A Nested Partitioning Algorithm for Adaptive Meshes on Heterogeneous Clusters

Hari Sundar
University of Utah
Salt Lake City, UT 84112
hari@cs.utah.edu

Omar Ghattas
University of Texas at Austin
Austin, TX 78712
omar@ices.utexas.edu

ABSTRACT

In the era of the accelerator, load balancing strategies that are well-understood for traditional homogeneous supercomputers must be re-worked in order to address the problem of distributing work across heterogeneous hardware such that neither the CPU nor the accelerator is left idle. Whereas partitioning for a homogeneous system need only balance node-level workload against single-node performance, partitioning for an accelerator-enabled system requires a nested partitioning scheme that ensures both an optimal intra-node and inter-node load balance. We refer to this as *enclave partitioning*. Our parallelization scheme allows the same shared-memory-level code to be used on both the CPU and the accelerator, and also allows inter-node communication code to be reused during CPU-accelerator communication. This is in contrast to the traditional “offload” model, in which accelerator code can differ significantly from CPU code in both form and programming language. Using a hybrid MPI-OpenMP implementation of the acoustic-elastic wave propagation, we demonstrate the efficacy of the proposed partitioning scheme on a heterogeneous, Intel[®] Xeon Phi[™]-accelerated supercomputer (Stampede). With our approach we have realized speedups of up to 5.78x using a 7th order discretization and 6.88x for 15th order discretization relative to a baseline, pure-MPI implementation. We present strong and weak scaling results as well as individual node performance to illustrate the benefits and limits of the accelerator-enabled scientific computing.

Categories and Subject Descriptors

D.1.3 [Concurrent Programming]: Distributed programming; C.1.4 [Parallel Architectures]: Distributed Architectures

Keywords

accelerator; offload; mic; heterogeneous; partitioning; wave equation; pde; hybrid parallelism;

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

ICS'15, June 8–11, 2015, Newport Beach, CA, USA.

Copyright © 2015 ACM 978-1-4503-3559-1/15/06 ...\$15.00.

DOI: <http://dx.doi.org/10.1145/2751205.2751246>.

1. INTRODUCTION

The trend in modern supercomputing is to move toward systems that realize the bulk of their computational power from accelerators [11]. According to the Nov 2014 version of the top500 list [18], 75 of the clusters on the list (15%) had accelerators and this number has steadily been rising in the last few years. On most of these machines the majority of the flops are available from the accelerator. For example on Stampede at the Texas Advanced Computing Center (TACC), 2.2PetaFlop/s is the peak performance of the base cluster while an additional 7.4PF/s is available from the use of the Intel[®] Xeon Phi[™] accelerator available on its 6400 nodes. Additionally, while the peak performance is usually 3-4 times higher for the accelerator, the total available memory is smaller on the accelerator by a similar factor. The situation is similar for other top machines utilizing accelerators. To make effective use of such machines, application codes need to parallelize across 4 levels; *a)* across distributed memory nodes, *b)* across the CPU and the accelerator, *c)* shared-memory parallelism with the CPU/accelerator, and *d)* vectorization at the finest level. The asymmetry in performance as well as memory at the cpu-accelerator makes this particularly challenging for complex applications.

Load balancing strategies that are well-understood for traditional homogeneous supercomputers must be re-worked in order to address the problem of distributing work across heterogeneous hardware such that neither the CPU nor the accelerator is left idle. A common paradigm for accelerator computing is to offload computationally intensive tasks to the accelerator. In many such cases, the CPU is left relatively idle. Additionally, this requires $\mathcal{O}(n)$ data transfer between the CPU and the accelerator. Given that the accelerators have less memory available, such an approach is inefficient unless the operations being offloaded are of extremely high computational efficiency. Instead, we believe it is preferable to treat the accelerator the same as any other compute node, reducing the data transfer costs to $\mathcal{O}(n^{2/3})$ for 3D problems. This allows programmers to use a Single-Program, Multiple-Data (SPMD) programming paradigm, which is much easier than programming separate logic for the CPU and the accelerator. This is particularly attractive on clusters such as Stampede and Tianhe-2 using the Intel[®] Xeon Phi[™] as the accelerator, as it is based on the x86 architecture as the CPU. This means that on such clusters, we can have true SPMD code¹.

¹there are still minor differences, but this statement is for the most part true.

Figure 1: Enclave partition: The computational domain is first partitioned among MPI processes, shown here separated and in different colors. Each MPI process then partitions its subdomain into a set of elements to assign to the CPU (transparent) and a set of elements to assign to the accelerator.

The key *challenge* for the equivalent treatment of CPUs and accelerators—besides the differences in performance and memory—is that the communication cost from an accelerator to any other process other than its host node is significantly higher. In this regards, they are not the same as the CPUs and therefore when we divide work across the nodes, and between the CPU and the accelerator, we need to partition such that the accelerators only require data transfers with their hosts. In other words, the task graph for the problem is such that the tasks assigned to any accelerator have data-dependencies only with their host CPUs. A novel $\mathcal{O}(n)$, node-local partitioning algorithm that facilitates such a data-dependency is the main contribution of this work. We call this *enclave partitioning* and an example of such a partition is shown in Figure 1. The partitioning improves the overall performance of the system by ensuring proper utilization of both the CPU and the accelerator, and also makes it easier to program for such systems by using a SPMD programming paradigm. Using a hybrid MPI-OpenMP implementation of the acoustic-elastic wave propagation, we demonstrate the efficacy of the proposed partitioning scheme on Stampede. Using our approach, we have realized speedups of up to 5.78x using a 7th order discretization and 6.88x for 15th order discretization relative to a baseline, pure-MPI implementation. We present strong and weak scaling results as well as individual node performance to illustrate the benefits and limits of the coprocessor-accelerated scientific computing. To the best of our knowledge, this is the first work considering such a partition that enables SPMD programming for heterogeneous clusters.

Organization of this paper:

The rest of this paper is organized as follows. In §2, we present the Seismic wave propagation problem—our test application—as it is important to understand the computational structure of the application to appreciate the algorithmic choices. In §3 we provide a baseline analysis that we performed prior to this work, that led to the development of the current work. In §4, we present the enclave partitioning algorithm and in §5 we discuss the implementation details of our algorithms. Finally, in §6 we present experiments and scalability results in support of enclave partitioning and conclude with directions for future research.

Figure 2: (a) Nodes on the faces of adjacent elements are collocated. (b) Fluxes on virtual face nodes get "lifted" back to the elements. (c) Nodes from parent faces are interpolated to four virtual hanging faces on the facemesh.

2. SEISMIC WAVE PROPAGATION

We are primarily concerned with accelerating the solution of the acoustic-elastic wave equations to simulate the propagation of seismic waves in coupled acoustic-elastic media. Our long-term goal is the solution of the global-scale inverse problem of determining the interior structure of the earth using full seismic waveform inversion [20]. Each step of the inverse problem requires a full, explicit forward-in-time solution of the acoustic-elastic wave equations. As such, a moderate speedup in the forward solution translates to a significant speedup in the inverse solution [5].

For isotropic media, seismic wave propagation can be described in velocity-strain form by the first-order system [10]

$$\rho \frac{\partial \mathbf{v}}{\partial t} = \nabla \cdot (\mathbf{C}\mathbf{E}) + \mathbf{f} = \nabla \cdot (\lambda \text{tr}(\mathbf{E})\mathbf{I} + 2\mu\mathbf{E}) + \mathbf{f} \quad (1a)$$

$$\frac{\partial \mathbf{E}}{\partial t} = \frac{1}{2}(\nabla \mathbf{v} + \nabla \mathbf{v}^T) \quad (1b)$$

where we make use of the Lamé parameters λ and μ as our material parameters. This formulation allows us to describe wave propagation in an acoustic medium by simply setting μ to zero [20]. To make the system well-posed, we assume initial conditions $\mathbf{v}(\mathbf{x}, 0) = \mathbf{v}_0(\mathbf{x})$ and $\mathbf{E}(\mathbf{x}, 0) = \mathbf{E}_0(\mathbf{x})$ on the problem domain B , and boundary condition $\mathbf{S}\mathbf{n} = \mathbf{t}_{bc}$ on the boundary of the domain, ∂B , where we prescribe the traction $\mathbf{t}_{bc}$, $\mathbf{n}$ is the outward unit normal vector at the boundary, and $\mathbf{S} = \mathbf{C}\mathbf{E}$ is the Cauchy stress tensor. At an interface Γ between two elastic media, we impose continuity in the velocity and the traction

$$\mathbf{v}^+ = \mathbf{v}^- \quad \mathbf{x} \in \Gamma \quad (2a)$$

$$\mathbf{S}^+ \mathbf{n} = \mathbf{S}^- \mathbf{n} \quad \mathbf{x} \in \Gamma \quad (2b)$$

whereas at an interface between an elastic medium and an acoustic medium or between two acoustic media we impose continuity between the normal component of velocity and

the traction

$$\mathbf{n} \cdot \mathbf{v}^+ = \mathbf{n} \cdot \mathbf{v}^- \quad \mathbf{x} \in \Gamma \quad (3a)$$

$$\mathbf{S}^+ \mathbf{n} = \mathbf{S}^- \mathbf{n} \quad \mathbf{x} \in \Gamma \quad (3b)$$

Additional details on the mathematical formulation of the problem can be found in [20].

2.1 Discontinuous Galerkin Method

The discontinuous Galerkin (DG) finite element method is characterized by its use of discontinuous basis functions, which allow the methods to operate on h -adaptive, nonconforming meshes [8]. Global solution continuity is enforced weakly on element boundaries through the use of a numerical flux. This makes most DG operators highly local. In this paper we will present only a cursory examination of the features of DG that are relevant to an optimal implementation on a heterogeneous system. For a more detailed treatment of the solution of the acoustic-elastic wave equations using a DG method, see [20], [13], [4].

Our DG formulation uses an h -adaptive, possibly nonconforming, mesh of hexahedral curvilinear elements, which we require conforms to a 2:1 balance. A 2:1 balance indicates that only 1:1 and 2:1 size relations are allowed between neighboring elements. Following [8], the basis of each element is composed of tensor products of 1-D Lagrange polynomials. Numerical integration points are chosen to coincide with the basis function nodes, which we have chosen to place at the Legendre-Gauss-Lobatto points (again following [8]). Each element, then, can be conceptualized as a collection of nodes. Element face nodes collocate when elements are adjacent to each other (Figure 2a). At the beginning of each iteration, we isolate the nodes on the faces of the elements and copy them to the “facemesh”. We call a face that is adjacent to four faces from neighboring elements a “parent face”, and the faces of the smaller neighboring elements “hanging faces”. When transferring face data to the facemesh, we interpolate the data from a parent face to four corresponding virtual hanging faces in the face mesh. This ensures that neighboring faces always conform to one another in the facemesh (Figure 2c).

The key kernels used in the DG formulation of the acoustic-elastic wave equations are shown in Table 1. The “Locality” column in the table describes the data domain on which each kernel operates. Element- and face-local kernels can operate on each element independently, while face-global kernels require nodal values from multiple neighboring elements. In this sense of the word “global”, the only global kernel in our DG formulation is the *Lift* kernel, which applies the fluxes from adjacent faces to the volume nodes of the elements to which they correspond (Figure 2b).

Table 1: Key kernels used in DG solution of acoustic-elastic wave equations.

Kernel	Locality	FLOPS/MOPS
<i>Interpolate</i> $\mathbf{q} \rightarrow \mathbf{qf}$	Face-local	$[0, 4/15(2N + 1)]$
<i>Derivatives</i>	Element-local	$3/8(2N + 1)$
<i>Numerical Flux</i>	Face-local	5.27
<i>Project</i> $\mathbf{qf} \rightarrow \mathbf{q}$	Face-local	$N + 7/4$
<i>Lift</i> flux to volume	Face-global	0.90
<i>Runge-Kutta</i>	Element-local	1.25

Figure 3: Partitioning between MPI processes is done by segmentation along the Morton curve. The mesh on the left is shown with the space-filling Morton curve shown in blue connecting the elements. Partitioning can be thought of as untangling this curve and laying it out in one dimension, as in the color-coded curve at the bottom of the figure. The colors represent the different processes to which these elements are assigned.

2.2 Adaptive Meshing

In this section, we briefly describe our adaptive meshing framework. Octrees are axis-aligned space partitions in 3D. The root of an octree can be associated with a topologically equivalent 3D domain that can be mapped smoothly to a cube. A forest is a collection of octrees, which are connected conformingly through faces, edges and corners [6]. Such a forest of octrees allows for a two-tier decomposition of geometric domains, where the first tier, the macromesh, is an unstructured mesh of conforming hexahedral elements (possibly with curved boundaries). Recursive octree subdivision then allows for efficient octree-based adaptivity of the mesh in the geometric domain. Localized refinement generally creates non-conforming meshes, which can still be used to represent continuous finite element solutions by incorporating algebraic constraints at the element level (hanging nodes).

The adaptive octree structure can naturally be interpreted to define a recursive space filling curve that traverses all octants in all trees in the so-called Morton- or z -order [12]. This curve provides the basis for indexing and fast partitioning of octants between processes, a fact that we exploit while assigning work across nodes.

3. BASELINE ANALYSIS

We first performed baseline analysis to estimate what changes would be necessary to achieve good performance on a heterogeneous cluster like Stampede. All test runs in this paper are made up of 300 time steps of the solution of a wave propagation through a domain with a material discontinuity in the center of the domain. A time-varying boundary strain is applied at the lowest z -face of the domain, generating compressional waves that travel toward the material discontinuity, at which they are partially reflected and partially refracted according to Snell’s law (Figure 4). The mesh was refined according to two Gaussian density functions centered at arbitrarily selected points within the domain. Though meshes with varying numbers of elements were generated by specifying different minimum allowable element sizes, the under-

Figure 4: Compressional strain in the x-direction. Our test problem is a time-varying wave propagation through a medium with a central material discontinuity.

lying Gaussian densities were the same for all runs. One such mesh is shown in Figure 1.

The fraction of execution time taken by each of the key kernels discussed in the previous section was measured for the baseline code, the results are shown in Figure 5. As can be seen, the derivatives kernel accounts for almost half the execution time. This would normally point to a task-offloading approach, wherein the derivative computation is offloaded to the Xeon Phi, while the CPU takes care of the rest. Unfortunately, given that the derivatives need to be computed on the volume unknowns, the overall speedup factoring-in the cost of transferring data back and forth to the accelerator was $< 10\%$. Equating this with communication costs normally observed on homogeneous clusters, one can see that a similar setup where the communication is proportional to the surface-area instead of the volume would be highly beneficial. As mentioned earlier, it would have the added benefit of simplifying the programming by allowing us to adopt the SPMD programming paradigm. In the next section, we expand on this idea and introduce our partitioning scheme.

4. PARTITIONING

As previously discussed, in the discontinuous Galerkin finite element method, inter-process communication consists entirely of the transfer of shared element faces. Thus, the partitioning of the global domain into processor-local domains directly controls the amount of communication required. If partitions are created such that the number of faces shared amongst processors is minimized, then communication will also be minimized. Partitioning must take place at two levels—first, the inter-node partitioning is done using the Morton ordering of the elements to ensure load-balance at the node level. For some total number of elements K and compute nodes p , we assign K/p elements to each node with node- i receiving elements $\mathcal{E} = [(i-1)K/p, iK/p]$ according to the Morton order. We assign one MPI process to each compute node, and so each MPI process owns a subdomain of contiguous² elements. This domain is then partitioned—locally on each node by the CPU—into two domains such

²in the Morton order

Figure 5: Fraction of execution time for key kernels in the solution of the acoustic-elastic wave propagation problem for a fixed problem size and varying number of compute nodes. All runs were completed using a fixed number of 7th order elements.

that one domain is an *enclave*³ within the other. We refer to this as *enclave partitioning*.

This approach differs from the way in which accelerators have traditionally been used. A common paradigm for accelerators computing is to offload whole computationally intensive tasks to the accelerator. For example, in the solution of the acoustic-elastic wave equation, one might offload the *Derivatives* kernel to the accelerator while the CPU computes the *Flux* kernel. While the *Derivatives* kernel is certainly computationally intensive, and maps well to accelerator architectures, the amount of data that must be transferred at each timestep is proportional to the total number of elements K owned by the compute node, times the number of degrees of freedom per element $(N+1)^3$. In contrast, transferring only a subset K_{MIC} of elements to the accelerator and transferring only shared face data, assuming that K_{MIC} is structured such that inter-facial area is minimized, is proportional to $6K_{MIC}^{2/3}(N+1)^2$. The tradeoff is that, although less data is transferred, we must ensure the entire timestep runs optimally on the Xeon PhiTM, rather than just one or two computationally-intensive kernels. As mentioned earlier, this supports the SPMD paradigm, which is a huge advantage when the accelerator and CPU share the same architectural basis, as is the case for Stampede. While this can be an issue with using say GPUs with x86 CPUs, where it might be easier to take an incremental approach of only porting a single kernel, we believe that the SPMD approach is better and more scalable and maintainable in the long run.

The approach proposed in this paper is to treat the relationship between the host and offload processes in much the same way as the relationship between compute nodes is treated. Once an enclave has been identified within the node's domain, the enclaved domain is assigned to the accelerator and therefore only needs to communicate with the host CPU. Note that due to the nature of the partition, the inter-node communication patterns are identical to the case when no accelerator is used. This is important as we do not pay any penalty for the use of accelerators. Each host ini-

³enclave: a region completely surrounded by a different territory.

tializes its accelerator, then launches an offload process that runs each timestep in parallel with the host. Synchronization is only required once per time step, when the CPU and accelerator exchange their shared face data. At the end of the computation, all of the degrees of freedom computed on the accelerator are transferred back to the host, and so we incur the cost of transferring large datasets only once.

Given the different computational capabilities of the CPU and the accelerator, enclave partitioning has to be able to produce enclaves of different sizes such that the ratio of the number of elements in the enclave to those surrounding it is similar to the relative performance of the accelerator to that of the CPU on the problem(s) of interest. We now explain our algorithm for enclave partitioning.

4.1 Enclave Partitioning

Our goal is to extract a specified number of elements $\mathcal{E}_\phi \subset \mathcal{E}$, where $\mathcal{E}_\phi$ is the set of elements owned by the accelerator and $\mathcal{E}$ is the initial subdomain assigned to a given node. The goal of enclave partitioning is to determine $\mathcal{E}_\phi$ such that it does not share any faces with elements $\notin \mathcal{E}$. We would also like to minimize the number of faces shared between $\mathcal{E}_\phi$ and $\mathcal{E} \setminus \mathcal{E}_\phi$ to minimize the cost of exchanging data between the CPU and the accelerator. It is important to note that it might not be possible in every case to find an enclave of the specified size satisfying these criteria. Specifically, $\mathcal{E}_\phi$ is by definition made up of interior elements, and if the number of elements in $\mathcal{E}$ is small, then it is entirely possible that the number of interior elements is less than requested or even zero. For real applications however, we expect to have significant number of elements in $\mathcal{E}$ such that it will be possible to compute $\mathcal{E}_\phi$.

The algorithm first extracts the boundary elements $\mathcal{E}_b$, i.e., elements that share at least one face with elements $\notin \mathcal{E}$. A simple version of the algorithm can then simply select the specified number of elements from $\mathcal{E} \setminus \mathcal{E}_b$ to belong to $\mathcal{E}_\phi$. This does not however minimize the number of faces shared between $\mathcal{E}_\phi$ and $\mathcal{E} \setminus \mathcal{E}_\phi$. Heuristically, we can minimize the number of shared faces by ensuring that the boundary is largely a smooth face and not made of jagged faces. For our octree-based meshes, this implies selecting sets of elements that are contained within *coarser* octants. Following this heuristic, we construct a coarse interior region, $\mathcal{E}_{coarse}$ by constructing the minimal linear octree between successive elements $\in \mathcal{E}_b$ using Algorithm 1 (see [17] for a detailed discussion). This algorithm runs in $\mathcal{O}(nl)$, where l is the number of levels used in the octree (practically a constant) and n is the number of elements in the boundary.

Initializing $\mathcal{E}_\phi$ with the elements $\in \mathcal{E}$ contained within the “largest” octant in $\mathcal{E}_{coarse}$, we keep adding the “largest” adjacent (next or previous octant in the Morton order) octants till $\mathcal{E}_\phi$ contains the desired number of elements. When we speak of a “large” coarse octant $\mathcal{O}$, we are not referring to the geometric size of the octant, but the number of octants in $\mathcal{E}$ that are overlapped by $\mathcal{O}$. Generating the coarse interior region helps us in two ways. First, it attempts to minimize the effective number of shared faces between the CPU and the Xeon PhiTM. Secondly, while adding additional elements to $\mathcal{E}_\phi$, $\mathcal{E}_{coarse}$ acts as a skip list and speeds up the partitioning.

Determining the size of the enclave.

In order to determine the optimal number of elements to offload to the accelerator, we estimate the accelerator run-

Algorithm 1. *Constructing a minimal linear octree between two octants (sequential) - CompleteRegion*

Input: Two octants, a and $b > a$.
Output: R , the minimal octree between a and b .
Work: $\mathcal{O}(n \log n)$, where $n = \text{len}(R)$.
Storage: $\mathcal{O}(n)$, where $n = \text{len}(R)$.

```

1.  $W \leftarrow \mathcal{C}(\mathcal{A}_{finest}(a, b))$ 
2. for each  $w \in W$ 
3.   if  $(a < w < b)$  AND  $(w \notin \{\mathcal{A}(b)\})$ 
4.      $R \leftarrow R + w$ 
5.   else if  $(w \in \{\mathcal{A}(a)\}, \{\mathcal{A}(b)\})$ 
6.      $W \leftarrow W - w + \mathcal{C}(w)$ 
7.   end if
8. end for
9. Sort ( $R$ )

```

time, CPU runtime, and PCI-bus data transfer time for a given set of partition parameters, such as the number of elements on the CPU and the accelerator, the number of faces shared between the CPU and the accelerator, and the order of the elements. We can calculate the number of FLOPs that each key kernel will execute for the given partition parameters, then use measured FLOP rates from each of the kernels (gathered under a variety of partitions and element orders) to estimate the per-timestep runtime of the kernels for the theoretical partition. We select the number of elements to offload by calculating at which point the estimated accelerator runtime plus the PCI-bus data transfer time is equal to the estimated CPU runtime, under the assumption that the global domain is a perfect cube of uniform elements, and that any accelerator subdomain will also be a cube of uniform elements. The resulting number of elements to offload K_{MIC} is then passed as an input to our intra-node partitioning algorithm.

5. IMPLEMENTATION

In this section we describe the parallelization at the two lowest levels of parallelism shown in Figure ??—shared memory parallelism using OpenMP and vectorization. While several of these optimizations were done to improve the performance on the Intel[®] Xeon PhiTM, they also helped improve the performance of the code on the CPU. This is another big advantage of the SPMD paradigm.

5.1 Vectorization

Several authors have identified vectorization as a key component in achieving good performance on the Xeon PhiTM [16] [14] [21] [15]. Indeed, the 512-bit wide SIMD vector processing capabilities of the Xeon PhiTM allow for up to 16 DP FLOPS per clock cycle per core (for a fused multiply-add). Code which is not vectorizable, then, suffers a significant performance hit relative to vectorized code. There are several approaches to vectorization, including auto-parallelization by the Intel[®] compiler, the use of Intel[®] CilkTM Plus directives [1], and hand-coding using vector intrinsics or assembly. For each of the key kernels presented in Table 1 we implemented each of these vectorization strategies, though only the results for the *Derivative* kernel will be presented here.

Table 2: FLOP rates in GFLOPS for the CPU and Xeon Phi™ 7th order *Derivate* kernels for various vectorization methods. 16 OpenMP threads were used across the two Xeon™ sockets on the CPU side, and 120 OpenMP threads were used on the Xeon Phi™.

Vectorization	CPU	Xeon Phi™
No vector	53.8	10.9
Compiler	74.7	39.6
Cilk	126.9	161.8
Hand	205.6	159.4

For all other kernels there was no significant performance difference between the three vectorization methods.

The *Derivative* kernel calculates 27 tensor products, producing the partial derivatives of the nine field variables in the three canonical directions. There are three tensor application kernels, one for each canonical direction, and for order N elements the operation of each tensor kernel is analogous to performing $N + 1$ matrix-matrix multiplications of size $(N + 1) \times (N + 1)$. This is the reason that we choose 7th and 15th order elements, to allow the matrices to be 8×8 and 16×16 , enabling better vectorization. Though many optimized *dgemm* kernels are available [19], [7], including the Phi-optimized Intel® Math Kernel Library [2], these kernels achieve high performance only for matrices that are relatively large compared to the amount of data available in registers and the L1 cache. Our problem deals with much smaller data structures, however, as for, e.g., 7th order elements, four $(N + 1) \times (N + 1)$ matrices can fit in the vector registers of a single Xeon Phi™ core. A full examination of our hand-optimized tensor kernels exceeds the scope of this paper, and so we will simply say that our optimization strategy was to minimize loads and stores between registers and the cache, perform all arithmetic operations in the VPU, and amortize loads and stores by reordering instructions such that arithmetic operations could be executed concurrent to memory transfers. For the CPU, we observed a significant speedup over both compiler-vectorized and Cilk™ Plus-vectorized code, as Table 2 shows. On the Xeon Phi™, however, although our vectorized code performs significantly better than unvectorized code, we were not able to produce performance significantly better than the Cilk™ Plus-vectorized code.

Anomalously, the only vectorization method for which we observed higher performance on the Xeon Phi™ than on the CPU was the Cilk™ Plus-vectorized kernel. We expect the Xeon Phi™ to outperform the CPU by a factor of three, and are currently investigating the reasons for the weaker-than-expected performance on the Xeon Phi™.

5.2 Multithreading and Offload

Currently, OpenMP is the method of choice for making full use of the many hardware threads available on the Xeon Phi™ [9]. While MPI libraries are available on the Xeon Phi™, treating the Xeon Phi™ as a distributed memory system would give each core only 134 MB of RAM. When we consider that each Xeon Phi™ core can only issue a vector instruction every other clock cycle, which necessitates the use of at least two hardware threads per core, we see that

using MPI to keep the Xeon Phi™ fully scheduled would lead to only 67 MB of RAM per MPI process.

With the decision to use OpenMP on the Xeon Phi™ firmly in place, we are faced with the remaining decisions of how to implement parallelism on the CPU and how to implement communication between the CPU and the Xeon Phi™. The existing codebase we have inherited for the solution of the acoustic-elastic wave equations utilizes pure MPI for interprocess communication and parallelization, spawning one MPI process per core on each compute node. On Stampede this translates to 16 MPI processes per compute node, each process having its own address space, arrays of unknowns, and computational subdomain. At each time step, facial degrees of freedom on faces that fall on inter-processor boundaries are communicated between MPI processes. An obvious option is to maintain this distributed-memory parallelism on the CPU side of the compute nodes and launch a single MPI process on the Xeon Phi™ that then spawns OpenMP threads. This would require an overhaul of our existing MPI scheduling algorithm, however, which assigns a roughly equal amount of work to each MPI process. It would also mean that the computational subdomain owned by the Xeon Phi™ could have a potentially large number of neighboring subdomains, which would lead to a large amount of data being transferred over the PCI bus and subsequently over the InfiniBand network. The worst case data path would be a Xeon Phi™ whose subdomain was neighbored by a subdomain owned by a Xeon Phi™ on another compute node, in which case the shared faces would have to travel across the PCI bus, then between nodes via Infiniband, then back across the PCI bus after arrival at the destination node.

To reduce traffic on the PCI bus and simplify load balancing, then, we choose to assign only one MPI process to the CPU side of each compute node, which then spawns OpenMP threads on the CPU-side to take advantage of the multiple Xeon™ cores. This same MPI process also controls offload to the Xeon Phi™ and MPI communication. We restrict the subdomain assigned to the Xeon Phi™ to a subset of the elements owned by the parent MPI process whose faces do not lie on inter-processor boundaries. This eliminates the need for a reworking of our MPI load balancing algorithm, as each compute node can be given the same amount of work and the partitioning between the CPU and the Xeon Phi™ domains can be done independently and locally on each compute node.

SPMD programming differences.

Though each host MPI process does technically “offload” in the sense that it initiates the execution of the Xeon Phi™ code through offload compiler directives, the behavior of the Xeon Phi™ and the dynamic between the CPU and the Xeon Phi™ is more like that of a distinct compute node. Each Xeon Phi™ owns a set of elements forming a computational subdomain and executes full time steps in the solution of the acoustic-elastic wave equation. The only difference between the CPU and Xeon Phi™ codes lies in the communication phases of each time step. The CPU maintains communication buffers and maps to inter-processor boundary faces for both inter-node MPI communication and intra-node Xeon Phi™ communication, whereas the Xeon Phi™ concerns itself only with faces along the CPU-Xeon Phi™ boundary. CPU-CPU communication is carried out asyn-

Table 3: Key performance characteristics of Stampede compute node.

	Xeon Phi™	2x Xeon E5
Peak DP GFLOPS	1070	346
Peak Bandwidth (GB/s)	320	102.4
Memory (GB)	8 GDDR5	32 DDR3

Figure 6: Incremental optimization of 7th order code.

chronously via MPI, while CPU-Xeon Phi™ communication is implemented synchronously through compiler offload directives. Although the Xeon Phi™ is capable of asynchronous computation, and theoretically capable of asynchronous data transfer, we have not observed a significant time difference in data transferred between the CPU and the Xeon Phi™ when using asynchronous offload directives versus synchronous directives. We postulate that this is due to a fixed overhead in the offload runtime that, though possibly amortized by actual asynchronous transfer for larger array sizes, is significant enough for the amount of face data we need to transfer to be very nearly the same as the time required to synchronously transfer that data.

6. RESULTS

We present results in support of the proposed partitioning algorithm in this section. All experiments were carried out on Stampede at the Texas Advanced Computing Center (TACC). Stampede consists of 6400 compute nodes, each of which houses two 8-core Intel® Xeon™ E5 CPUs and up to two 61-core Intel® Xeon Phi™ SE10P accelerators [3]. As table 3 shows, in terms of floating point performance and memory bandwidth the Xeon Phi™ outperforms the CPU by approximately a factor of three. As with most accelerators, the amount of work that can be done by the Xeon Phi™ is constrained by its limited amount of random-access memory. To begin with, the Xeon Phi™ has only 25% as much RAM as is available to its host-node CPUs. Considering the Xeon Phi™’s higher core count the situation becomes worse, as there are approximately 134 MB of RAM for every Xeon Phi™ core, compared to 2 GB of RAM for each CPU core. Thus, the Xeon Phi™ is most effectively used as a large SMP node, programmed using a shared-memory paradigm.

Figure 7: Incremental optimization of 15th order code.

6.1 Single-Node Optimization and Offloading

We first present the results of each step in the transformation from our baseline pure-MPI code to our vectorized, hybrid MPI-OpenMP, coprocessor-accelerated code. Figures 6 and 7 show the speedup realized after each subsequent optimization step relative to the baseline MPI-only code run on a single socket of Stampede. Some compute nodes on Stampede contain two Xeon Phi™. Our code seamlessly handles the utilization of two Xeon Phi™ per compute node by simply scheduling two MPI processes per node, each pinned to a different socket. Each of these MPI processes then spawns 8 OpenMP threads on the 8 cores of their host socket and controls offloading to one of the two available Xeon Phi™. Though we expected a performance hit when moving from pure MPI code to hybrid MPI-OpenMP code, we did not observe any difference in runtime between our OpenMP and pure MPI codes. We took care in our OpenMP implementation to minimize memory sharing amongst threads as much as possible so as to mimic the distributed-memory behavior of MPI processes.

6.2 Partitioning

To measure the consistency of the partitions produced by our partitioning algorithm we examine the distribution of Xeon Phi™ runtimes across a large run. Examining Figure 8 we see that most of the Xeon Phi™ runtimes are within 5% of the mean runtime. We believe this is a very good result and within acceptable bounds given the variable nature of adaptive meshes. While it might be possible to get a more uniform load, the simplicity of the current algorithm makes it very efficient.

6.3 Scaling Results

The difficulty in achieving strong scalability using coprocessor-enabled code is that as the number of compute nodes is increased for a fixed number of elements, the ratio of boundary to interior elements on each node’s subdomain also increases. Since only interior elements are assigned to the Xeon Phi™, this means that the percentage of elements eligible for offload to the accelerator decreases as the number of compute nodes increases. At very high compute node counts, the performance of the Phi-accelerated code approaches the performance of the baseline code due to the increased ratio of work

Table 4: Speedup of optimized, accelerated strong scaling runs relative to baseline MPI.

Cores	PHIs	7th Order	15th Order
32	4	5.78x	6.88x
64	8	5.46x	4.51x
128	16	4.08x	4.15x
256	32	3.44x	3.44x
512	64	2.75x	2.75x
1024	128	2.33x	1.89x
2048	256	1.57x	

being assigned to the CPU. This behavior is illustrated in Figures 9 and 11.

We observed weak scaling comparable to or better than our baseline pure-MPI code. For weak scaling runs using dual-Phi compute nodes, the size of our experiments were constrained by the available number of compute nodes with two Phis. We present dual-Phi weak scaling for up to as many nodes as were available. Table 4 shows the speedup factor for our vectorized, MPI-OpenMP, Xeon PhiTM-enabled code relative to our MPI baseline for the strong scaling runs shown in Figures 9 and 10. The “Cores” column specifies how many CPU-side cores were used in the computation. Baseline computations used one MPI process per core, while accelerated runs used one MPI process per socket, each spawning 8 OpenMP threads to make full use of the available cores. This table illustrates that when compute nodes are given nontrivial workloads, as is the case in the strong scaling runs at low core count, the Xeon PhiTM contributes to a significant speedup.

Figure 8: Distribution of Xeon PhiTM runtimes for a 64-node run. The solid blue line indicates the mean runtime, while the dotted lines indicate deviation from the mean. The curve in magenta ignores the core numbers and plots the sorted times, to make it easier to visualize the distribution.

Figure 9: Strong scaling for 93k 7th order elements. Phi-enabled runs assigned 63% of the elements per node to the Xeon PhiTM (s). Baseline runs were original MPI-only code. Number of processors refers to total number of CPU cores used in baseline case, and total number of OpenMP threads used on the host CPU side in Xeon PhiTM-accelerated cases.

Figure 10: Strong scaling for 17.4k 15th order elements. Phi-enabled runs assigned 66% of the elements per node to the Xeon PhiTM (s). Baseline runs were original MPI-only code. Number of processors refers to total number of CPU cores used. Phi-accelerated runs used 1 Xeon PhiTM for every 8 CPU cores (1 socket).

7. CONCLUSION

In this work we have presented a new partitioning algorithm targeting hybrid codes on heterogeneous clusters. We demonstrated the efficacy of our algorithm using a large-scale seismic wave propagation application on the Stampede Supercomputer. We observed speedups of up to 5.78x for runs using 7th order elements and up to 6.88x for 15th order elements. Our partitioning scheme was able to construct relatively consistent subdomains for the Xeon PhiTM. We were able to increase the performance of the baseline MPI code through the use of vector instructions, and were also able to maintain performance when moving from pure MPI

Figure 11: Number of elements assigned to the CPU and Xeon Phi™ in the single-socket strong scaling runs. Number of processors refers to total number of CPU cores used in baseline case, and total number of OpenMP threads used on the host CPU side in Xeon Phi™-accelerated cases.

Figure 12: Weak scaling for 7th order elements. Phi-enabled runs completed with 63% of the elements given to the Xeon Phi™ (s). Baseline runs were original MPI-only code. Runs labeled “8 MPI” or “8 OMP” indicate that only one socket per node was used.

to an MPI-OpenMP hybrid. On a system like Stampede where the majority of computational power is delivered by accelerators, we have seen that optimizing code for effective execution on the accelerator can have a dramatic impact on performance. The partitioning scheme also enables the developers to program using a SPMD paradigm, which does not appear straightforward while considering heterogeneous architectures. We believe our work is the first to take such an approach and will make it easier to obtain good performance and scalability on large heterogeneous clusters.

Acknowledgements

We would like to thank Jesse Kelly for contributions to the preliminary version of this project. Support for this work

Figure 13: Weak scaling for 15th order elements. Phi-enabled runs completed with 66% of the elements given to the Xeon Phi™ (s). Baseline runs were original MPI-only code. Runs labeled “8 MPI” or “8 OMP” indicate that only one socket per node was used.

was provided through the U.S. National Science Foundation (NSF) grant CMMI-1028889 and through the Scientific Discovery through Advanced Computing (SciDAC) project DE-SC0009286 funded by the U.S. Department of Energy Office of Science, Advanced Scientific Computing Research and Biological and Environmental Research.

8. REFERENCES

- [1] Intel Cilk plus, October 2014.
- [2] Intel Math Kernel Library, October 2014.
- [3] Stampede user guide, October 2014.
- [4] J. D. D. Basabe and M. K. Sen. Stability of the high-order finite elements for acoustic or elastic wave propagation with high-order time stepping. *Geophysical Journal International*, 181:577–590, 2010.
- [5] T. Bui-Thanh, C. Burstedde, O. Ghattas, J. Martin, G. Stadler, and L. C. Wilcox. Extreme-scale UQ for Bayesian inverse problems governed by PDEs. In *SC12: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2012.
- [6] C. Burstedde, L. C. Wilcox, and O. Ghattas. **p4est**: Scalable algorithms for parallel adaptive mesh refinement on forests of octrees. *SIAM Journal on Scientific Computing*, 33(3):1103–1133, 2011.
- [7] K. Goto and R. A. V. D. Geijn. Anatomy of high-performance matrix multiplication. *ACM Transactions on Mathematical Software*, pages 1–25, 2008.
- [8] J. S. Hesthaven and T. Warburton. *Nodal Discontinuous Galerkin Methods*, volume XIV of *Texts in Applied Mathematics*. Springer, 2008.
- [9] J. Jeffers and J. Reinders. *Intel Xeon Phi Coprocessor High Performance Programming*. Morgan Kaufmann, March 2013.
- [10] A. A. Kaufman and A. L. Levshin. *Acoustic and Elastic Wave Fields in Geophysics, III*, volume 39 of

Methods in Geochemistry and Geophysics. Elsevier B.V., 2005.

- [11] V. Kindratenko and P. Trancoso. Trends in high-performance computing. *Computing in Science Engineering*, 13(3):92–95, 2011.
- [12] G. M. Morton. *A computer oriented geodetic data base and a new technique in file sequencing*. International Business Machines Co., 1966.
- [13] B. Riviere and M. F. Wheeler. Discontinuous finite element methods for acoustic and elastic wave problems. In *Current Trends in Scientific Computing: ICM 2002 Beijing Satellite Conference on Scientific Computing, August 15-18, 2002, Xi'an Jiaotong University, Xi'an, China*, volume 329, page 271. American Mathematical Society, 2003.
- [14] C. Rosales. Porting to the intel xeon phi: Opportunities and challenges. Technical report, Texas Advanced Computing Center, The University of Texas at Austin, Extreme Scaling Workshop 2013, September 2013.
- [15] E. Saule, K. Kaya, and U. V. Catalyurek. Performance Evaluation of Sparse Matrix Multiplication Kernels on Intel Xeon Phi, Feb. 2013.
- [16] K. Schulz, R. Ulerich, N. Malaya, P. Bauman, R. Stogner, and C. Simmons. Early experiences porting scientific applications to the many integrated core (mic) platform. Technical report, Texas Advanced Computing Center, The University of Texas at Austin, April 2012.
- [17] H. Sundar, R. Sampath, and G. Biros. Bottom-up construction and 2:1 balance refinement of linear octrees in parallel. *SIAM Journal on Scientific Computing*, 30(5):2675–2708, 2008.
- [18] Top 500 Supercomputing Sites. <http://www.top500.org/>.
- [19] R. C. Whaley, A. Petitet, and J. J. Dongarra. Automated empirical optimization of software and the ATLAS project. *Parallel Computing*, 27(1–2):3–35, 2001. Also available as University of Tennessee LAPACK Working Note #147, UT-CS-00-448, 2000 (www.netlib.org/lapack/lawns/lawn147.ps).
- [20] L. Wilcox, G. Stadler, C. Burstedde, and O. Ghattas. A high-order discontinuous galerkin method for wave propagation through coupled elastic–acoustic media. *Journal of Computational Physics*, 229(24):9373–9396, 2010.
- [21] S. Williams, D. D. Kalamkar, A. Singh, A. M. Deshpande, B. Van Straalen, M. Smelyanskiy, A. Almgren, P. Dubey, J. Shalf, and L. Oliker. Optimization of geometric multigrid for emerging multi- and manycore processors. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, SC '12*, pages 96:1–96:11, Los Alamitos, CA, USA, 2012. IEEE Computer Society Press.