14.9	32.9	15.8	4.9	2.8	1.1
$1024^3/1,024$	Setup	5.4	1.5	0.3	4.5	4.5	30.8
	Comm	–	2.8	3.0	0.2	0.2	0.2
	Solve	14.9	32.2	16.9	5.1	3.0	1.1
$2048^3/8,192$	Setup	6.5	1.7	0.5	5.6	5.6	36.1
	Comm	–	4.3	4.6	0.3	0.3	0.4
	Solve	15.2	35.7	17.9	5.3	3.1	1.4

Table 3: *Here we report weak scaling results on the Stampede system. For the AMG runs, we could not directly measure the communication costs. The FFT is quite high because it performs parameter tuning to optimize inter-node and intra-node performance.*

Strong scaling analysis on Stampede, uniform grid. To demonstrate the overhead and performance characteristics of the methods, we consider strong scaling for a problem with one billion unknowns (1024^3). 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 due to the cost per V-cycle and the number of sweeps. All the methods scale quite well. Also notice that GMG-4 is faster than GMG-1 due to better compute intensity. The FFT setup time is quite high, but only depend on the problem size and number of processors and thus done just once during the course of a calculation since it is unusual to change resolution online when using FFTs.

Weak scaling on Stampede. 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 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. The low-order versions for these methods have already scaled to 260K cores on Jaguar [33].)
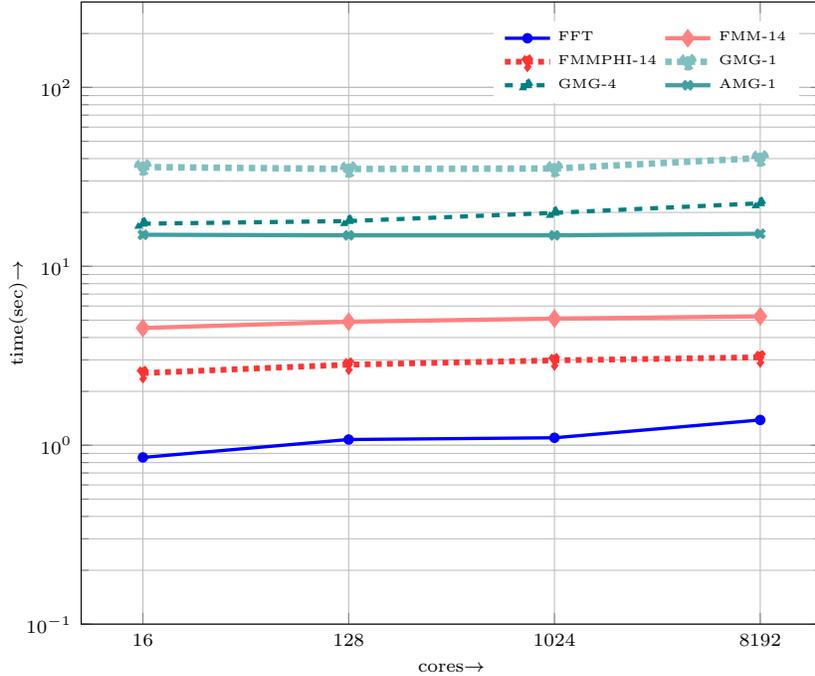
Fig. 4: *Weak scaling on Stampede: We report "Solve" time in seconds of different methods for a problem with a grain size of one million unknowns per core; the largest problem on Stampede has 8.5 billion unknowns on 8,192 cores. All the methods scale quite well. FFT is the fastest and the difference is over 30× over GMG-1.*

p	phase	**FMM-14**	**FFT**
57,344	Setup	105	227.4
	Comm	14.4	-
	Solve	230	44.2
114,688	Setup	44.4	97.4
	Comm	6.3	-
	Solve	105	19.1
229,376	Setup	30	-
	Comm	4.2	-
	Solve	58	11.3

Table 4: *Strong scaling on Titan (for the FFT and FMM only). We report wall-clock time in seconds for the FFT and FMM with 14th order polynomials for a problem with 550 billion unknowns ($N = 8192^3$). The efficiency is 96% for the FFT, and 99% for the FMM "Solve" phases. The communication costs for the FMM are order-of-magnitude smaller than the "Solve" and "Setup" phases. The setup costs for the FFT are substantial, but these are often one-time costs (unless the discretization is changed).*

The strong scaling 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× slower than the FFT. This is because FFT has much less work per unknown. Also it is "more" memory bound and less sensitive to lower per-node 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.

N	p	phase	FMM-14	FFT
$1,024^3$	448	Setup	16.0	–
		Comm	2.5	–
		Solve	51.8	4.5
$2,048^3$	3,584	Setup	16.7	–
		Comm	2.7	–
		Solve	52.5	6.0
$4,096^3$	28,672	Setup	17.6	–
		Comm	3.6	–
		Solve	53.1	8.0
$8,192^3$	229,376	Setup	19.0	–
		Comm	4.2	–
		Solve	57.8	11.3

Table 5: *Weak scaling results 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.*

		GMG-14			FMM-14			FMMPHI-14			FFT	
p	k	T	ℓ_∞	L	T	ℓ_∞	L	T	ℓ_∞	L	T	L
16	16	10	1E-8	4	4.5	3E-7	5	2.5	3E-7	5	<0.5	2
128	32	9.8	4E-8	5	4.9	3E-7	6	2.8	3E-7	6	<0.5	2
1,024	64	10	1E-7	6	4.9	4E-7	7	3.0	4E-7	7	<0.5	3
8,192	128	10	3E-7	7	5.1	5E-7	8	3.1	5E-7	8	<0.5	3

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 ℓ_∞ of the error in u, "Solve" time "T" and the uniform refinement level "L", required to achieve single-precision (seven digits) accuracy or better. For the FFT we do not 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.*

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 were observed. However, what matters most is time to solution. Once this is factored in, the differences between the solvers become more pronounced.

We 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 this problem. 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. This is of course expected given the spectral accuracy of the FFT and the very law work per unknown required by the algorithm. The purpose of the experiment is to highlight the performance gap even if we use very high-order FMM or Galerkin methods.

But what about solutions with localized features as in test case 2? We report the "Solve" time $u = \exp(-(r/R)^\alpha)$ for Table 7 as a function of α. The larger the α the more localized features u has. The performance of the different solves is now quite different. FFT works well for relatively small values of α but as we make the solution sharper (larger α) it cannot resolve the length-scales of the solution and the uniform grid size shoots up; for the largest problem size FFT requires $100\times$ more unknowns than those required by FMM and GMG. Furthermore, FMM outperforms

p	α	GMG-5		FMM-14		FMMPHI-14		FFT	
		T	N	T	N	T	N	T	N
32	10	4.4	4.4E+6	0.5	2.9E+6	0.2	2.9E+6	0.4	1.7E+7
512	40	5.0	1.8E+7	0.7	6.4E+7	0.3	6.4E+7	2.3	1.1E+9
8,192	160	6.0	2.5E+8	0.8	1.1E+9	0.5	1.1E+9	11.8	6.9E+10
32,768	320	6.6	9.8E+9	1.8	4.3E+9	0.9	4.3E+9	30.7	5.5E+11

Table 7: *Time-to-solution for test case 2 (discontinuous field) performed on Stampede: Here α is a measure of the difficulty of resolving u. The higher the α 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 ℓ_∞ is less than $4E-7$. We used GMG-5, because in our test it was the fastest variant of GMG (see Tab. 8).*

p	q	N	T (AMG)	T (GMG)
32	4	8.9E6	29.3	8.6
32	5	4.4E6	27.7	4.4
32	6	2.1E6	22.8	4.9
32	8	6.0E5	51.4	6.1

Table 8: *Corresponding results in Tab. 7 with $\alpha = 10$ using AMG/GMG with different orders. As the order is increased up to 6th order, the cost per unknown reduces. However, further increase in the order increases this cost and thus the time to solution increases.*

multigrid but the difference is not as dramatic. Understanding this requires a lengthier analysis and is sensitive to the quality of per core performance. In passing, let us mention that the adaptive multigrid is quite efficiently implemented. Both codes used dense matrix-matrix multiplication kernels (that is the use the BLAS DGEMM routine) but their intensity is quite different. GMG uses a high-order tensor basis. The main computational kernel for GMG is the element traversal to apply the local stiffness matrix to a vector. This kernel is decomposed to 1D slices to reduce the complexity. The decomposition involves more integer operators and multiple DGEMM invocations. For near interactions FMM truncates the tensor basis and uses a single DGEMM of larger size than GMG for the near interaction. The far interaction is much more involved and briefly summarized in §2.2.

Due to the significance difference between the GMG and FMM kernels, the result is that their floating point performance per core differs. For our implementations FMM outperforms GMG. We also report AMG/GMG timings for different orders in Table 8. As expected AMG is significantly slower than GMG. We can see that the time to solution decreases by increasing the order up to $q = 5$, and then increases afterwards. Finally, we would like to remark that both GMG and FMM over-refine to maintain 2:1 balancing.

5. Conclusions. We outlined two solvers developed in our group for Poisson problems with high-order discretizations and we compared them for the Poisson problem with constant coefficients on the unit cube with periodic boundary conditions. We also consider two solvers developed by other groups, FFT and AMG. Below we summarize the conclusions of our study.

- If we consider just the cost per unknown, FFT is the fastest. FMM and in particular FMMPHI perform comparably with FFT for the same number of unknowns (on Stampede). Which method is preferable depends on the type of resolution scheme (uniform vs non-uniform). For this reason, one should note that it is not possible to compare these methods using exactly the same N. Often in the literature, in particular the HPC literature, such comparisons

appear. Without accuracy for a specific problem such information can be misleading.

- 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. We report results up to 230K cores that confirm this.
- For fields that have sharp local features 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.
- Low-order methods are hopelessly slow with respect high-order methods. The performance penalty due to excessive number of unknowns and suboptimal CPU performance can result in orders of magnitude in performance penalties. We expect this to hold for problems for which the solution has localized discontinuities.
- 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 quite high performance penalties.
- All three methods can be further improved: by integrating tightly with accelerators, further improving single-node performance, and trying to 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 $T_{\mathrm{FMM}}, T_{\mathrm{FFT}}, T_{\mathrm{GMG}}$ 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.

REFERENCES

[1] Satish Balay, Mark F. Adams, Jed Brown, Peter Brune, Kris Buschelman, Victor Eijkhout, William D. Gropp, Dinesh Kaushik, Matthew G. Knepley, Lois Curfman McInnes, Karl Rupp, Barry F. Smith, and Hong Zhang, *PETSc Web page.* http://www.mcs.anl.gov/petsc, 2014.

[2] Wolfgang Bangerth, Ralf Hartmann, and Guido Kanschat, *Deal.II – a general purpose object oriented finite element library*, ACM Transactions in Mathematical Software, 33 (2007), pp. 24/1–24/27.

[3] Abhinav Bhatele, Pritish Jetley, Hormozd Gahvari, Lukasz Wesolowski, William D Gropp, and Laxmikant Kale, *Architectural constraints to attain 1 exaflop/s for three scientific application classes*, in Parallel & Distributed Processing Symposium (IPDPS), 2011 IEEE International, IEEE, 2011, pp. 80–91.

[4] Achi Brandt, *Multi-level adaptive solutions to boundary-value problems*, Mathematics of Computation, 31 (1977), pp. 333–390.

[5] A. Brandt, *Multigrid adaptive solution of boundary value problems*, Mathematics of Computations, 13 (1977), pp. 333–390.

[6] Carsten Burstedde, Omar Ghattas, Michael Gurnis, Tobin Isaac, Georg Stadler, Tim Warburton, and Lucas C. Wilcox, *Extreme-scale AMR*, in SC10: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, ACM/IEEE, 2010.

[7] Carsten Burstedde, Lucas C. Wilcox, and Omar Ghattas, *p4est: Scalable algorithms for parallel adaptive mesh refinement on forests of octrees*, SIAM Journal on Scientific Computing, 33 (2011), pp. 1103–1133.

[8] J. Carrier, L. Greengard, and V. Rokhlin, *A fast adaptive multipole algorithm for particle simulations*, SIAM Journal on Scientific and Statistical Computing, 9 (1988), pp. 669–686.

[9] H. Cheng, Leslie Greengard, and Vladimir Rokhlin, *A fast adaptive multipole algorithm in three dimensions*, Journal of Computational Physics, 155 (1999), pp. 468–498.

[10] Kenneth Czechowski, Casey Battaglino, Chris McClanahan, Kartik Iyer, P-K Yeung, and Richard 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.

[11] M. O. Deville, P. F. Fischer, and E. H. Mund, *High-Order Methods for Incompressible Fluid Flow*, vol. 9 of Cambridge Monographs on Applied and Computational Mathematics, Cambridge University Press, 2002.

[12] Ron O Dror, J.P. Grossman, Kenneth M Mackenzie, Brian Towles, Edmond Chow, John K Salmon, Cliff Young, Joseph A Bank, Brannon Batson, Martin M Deneroff, Jeffrey S Kuskin, Richard H Larson, Mark A Moraes, and David E Shaw, *Exploiting 162-Nanosecond End-to-End Communication Latency on Anton*, in 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis, IEEE, Nov. 2010, pp. 1–12.

[13] Frank Ethridge and Leslie Greengard, *A new fast-multipole accelerated poisson solver in two dimensions*, SIAM Journal on Scientific Computing, 23 (2001), pp. 741–760.

[14] Ian T Foster and Patrick H Worley, *Parallel algorithms for the spectral transform method*, SIAM Journal on Scientific Computing, 18 (1997), pp. 806–837.

[15] Matteo Frigo and Steven G Johnson, *The design and implementation of fftw3*, Proceedings of the IEEE, 93 (2005), pp. 216–231.

[16] Hormozd Gahvari, Allison H Baker, Martin Schulz, Ulrike Meier Yang, Kirk E Jordan, and William Gropp, *Modeling the performance of an algebraic multigrid cycle on HPC platforms*, in Proceedings of the international conference on Supercomputing, ACM, 2011, pp. 172–181.

[17] Hormozd Gahvari and William Gropp, *An introductory exascale feasibility study for FFTs and multigrid*, in 2010 IEEE International Symposium on Parallel & Distributed Processing (IPDPS), IEEE, 2010, pp. 1–9.

[18] Michael W. Gee, Christopher M. Siefert, Jonathan J. Hu, Ray S. Tuminaro, and

Marzion G. Sala, *ML 5.0 smoothed aggregation user's guide*, Tech. Report SAND2006-2649, Sandia National Laboratories, 2006.

[19] Wolfgang Hackbusch, *Multigrid methods and applications*, vol. 4 of Springer Series in Computational Mathematics, Springer-Verlag, Berlin, 1985.

[20] Michael A. Heroux, Roscoe A. Bartlett, Vicki E. Howle, Robert J. Hoekstra, Jonathan J. Hu, Tamara G. Kolda, Richard B. Lehoucq, Kevin R. Long, Roger P. Pawlowski, Eric T. Phipps, Andrew G. Salinger, Heidi K. Thornquist, Ray S. Tuminaro, James M. Willenbring, Alan Williams, and Kendall S. Stanley, *An overview of the trilinos project*, ACM Trans. Math. Softw., 31 (2005), pp. 397–423.

[21] Harper Langston, Leslie Greengard, and Denis Zorin, *A free-space adaptive FMM-based PDE solver in three dimensions*, Communications in Applied Mathematics and Computational Science, 6 (2011), pp. 79–122.

[22] Ilya Lashuk, Aparna Chandramowlishwaran, Harper Langston, Tuan-Anh Nguyen, Rahul Sampath, Aashay Shringarpure, Richard Vuduc, Lexing Ying, Denis Zorin, and George Biros, *A massively parallel adaptive fast multipole method on heterogeneous architectures*, Communications of the ACM, 55 (2012), pp. 101–109.

[23] Xiaoye S. Li and James W. Demmel, *SuperLU_DIST: A Scalable Distributed-Memory Sparse Direct Solver for Unsymmetric Linear Systems*, ACM Transactions of Mathematical Software, 29 (2003), pp. 110–140.

[24] James W Lottes and Paul F Fischer, *Hybrid multigrid/Schwarz algorithms for the spectral element method*, Journal of Scientific Computing, 24 (2005), pp. 45–78.

[25] Dhairya Malhotra and George Biros, pvfmm*: A distributed memory fast multipole method for volume potentials*, 2014. preprint, padas.ices.utexas.edu/static/paper/pvfmm.pdf.

[26] Peter McCorquodale, Phillip Colella, Gregory T Balls, and Scott B Baden, *A local corrections algorithm for solving Poisson's equation in three dimensions*, Communications in Applied Mathematics and ComputationalScience, 2 (2006).

[27] Akira Nukada, Kento Sato, and Satoshi Matsuoka, *Scalable multi-GPU 3-d FFT for TSUBAME 2.0 supercomputer*, in 2012 International Conference for High Performance Computing, Networking, Storage and Analysis (SC), IEEE, 2012, pp. 1–10.

[28] Dmitry Pekurovsky, *P3DFFT: A framework for parallel computations of Fourier transforms in three dimensions*, SIAM Journal on Scientific Computing, 34 (2012), pp. C192–C209.

[29] James C. Phillips, Gengbin Zheng, Sameer Kumar, and Laxmikant V. Kalé, *NAMD: Biomolecular simulation on thousands of processors*, in Proceedings of Supercomputing, The SCxy Conference series, Baltimore, Maryland, November 2002, ACM/IEEE.

[30] Joel R. Phillips and Jacob K. White, *A precorrected-FFT method for electorstatic analysis of complicated 3-D structures*, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 16 (1997), pp. 1059–1072.

[31] Rahul S. Sampath and George Biros, *A parallel geometric multigrid method for finite elements on octree meshes*, SIAM Journal on Scientific Computing, 32 (2010), pp. 1361–1392.

[32] IF Sbalzarini, JH Walther, M. Bergdorf, SE Hieber, EM Kotsalis, and P. Koumoutsakos, *PPM–A highly efficient parallel particle–mesh library for the simulation of continuum systems*, Journal of Computational Physics, 215 (2006), pp. 566–588.

[33] Hari Sundar, George Biros, Carsten Burstedde, Johann Rudi, Omar Ghattas, and Georg Stadler, *Parallel geometric-algebraic multigrid on unstructured forests of octrees*, in Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, IEEE Computer Society Press, 2012, p. 43.

[34] Hari Sundar, Rahul S. Sampath, and George Biros, *Bottom-up construction and 2:1 balance refinement of linear octrees in parallel*, SIAM Journal on Scientific Computing, 30 (2008), pp. 2675–2708.

[35] Hari Sundar, Georg Stadler, and George Biros, *Comparison of multigrid algorithms for high-order continuous finite element discretizations*, Submitted for publication, (2014).

[36] Paul N Swarztrauber, *Multiprocessor FFTs*, Parallel computing, 5 (1987), pp. 197–210.

[37] Roland A Sweet, William L Briggs, Suely Oliveira, Jules L Porsche, and Tom Turnbull, *FFTs and three-dimensional poisson solvers for hypercubes*, Parallel computing, 17 (1991), pp. 121–131.

[38] O. Tatebe and Y. Oyanagi, *Efficient implementation of the multigrid preconditioned conjugate gradient method on distributed memory machines*, in Supercomputing'94. Proceedings, IEEE, 1994, pp. 194–203.

[39] Trottenberg, U. and Oosterlee, C. W. and Schuller, A., *Multigrid*, Academic Press Inc., San Diego, CA, 2001.

[40] Raymond S Tuminaro, Erik G Boman, Jonathan J Hu, Andrey Prokopenko, Christopher Siefert, Paul H Tsuji, Jeremie Gaidamour, Luke Olson, Jacob Schroder,

Badri Hiriyur, et al., *Toward flexible scalable algebraic multigrid solvers.*, tech. report, Sandia National Laboratories Albuquerque, NM; Sandia National Laboratories (SNL-CA), Livermore, CA (United States), 2013.

[41] Lexing Ying, George Biros, and Denis Zorin, *A kernel-independent adaptive fast multipole method in two and three dimensions*, Journal of Computational Physics, 196 (2004), pp. 591–626.

[42] Rio Yokota and Lorena A Barba, *A tuned and scalable fast multipole method as a preeminent algorithm for exascale systems*, International Journal of High Performance Computing Applications, 26 (2012), pp. 337–346.