This paper describes objective technical results and analysis. Any subjective views or opinions that might be expressed in the paper do not necessarily represent the views of the U.S. Department of Energy or the United States Government.

SAND2016-9011C

Topology-Aware Performance Optimization and Modeling of Adaptive Mesh Refinement Codes for Exascale

Cy P Chan, John D Bachan, Joseph P Kenny, Jeremiah J Wilke, Vincent E Beckner, Ann S Almgren, John B Bell

Lawrence Berkeley National Laboratory, Berkeley, CA {cychan, jdbachan, vebeckner, asalmgren, jbbell}@lbl.gov
Sandia National Laboratories, Livermore, CA {jpkenny, jjwilke}@sandia.gov

***Abstract*—We introduce a topology-aware performance optimization and modeling workflow for AMR simulation that includes two new modeling tools, ProgrAMR and Mota, which interface with the BoxLib AMR framework and the SST-macro network simulator. ProgrAMR allows us to generate and model the execution of task dependency graphs from high-level specifications of AMR-based applications, which we demonstrate by analyzing two example AMR-based multigrid solvers with varying degrees of asynchrony. Mota generates network topology-sensitive box mappings, which we apply to optimize the data layout for the example multigrid solvers. While the sensitivity of these solvers to layout and execution strategy appears to be modest for balanced scenarios, the impact of better mapping algorithms can be significant when performance is highly constrained by network hop latency. Furthermore, we show that network latency in the multigrid bottom solve is the main contributing factor preventing good scaling on exascale-class machines.**

I. Introduction

The solution of partial differential equations (PDE's) using adaptive mesh refinement (AMR) has proven to be a computationally efficient approach for simulating a broad range of physical phenomena in which feature sizes vary widely in scale, such as those occurring in astrophysics, combustion, and geophysical modeling [1], [2]. Significant effort has gone into tuning AMR-based applications for current high performance computers.

However, the push towards exascale computing is forcing changes in the way these high performance systems are designed. In order to keep aggregate system power in check while continuing to push the performance envelope, future systems will need to be composed of simpler, more power efficient compute elements [3]. The composition of leadership class machines is expected to transition quickly over the next few generations with nodes becoming much more powerful, data locality becoming increasingly important, and the balance between compute and communication performance shifting.

This evolution in hardware design will compel scientists to reexamine several core aspects of AMR in order to achieve high performance on future architectures, including a) greater attention to *data locality* and b) *hiding communication latency* through increased asynchrony. Fundamental to block-structured AMR is the spatial decomposition of the domain into multiple grids at multiple levels of resolution. The specific application determines how the domain is decomposed, but performance considerations dictate the placement of *grids* (or *boxes*) onto processors. The placement of data affects both the computational load balance and the cost of communication over the network. As parallelism and aggregate compute performance increase within compute nodes, the relative cost of data movement and messaging will increase, with the possible outcome that optimizing the locality of the grids at the expense of load balance may improve performance at larger scales. Additionally, asynchronous task-based programming models may provide more flexible execution strategies on exascale architectures to help overlap communication latencies with independent computation. [4], [5], [6], [7], [8]. Since porting existing large, complex scientific applications to task-based programming models may be challenging, it is important to understand the potential benefits on future hardware before investing significant engineering effort.

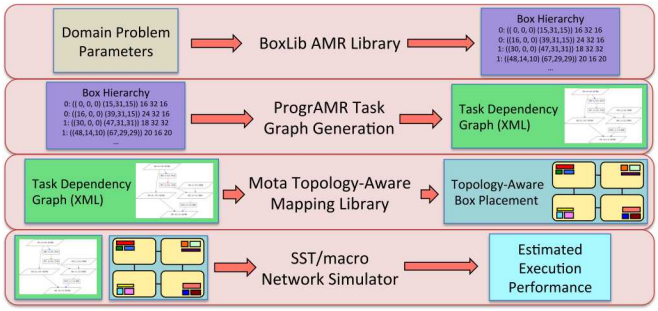


Figure 1: Our AMR performance modeling workflow

Figure 1 illustrates our AMR modeling workflow. First, we use applications from the BoxLib [1], [9] AMR framework to generate multilevel grid hierarchies for analysis. The first modeling tool, ProgrAMR, produces skeletonized task-graph representations from high-level expressions of algorithms. The second, Mota Mapper, is a multi-objective topology-aware data mapping library that we use to map block-structured AMR boxes to network nodes in a way that simultaneously reduces communication costs while balancing compute loads and memory constraints. Finally, the algorithm task graph and mappers are fed into the SST-macro network simulator [10] to illustrate various trade-offs in system performance.

We utilize this methodology to examine the implications of changes in system architecture and execution models on the performance and scalability of multigrid linear solvers, whose execution typically contributes a significant fraction to the total simulation time of AMR applications that utilize them. Some of the authors have previously collaborated on a tool-chain that allows AMR applications to be represented as a task dependency graph that can be simulated at system scale [11].

Sandia National Laboratories is a multimission laboratory managed and operated by National Technology & Engineering Solutions of Sandia, LLC, a wholly owned subsidiary of Honeywell International Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525.

This paper follows a similar approach, but introduces new tools that allow more flexible algorithm, data layout, and execution model exploration than the previous work. Specifically, ProgrAMR allows higher-level algorithm specification, enabling much more complex algorithmic variants (such as multigrid) to be evaluated. It also tags tasks with metadata to allow varying the degree of asynchrony during execution. Furthermore, Mota Mapper enables us to evaluate new topology-aware box mapping strategies.

The rest of the paper is structured as follows. Section II gives an overview of the BoxLib AMR framework. Section III introduces the ProgrAMR semantics that provides a concise way to generate AMR task dependency graphs, and describes the set of AMR-based multigrid algorithms we modeled. Section IV introduces the Mota Mapper library of multi-objective, topology-aware mapping algorithms for assigning AMR boxes to ranks on a network. Section V gives an overview of the SST-macro network interconnect simulation framework. Section VI discusses four different models of synchronization that we explore using the task-graph representation of the solvers. Finally, Section VII presents experimental results gathered using our tools in conjunction with the SST-macro network simulator to show:

1) a baseline analysis of scaling behavior of the AMR multigrid algorithms,
2) the impact of new mapping algorithms on performance with varying network configurations,
3) the sensitivity of multigrid variants to mapping optimizations,
4) a comparison between execution models with different degrees of asynchrony, and
5) the impact of box consolidation to fewer ranks during multigrid coarsening.

II. BoxLib: A Block-Structured AMR Simulation Framework

BoxLib is a mature software framework for building massively parallel block-structured adaptive mesh refinement (AMR) applications [1], [9]. It is one of a number of current publicly available AMR frameworks; see e.g., [12], for an overview. Many application codes spanning a large range of scientific domains are built on these frameworks – BoxLib-based codes, for example, include those for astrophysical, combustion, atmospheric, and subsurface flow simulations.

BoxLib supports explicit and implicit grid-based operations as well as particle and particle-mesh operations on adaptive hierarchical meshes. Multiple time-subcyling modes are supported for adaptive mesh simulations, and multilevel multigrid solvers are included for cell-based and node-based data. BoxLib application codes have demonstrated good scaling behavior on up to 100,000 cores on current multi-core architectures [13], [14]. BoxLib itself and several BoxLib-based codes are publicly available at `https://github.com/BoxLib-Codes`.

III. ProgrAMR: Describing AMR Pure-Functionally

A. Description

At its core, ProgrAMR is a semantics for describing a broad set of AMR-related algorithms at a high level. By design, it avoids execution details, giving rise to the two salient features: 1) conciseness of algorithm expression, and 2) decoupling of algorithm specification from execution to aid the exploration of execution strategies as a separate concern. It leverages pure-functional semantics so that parallelism is exposed via data-driven behavior. ProgrAMR currently meets the needs of task graph driven system simulation; however, we may extend it in the future to provide a full task-based AMR runtime. The AMR-specific aspects of the API currently allow description of *skeletonized* algorithms, meaning that they generate the computation and communication event traces necessary to drive the network simulator, without doing any actual floating-point computation. The time spent in on-node compute kernels is represented by performance model profiles generated by the ExaSAT performance modeling framework [15].

We have implemented ProgrAMR as a C++ library in the spirit of an embedded DSL. The operations exposed by the library are mostly constructors for the various AST nodes pertaining to our custom AMR semantics. For the user, C++ is used as a scripting language for generating the AST. The embedded AST semantics is pure-functional, meaning that the whole program is one large expression tree of function applications, absent of any mutable state. The application variables are *big* in the sense that they are implicitly distributed over the entire machine's partitioned memory, and application functions then operate on these whole distributed variables. This leads to a very SIMD-like programming experience where the scope of the instruction's parallelism is machine-wide, and the tedious details about when to communicate data and schedule computation are abstracted away.

Among the catalog of built-in data structures and operations, the first to note is the `Ex<T>` type: an AST expression node that *at runtime* will produce a value of type `T`. For AMR simulations, state variables are `Ex<Slab<T>>`s, which are expressions representing collections of boxes all at the same level of refinement (called the slab's domain), where each cell of a box holds a single value of type `T`. Since ProgrAMR is pure-functional, slabs are immutable and may not be altered; instead, new slabs are generated by applying one of the built-in operations to existing slabs. For instance, the slab operation

```
Ex<Slab<T>> slab_halo(Ex<Slab<T>> x, int width)
```

takes an expression for computing a slab `x`, and returns a new expression for a new slab that has the same interior values as `x`, but with halo (or ghost) zones added to the sides (thus implying communication). This is commonly followed by

```
Ex<Slab<T>> slab_stencil(Ex<Slab<T>> x, ...)
```

which takes an expression for a slab containing halo regions, applies a stencil to the cell values, and produces an expression for a new slab with updated interior cells but devoid of the consumed halo regions. We have similar operations for prolongation and restriction between parent/child slabs, and a halo fill that accepts two parent slabs to be interpolated between. This makes for a very intuitive coding style where slabs expressions are simply passed in and out of slab operations. Also, common patterns of slab operations can be factored into vanilla C++ user functions.

The ProgrAMR slab operations used to build an application are typically data-independent, i.e. the amount of computational work they represent is independent of their inputs' numerical values. However, in order to include the costs of convergence detection for iterative methods and handle adaptive time-stepping, we added support for `allreduce` collectives. We believe we handle the program dynamics of data-dependent control flow in a pure functional setting with a degree of novelty from the perspective of the HPC community. The technique we employ is a mainstay for any pure-functional programmer: the monad [16]. To understand the problem the monad solves, consider that data-independent expression trees are all static and finite. However, most iterative algorithms dynamically decide what sequence of operations to perform based on their intermediate results, hence they cannot know their fully expanded expression tree up front. In order to represent a whole dynamic, data-dependent program as an expression tree (and retain pure-functional semantics), we need a representation that allows expressions to dynamically unfurl. By adding just one additional AST node type, the *monadic bind* operation, we are able to wield an expression unfurling mechanism that is Turing-complete, making it powerful enough to encode any algorithm based on the primitives already discussed.

Expressing reduction-dependent program control flow requires two operations:

```
Ex<T> slab_reduce(Ex<Slab<T>> x, T modeled_result)
```

folds a slab worth of values down to a single scalar. Since ProgrAMR uses skeletonization to avoid doing numerical computation, the user simply provides the result of the modeled reduction.

```
Ex<T> mbind(Ex<U> x, std::function<Ex<T>(U)> f)
```

produces a data-dependent expression. The two arguments are the expression `x` on which we have a dynamic dependence, and the C++ function `f` that will produce the dependent expression given `x`'s runtime value. The returned expression is a proxy for the dependent expression so that it can nest into further (perhaps data-dependent) expressions.

Once the user program is specified, ProgrAMR can then generate the machine-wide task graph to drive the network simulator. We first run the user program to generate the AST, then visit the slab operations embedded in the tree's nodes in topologically sorted order. For each box in every operation's output, we generate a compute task that represents the computation of the box's data and generate communication messages for the task's true data dependencies on individual boxes from the operation's inputs. The set of tasks produced is the Cartesian-product of the set of operation nodes and the boxes in the mesh. The tasks and messages are then saved to an XML file to be read by the SST-macro simulation framework. This process is depicted in Figure 2. The following section describes the AMR-based multigrid algorithms we implemented and modeled with ProgrAMR.

B. AMR-Based Multigrid Algorithm Skeletons Using ProgrAMR

Many algorithms for multiphysics PDE-based applications require the iterative solution of large linear systems arising from either discretization of elliptic equations or implicit treatment of parabolic equations. Multigrid algorithms are often the method of choice for solving these systems. AMR simulations with subcycling in time typically require linear solves across the grids at a single AMR level only; algorithms without subcycling typically require solves across all the levels in the hierarchy. For the rest of this paper we focus on the performance of multigrid on multiple levels of an AMR hierarchy, using either V-cycles or F-cycles (see, e.g., [17], for more detail). For each of the two cases we implemented a skeletonized version of the algorithm in ProgrAMR. While the variants are not directly comparable since they have different convergence characteristics, it is still informative to compare how the performance of each variant responds to other changes in the system, such as modifications to the box layout or network interconnect, relative to the other variant.

Figure 3 illustrates the differences between the variations of an integration strategy that requires linear solves. In the subcycled version, temporal resolution is coupled to spatial resolution, resulting in multiple fine grid time steps per coarse grid time step. Each time step at each level requires a single-level multigrid solve, potentially followed by a multi-level multigrid synchronization step. An advantage of this time integration strategy is that fewer time steps are taken on the coarser levels than if all the levels were advanced with the same small time step.

In contrast, the time integration algorithm without subcycling in time advances the data on all levels with the same time step, typically dictated by the data at the finest level. In this case the multigrid solver includes all AMR levels within each solve, requiring much more inter-level communication. Although this approach requires all levels to update at the finest time resolution, it can often be more efficient because the coarsest AMR level covering the entire domain can often be coarsened much more deeply than the union of a grids at a finer level. Deeper multigrid cycles can result in faster convergence and enable additional optimizations such as box consolidation or agglomeration.

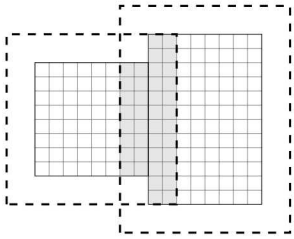
V-cycles and F-cycles differ in the order in which the multigrid levels are traversed and in how many times each level is visited within the cycle. Relative to V-cycles, F-cycles spend more of their time at the coarser multigrid levels, thus we expect their performance to be more sensitive to the performance of the bottom solver and other coarse grid optimizations. Results comparing the relative performance of these algorithms will be explored in Section VII-B.

IV. Mota Mapper: The Multi-Objective Topology-Aware Mapping Library

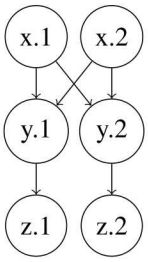
The AMR box layout problem is to determine an assignment (mapping) of boxes to ranks to simultaneously minimize the computational load imbalance and communication costs of the application. There is an obvious trade-off between the

```
typedef Ex<Slab<double>> ExSlabD;
// builds a serial expression tree:
//   x -> y -> z
ExSlabD smooth(ExSlabD x) {
  ExSlabD y = slab_halo(x, 2);
  ExSlabD z = slab_stencil(y, 2, ...);
  return z;
}
```

(a) User code snippet.



(b) Slab x's domain with two boxes (deduced for y and z). Halo of 2 cells outlined, and regions communicated shaded.



(c) Generated task graph.

Figure 2: An example of how ProgrAMR generates the task graph given an expression tree and initial mesh configuration.

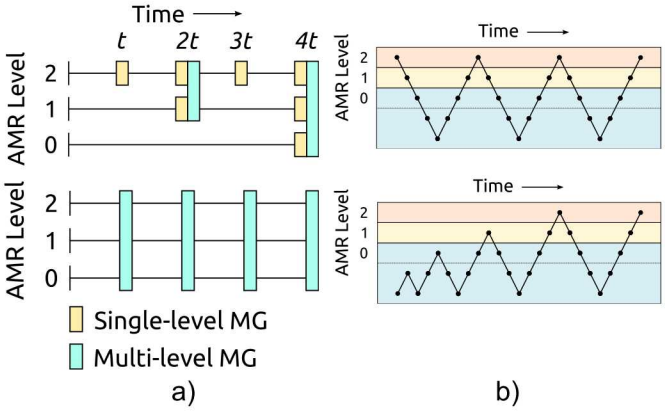


Figure 3: a) A comparison between AMR simulation with and without subcycling; b) A comparison between multigrid V-cycles and F-cycles

two objectives: mappings that favor good load balance tend to increase communication costs, and those that minimize communication costs tend to have poor load balance. Finding a communications-optimized load balanced mapping is an NP-complete multi-objective optimization problem [18], thus we are forced to resort to algorithms that provide an *approximate* solution.

These algorithms fall into two categories: those that do not take into account the network topology and those that do. As computer hardware evolves, data movement and communication will increasingly impact the runtime of our applications [3], and thus we expect reducing the cost of moving data across the network will become more important. This section introduces three new network-topology-aware approaches that produce approximate solutions to this multi-objective problem. In addition, we summarize and examine three current algorithms utilized in BoxLib for comparison with the new algorithms. Experimental evaluations of these mapping algorithms will be given in Section VII.

A. Non-topology-aware algorithms

Two of the current algorithms (knapsack and space-filling curve) are non-topology-aware, taking into account the size and/or geometry of the boxes but not network topology.

1) Knapsack (KS): This box mapping algorithm focuses on balancing the number of grid cells across ranks, resulting in a good balance of the memory footprint across ranks as well as computational load. The number of cells in a box is used as the weight of the box, and the mapping algorithm tries to evenly partition the total weight of the boxes in each level across available ranks in the network. It does not attempt to use any information about how the boxes communicate, only their computational and memory footprints.

2) Space-Filling Curve (SFCS): The space-filling curve mapping algorithm, like many of the algorithms considered in this paper, constructs a linear sequence of the boxes in a way that attempts to keep boxes that communicate close together in sequence order. It does so by identifying a point with each box (e.g. a corner or the center), and using a Z-Morton coordinate transformation [19] to order them. Each AMR level is mapped in turn, and a separate sequence of boxes is generated for each level independently. For each level, the boxes are distributed (in Z-Morton order) to a sequence of N buckets (where N is the number of ranks) in a way that maintains load balance across buckets. Using a new set of buckets for each level ensures that the per-level load balance is addressed while grouping boxes that probably communicate (based on their proximity) together. The buckets and ranks are then sorted and mapped bijectively such that buckets with larger load are mapped to ranks with smaller load (taking into account the load of the boxes from previously mapped AMR levels), and this process is repeated for each AMR level.

B. Topology-aware algorithms

We examine one current topology-aware mapping algorithm (PFCM) from BoxLib and introduce three new ones (GR, RCM, and RB).

1) Proximity-Filling Curve (PFCM): This BoxLib algorithm utilizes a similar technique to SFCS, but considers boxes across all levels simultaneously to construct the Z-Morton ordered sequence. Sequencing the boxes in all levels simultaneously places communicating boxes from different AMR levels closer to each other, potentially increasing the performance of restriction and prolongation operations. Additionally, the ranks are ordered in such a way to increase the topological locality between nearby ranks (the specific ordering mechanism depends on the type of network). Both the ordering of the boxes and the ordering of the ranks is used during box distribution, resulting in lower inter-node

messaging costs for neighboring boxes mapped to different ranks.

One trade-off of sequencing all levels at once is that the load balance of each AMR level on its own is not taken into account during box distribution, only the memory footprint of the boxes per rank. Thus per-level load balance may not be as good as the other algorithms, and performance may suffer for certain AMR solvers since there may be several computational stages that operate on levels individually, during which no work is available for boxes on other levels.

2) New Topology-Aware Algorithms: The new box mapping algorithms are based upon three topology mapping algorithms described in [18] and implemented in the LibTopoMap library [20]. In order to apply some of the ideas from that paper to the AMR box mapping problem, we made some significant enhancements to the algorithms. The topology mapping problem in that paper was formulated as mapping a set of equivalent processes to a set of compute nodes, each with a capacity for a number of processes (e.g. 1 process per core). The computational load and memory footprint of each process was not taken into account, so each process was effectively treated as if it took an equal amount of resources on the node. Furthermore, for some mappings there are constraints on the number of processes per node and restrictions on the cardinality of the application and network graphs.

We modified these algorithms to make them applicable to the AMR box mapping problem, which maps boxes to ranks, and must take into account the different computational costs and memory footprint of the boxes assigned. Our extended methods can handle multiple computational load balancing constraints and a memory constraint, and supports mapping an arbitrary number of boxes to an arbitrary number of ranks. We believe the methods we developed are generic enough to be applied to other application domains that require multi-objective topology mapping.

3) Network Topology Models: In order to map boxes in a topologically-aware fashion for proposed exascale interconnect configurations, we utilize a parameterized network interconnect model that allows us to represent the connectivity graph and routing behavior for various network topologies. We implemented four classes of interconnect topologies: a generic N-dimensional torus, a generic dragonfly, and specific models for the Edison and Cori supercomputers at NERSC. Our models are based on [21], [22], the SST-macro simulation framework [10], and information communicated from NERSC staff. The parameterized generic models allow us to generate mappings for future interconnects, while the specific models allow us to map boxes for runs on current machines.

These models allow us to customize both the connectivity and the routing algorithms so that we can examine not just the topology, but also edge/link utilization metrics that are needed in some mapping algorithms. For example, in the greedy algorithm boxes are placed to avoid utilizing network links that are already servicing heavy traffic between previously placed boxes, requiring a model of the interconnect routing behavior.

4) Greedy (GR): The greedy algorithm maps boxes to ranks one at a time, sequentially choosing the next box to map based on how heavily connected it is to the already mapped boxes. Ranks are filled up until they have no more capacity, then the next rank is chosen based on a single-source shortest path algorithm through the interconnect until one is found with sufficient capacity. For a more detailed description of non-AMR version of the greedy algorithm, see [18].

In the previous work, each rank has a capacity of processes, assumed to have equivalent cost. To suit the AMR box mapping problem, each rank k is instead given a tuple $c_k = (c_{k,1}, ..., c_{k,n})$ of grid cell capacities where the number of components n equals the number of AMR levels plus one, representing all level-specific compute loads plus the overall memory balance constraint.

Each box j with g_j grid cells in AMR level l is assigned a tuple to represent its weight $w_j = (w_{j,1}, ..., w_{j,n})$, where:

$$w_{j,i} = \begin{cases} g_j & \text{if } i \in \{l, n\}, \text{and} \\ 0 & \text{otherwise.} \end{cases} \tag{1}$$

The capacity constraint can then be defined as:

$$\sum_{j \in M(k)} w_{j,i} \leq c_{k,i}, \forall k, i, \tag{2}$$

where $M(k)$ is the set of boxes mapped to rank k.

One problem we had to solve was what capacities to use for the ranks. The capacity model of [18] does not face this problem since they simply assume the total process capacity (number of cores) of the system is fixed by the hardware and equals or exceeds the number of processes to be mapped. They also do not attempt to minimize the average or maximum process load across nodes, since it is assumed each additional process assigned to a node will receive an independent core.

In the AMR box mapping problem, there is no fixed compute load capacity, and furthermore, mappings that satisfy smaller maximum load capacities may perform faster than others. Our rank load capacities should thus be chosen to be as tight as possible while still allowing a solution that satisfies the capacity constraint (2). Such a solution will result in a mapping with an even distribution of compute load and memory footprint across N ranks.

Our approach to support the multi-objective capacity constraint is to use:

$$c_{k,i} = \max\left(\max_j w_{j,i},\ \alpha \sum_j w_{j,i}/N\right) \forall k, i, \tag{3}$$

which equals the larger of the maximum box weight and a scaled average box weight for each constraint. We set $\alpha = 1$ as an initial guess; however, with such a strict constraint, the algorithm often encounters a situation with boxes still remaining and no rank with sufficient capacity because a constraint is too tight for some i.

When this condition is detected, we retry the mapping from scratch with an increased capacity constraint until the mapping algorithm completes. Starting from scratch allows the boxes to be more evenly spread across ranks compared to simply increasing the capacity of the ranks at the point where we run out of space. Using a larger capacity multiplier between retries will speed up the time to find a solution, but reduces the average quality of load balance.

5) Reverse Cuthill-McKee (RCM): As with the SFC algorithm, the basic idea of the RCM mapping algorithm [23], [24] is to construct linear orderings of the boxes and ranks so as to reduce the distance between communicating boxes and connected ranks, respectively. As the name implies, this algorithm uses the Reverse-Cuthill-McKee sparse matrix bandwidth reduction algorithm on the adjacency graphs of the application and network. In the RCM process mapping algorithm presented in [18], the number of processes per node is either 0 or 1, and the number of available process slots must equal the number of processes to be mapped.

Our method allows an arbitrary number of boxes to be mapped to each rank by leveraging a new distribute algorithm that takes a linear sequence of boxes and distributes them to a linear sequence of ranks in a way that attempts to keep nearby boxes close together while simultaneously satisfying all of the multi-objective capacity constraints described in IV-B4. This new distribute algorithm is described in the following section.

6) New distribute algorithm: We designed a new box distribution algorithm to map an arbitrary number of boxes to an arbitrary number of ranks given a locality preserving ordering of both the algorithm and network graphs. The distribution algorithm also takes an arbitrary number of load and memory balance constraints to determine placement of the boxes.

This algorithm is distinguished from the current BoxLib distribution algorithm used in the SFCS and PFCM algorithms in that it handles multiple constraints, providing memory balance and per-level load balance on all AMR levels, as well as network topology-aware box placement that is sensitive to both intra- and inter-level communications. The original BoxLib distribution algorithm handled a single constraint, so there were two options: either 1) each AMR level was mapped individually, providing good load balance on each individual level but ignoring inter-level communication costs, or 2) all boxes were mapped together, discarding per-level compute load information and balancing memory only.

Algorithm 1 Distribute($boxes[M], ranks[N], \gamma$)

1: Compute box weights and initialize rank capacities using Eqs. (1) and (3) with $\alpha = 1$
2: $k \leftarrow 1,\ d \leftarrow 1$
3: **while** solution hasn't been found **do**
4: **for** j in $(1, ..., M)$ **do**
5: **while** rank k does not have capacity for box j according to (2) **do**
6: **if** all ranks have been checked for box j **then**
7: Break out of for loop (line 4)
8: **end if**
9: **if** $k = N$ or $k = 1$ **then**
10: $d \leftarrow -d$
11: **end if**
12: $k \leftarrow k + d$
13: **end while**
14: Assign box j to rank k (insert j into $M(k)$)
15: **end for**
16: **if** solution was not found **then**
17: $\alpha \leftarrow \gamma \cdot \alpha$
18: Reset box assignments and rank capacities
19: **end if**
20: **end while**

The new distribute algorithm is sketched in Algorithm 1. Like the outer retry loop of the greedy algorithm, it initializes the rank capacity tuple (2) to be very tight and exponentially loosens the capacity constraints by increasing α by some $\gamma > 1$ until a solution is found. The algorithm attempts to place boxes on the ranks in the order they occur in the sequences. Like the current BoxLib distribution, it will place multiple boxes on the same rank until the rank's capacity is reached; however, due to the presence of multiple constraints, ranks may not fill up all of their capacities on the first pass. Thus it becomes necessary to backtrack across the sequence of ranks to continue finding space for boxes in the hope that the remaining boxes will fill in the "holes" left on the previous passes. One might consider other methods for filling holes while traversing the ranks, such as using a dovetail search mechanism, but we have not yet explored these alternatives.

7) Recursive Bisection (RB): The recursive bisection approach given in [18] recursively divides both the application and network graphs, mapping the partitions to each other during each bisection call. This method is not suitable for AMR box mapping because each bisection of the graph balances a single constraint and requires that network capacity equals the application graph size.

In order to support AMR box mapping, we instead use the METIS recursive bisection algorithm to decompose the graph into a hierarchical tree structure, where at every node of the tree, the bisection is chose to minimize the edge cut cost. We then leverage the fact that the leaves of the tree (i.e. the returned partitions) are numbered in a depth-first traversal (ensuring that entire subtrees are enumerated before moving to the next subtree), thus providing a naturally locality-preserving linear ordering. We apply this algorithm to produce orderings of both the application and network graphs, then map boxes to ranks using the multi-constraint distribute algorithm described in Section IV-B6.

V. Coarse-grained Full System Simulation

The SST-macro element library of the Structural Simulation Toolkit (SST) has been developed to allow efficient discrete event simulation of HPC systems at full scale [10], [25], [26]. In order to keep full system simulations tractable, sensible approximations must be made. Simulating network traffic as flows holds an intuitive attractiveness in that messages can flow through the system with a very small number of events unless congestion is present. However, in the presence of congestion, flow updates must be propogated across switches in the system, and simulation costs rise unacceptably for large simulations with significant congestion. An alternative for speeding up simulation is to packetize messages, eliminating the need for system-wide updates, but keep these packets "coarse-grained" so that there are fewer of them to simulate. However, these large packets block resources for larger amounts of time than would happen on real hardware, so packets competing for the same resource will encounter delays which introduce significant errors. The hybrid packet-flow model within SST-macro provides a practical tradeoff between accuracy and simulation cost. Once bandwidth for a

packet is assigned, later packets can utilize unused bandwidth but cannot acquire bandwidth previously assigned to earlier packets. In this way, more realistic congestion delays can be produced while keeping the model tractable. In this work the packet-flow model is used in coarse-grained full system simulation.

VI. AMR Simulation Framework

A. Modes of Synchronization

As the degree of parallelism and relative cost of communication increases with evolving system architecture design, bulk-synchronous parallel execution approaches may not perform well when scaled to exascale system sizes. We investigate here the impact of applying varying degrees of asynchrony to the AMR application skeletons described in Section III by simulating the execution under four different synchronization mechanisms. In all models except for fully asynchronous, there are conditions where ranks unnecessarily idle even if there are local tasks whose individual data dependencies have been fulfilled. The trade-off for these models is that more synchronous models tend to be easier to program and verify for correctness.

All of the synchronization methods we analyzed are simulated from the same task-graph XML description files. The only differences are additional synchronizations added during simulation itself to emulate the false dependencies that are required by more synchronous execution models. We enabled this flexibility by utilizing *epochs*, which we define in this context to be sets of tasks (sequences of computations or communications) that may proceed concurrently with no dependencies.

An example of an epoch would be a single explicit time step update that may execute after a boundary fill or halo exchange has completed. Since the computational step is typically executed across multiple AMR boxes, this epoch consists of multiple events spread across multiple ranks, each corresponding to an update of an individual box. This method is implemented in our task-graph XML by tagging each event (computation or communication) with an epoch number, grouping together events that are part of the same epoch.

The execution models and their differences are described below.

1) Fully Synchronous: In the fully synchronous execution model, each computational epoch in the algorithm proceeds in lock-step across all ranks. In our example, every rank waits for the preceding epoch's halo exchange messages to complete across **all** ranks before proceeding with the time step computation. No event of a given epoch may proceed until events on all ranks in a previous epoch have completed. This method is often implemented using global barriers (e.g. `MPI_Barrier`) to separate consecutive computational stages.

2) Rank Synchronous: In the rank synchronous execution model, ranks are decoupled so that no global synchronization is required. However, there is a local restriction on the order in which tasks are processed within each rank. Specifically, events cannot begin until all events on the same rank with a previous epoch number have completed. In the explicit time stepping example, each rank would wait until **all** boundary fill messages for **all** boxes owned by the rank have been received before proceeding to **any** time step computation. Thus, even if the boundary has been completely filled for an individual box, it cannot proceed with its update until the boundaries for the other boxes owned by the same rank have also been filled. This execution model mostly closely corresponds to the current BoxLib implementation and is often implemented using local synchronization of posted non-blocking receives (e.g. `MPI_Waitall`).

3) Phase Asynchronous: In the phase asynchronous execution model, ranks are allowed to more freely overlap communication and computation between epochs. Specifically, computation events may overlap arbitrarily with communication events, but computes in different epochs cannot execute simultaneously on the same rank. This behavior corresponds to *data-parallel* asynchrony where the concurrency exposed by the programming model includes the events of a single computational step spread across multiple data.

This execution model corresponds to entering a fork-join concurrency mode during each computational step, where a set of concurrent tasks and their data dependencies are registered with the runtime. All communications are assumed to occur asynchronously, and any computational task within an epoch can execute as soon as its individual data dependencies have been satisfied. In the explicit time-stepping example, a box may do its time step update as soon as its boundary has been filled without waiting for the boundaries of other boxes on the same rank to be filled. This execution model corresponds to using a light-weight runtime to manage registration of tasks and processing them immediately when their data dependencies have been satisfied (see [27] for a similar, although more powerful, example of this type of data-parallel asynchrony). Note that this model requires the runtime to be more attentive to individual incoming receives (e.g. `MPI_Waitany`) rather than blocking until all posted receives complete.

Compared to the fully asynchronous execution model, this model will not progress to the next epoch until all tasks of the current epoch have completed, but communications are issued asynchronously to maximize the opportunity for overlap. This type of parallelism is one model being considered for the AMR exascale applications area since it is easier for the application programmer to specify at a high level compared to a full task dependency graph.

4) Fully Asynchronous: In the fully asynchronous execution model, there are no false dependencies between tasks, providing the most flexibility to overlap communication and computation. Computation and communication tasks in different epochs may execute in arbitrary order on each rank, so long as their data dependencies have been satisfied. This behavior corresponds to true *task-parallel* asynchrony, where the programming model must expose a full task-graph to the runtime. The runtime must either have the entire application task graph registered at initialization or have some way of dynamically unfurling the task graph so that tasks are created and executed as their dependencies are satisfied. In the example, boxes across the AMR grid can be progressing forward in time completely independently of one another. Another

possibility enabled by task-parallel asynchrony is to have tasks from independent subroutines executing in any order, allowing the possibility of interleaving light-weight tasks that have relatively high dependent communication latencies (such as the bottom solve of a multigrid solver). This execution model corresponds to using a full task-graph based asynchronous runtime such as Charm, HPX, Legion, OCR, etc.

VII. Results

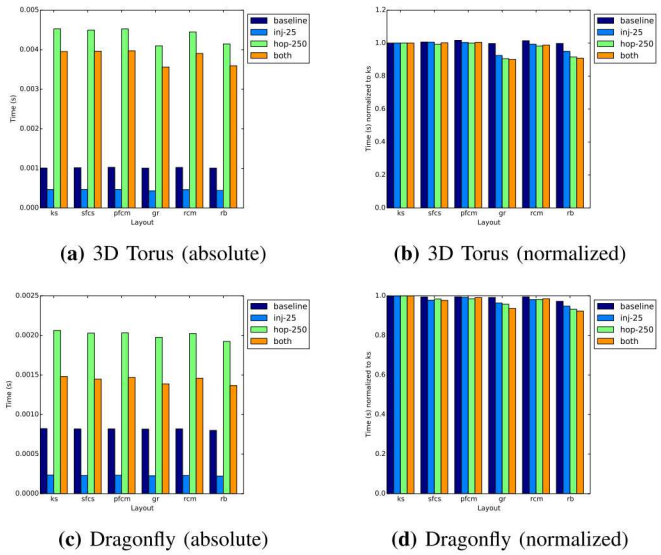
Since there are many parameter combinations we could explore, we chose a baseline configuration and deviated from it as needed to show specific parameter sensitivities. The baseline algorithm configuration is a multi-level AMR multigrid with no subcycling, using a V-cycle with 10 BiCGStab bottom solve iterations. The default execution model is fully-asynchronous. We analyze two classes of network topologies: the 3D torus and the dragonfly, both modeled by extending the configuration of current interconnects (modifying the topology and performance parameters) to represent a range of configurations from current to potential exascale designs. Broadly, the baseline exascale network consists of 64 port switches with injection bandwidth (per port) and latency of 100GB/s and 0.25us, respectively, and network bandwidth and latency of 120 GB/s and 25ns, respectively.

The on-node performance of the multigrid kernels (e.g. smooth, restrict, etc.) are modeled using the ExaSAT performance modeling framework [15], which predicts that the performance of the multigrid kernels will be determined by the hardware's on-node memory bandwidth and cache sizes. By tuning these parameters to represent an exascale class node architecture, we can estimate the compute time for each kernel during simulation.

In our simulations, we observed that the impact of box layout and asynchrony in execution is actually quite small for typical problem and hardware configurations. These optimizations appear to matter more for extreme configurations where the performance is bound by network hop latency. This situation happens when the injection latency is low relative to the switch hop latency or when compute resources are abundant, in both cases resulting in very communications-bound performance. Admittedly, these situations may not be typical, but we hope it is instructive to illustrate how sensitive the performance can be in such extreme cases. To this end, we chose two problem scenarios to illustrate: one balanced case with 10,449 boxes in the AMR hierarchy mapped to 3,072 ranks, and one extreme case, representing compute resource abundance, with 967 boxes freely mapped to any subset of 1,536 ranks.

A. Box Layout Optimization and Network Latency Sensitivity

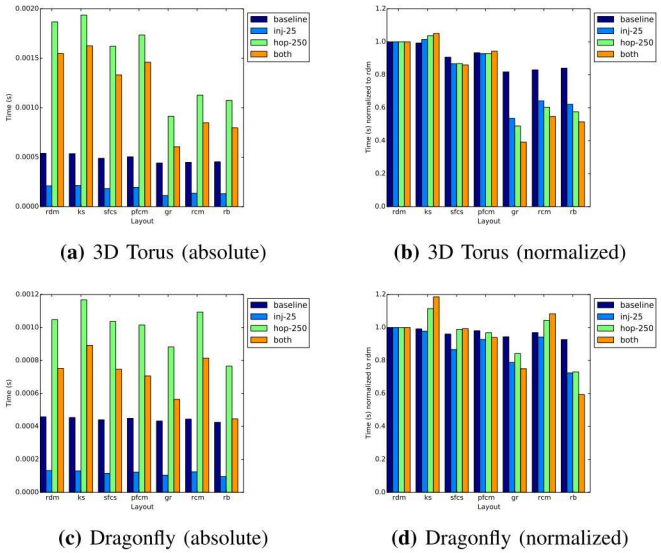
Using the Mota Mapper library, we examine the impact of optimizing box placement using the various algorithms described in Section IV. We also include random mapping of boxes to ranks as a standard reference. Random mapping (rdm) performs better than one might expect because boxes are distributed relatively evenly across ranks providing good load balance and avoiding congestion hotspots on the network,



(a) 3D Torus (absolute) **(b)** 3D Torus (normalized)

(c) Dragonfly (absolute) **(d)** Dragonfly (normalized)

Figure 4: A comparison of box layout algorithms in the larger, balanced case for different network latency configurations. Both absolute time and normalized time vs. random are shown.



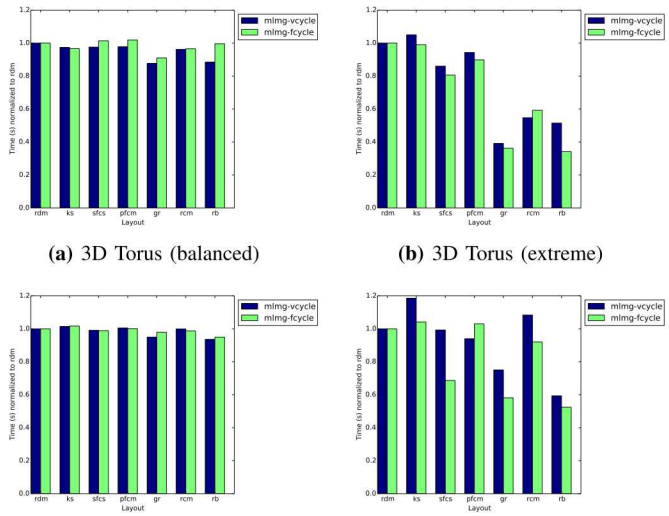
(a) 3D Torus (absolute) **(b)** 3D Torus (normalized)

(c) Dragonfly (absolute) **(d)** Dragonfly (normalized)

Figure 5: A comparison of box layout algorithms in the smaller, compute-abundant extreme case for different network latency configurations. Both absolute time and normalized time vs. random are shown.

though the average number of hops messages traverse through the network is high. Furthermore, we experiment with modifying the network latency parameters to see how reducing the injection latency from 250 ns to 25 ns, increasing the hop latency from 25 ns to 250 ns, or both modifications combined, affects performance.

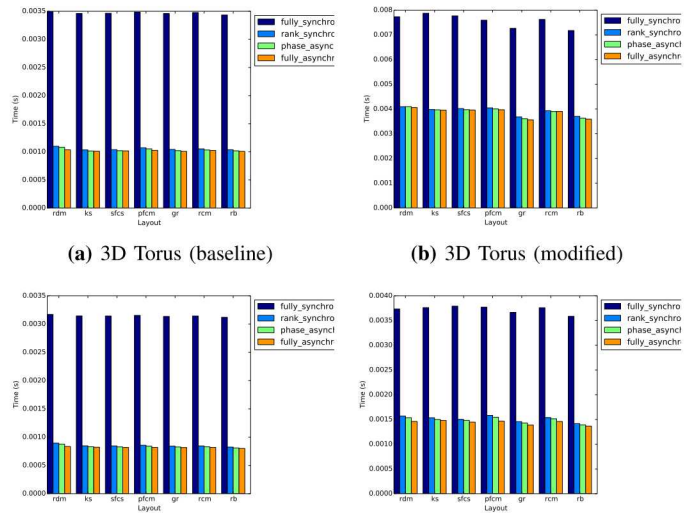
Figures 4 and 5 show the absolute and normalized performance for the large, balanced and small, extreme test configurations, respectively, on the 3D torus and Dragonfly networks. In the balanced case, the improvement from using topology-aware mapping algorithms is small, with the greedy or recursive bisection algorithms only improving the latency-modified performance by 6 to 10 percent relative to the best non-topology-aware mapping algorithm (usually the space-filling curve or knapsack algorithm). However, in the extreme



(a) 3D Torus (balanced) **(b)** 3D Torus (extreme)

(c) Dragonfly (balanced) **(d)** Dragonfly (extreme)

Figure 6: A comparison of relative performance of box layout algorithms for different solver algorithms. Results are shown for the latency-modified configuration with reduced injection and increased hop latency.



(a) 3D Torus (baseline) **(b)** 3D Torus (modified)

(c) Dragonfly (baseline) **(d)** Dragonfly (modified)

Figure 7: A comparison of asynchronous execution models in the larger, balanced case for different box layout algorithms. Both baseline and latency-modified configurations are shown.

case, the topology aware layouts improve the latency-modified performance by 37 to 54 percent. These large speedups are due to the algorithms placing boxes that communicate closer to one another, resulting in a packing of boxes into nearby ranks, rather than spreading them across the machine as with the knapsack algorithm.

B. Multigrid Algorithm Sensitivity

Figure 6 shows the relative performance of box layout algorithms for the V-cycle vs. F-cycle multigrid algorithms with modified network latency parameters. Since the solver algorithms are not directly comparable due to different convergence properties, the normalized execution time (relative to random placement) is shown to illustrate the relative sensitivity of each algorithm to the layout strategy. Again, the network topology-aware algorithms provide a greater improvement for the extreme case versus the balanced case.

In the balanced case, the V-cycle benefits marginally more than the F-cycle from box layout changes, while in the extreme case, the F-cycle is notably more sensitive. The increased sensitivity of the F-cycle in the extreme scenario could be due to the latency bound bottom solve that dominates performance in that configuration. Since the F-cycle spends relatively more time updating the coarse AMR levels (especially during the iterative bottom solves), using a layout that emphasizes good locality for that level is particularly important.

C. Impact of Asynchronous Execution

Figure 7 compares the performance of the V-cycle in the larger, balanced case under different degrees of asynchrony. There is a very large performance penalty when using the fully-synchronous execution model due the global barriers required between each computational epoch. Interestingly, the rank-synchronous model, where ranks can make progress independently of one another, produces almost all of the benefits of a fully asynchronous execution, even though it lacks the flexibility to either re-order tasks between compute epochs or begin tasks as soon as their dependent data arrives. While there is a 49 to 73 percent improvement from removing the global barriers, there is only another 3 to 4 percent improvement observed between rank synchronous and fully asynchronous. We suspect the algorithms explored here may not contain sufficient computation to hide the large network latency costs, explaining the lack of significant improvement – we will investigate this further in future work.

D. Scaling

Figure 8 presents strong scaling plots relative to 384 nodes for simulations containing 100,000 boxes. The 100 GB/s memory bandwidth curve shows reasonable scaling out to 12,288 nodes, but 1 TB/s and 10 TB/s curves, which are representative of the range of memory bandwidths that might be sustainable on exascale node architectures, show poor scaling. A possible explanation could be that the faster nodes inject traffic into the network at a faster rate, causing greater congestion in the network and slowing down progress in the computation. In SST-macro, bandwidth limits in the switches of the simulator can be turned off, eliminating congestion from the simulation. In the right-hand plot of Figure 8 congestion has been turned off and the strong scaling curves change very little, demonstrating that congestion is not a significant factor in the observed poor scaling.

E. Compute Efficiency and Idle Time

Another factor which might impact the scaling is the balancing of compute activity between compute nodes. Figure 9 shows the percent of time each rank spends computing vs waiting idly for network messages during simulations using three memory bandwidths. It is clear that, though the percentage of time devoted to computation decreases drastically as memory bandwidth increases, the distribution of compute work amongst the nodes is quite even, indicating that load imbalance is not the cause of the poor scaling behavior.

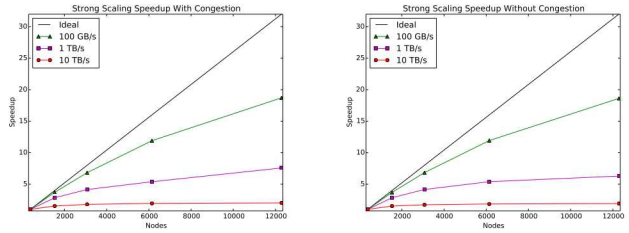


Figure 8: Strong scaling speedups for the simulated V-cycle using network models both including and ignoring traffic congestion. Results are shown for memory bandwidths ranging from current-generation (100 GB/s) to optimistic exascale (10 TB/s).

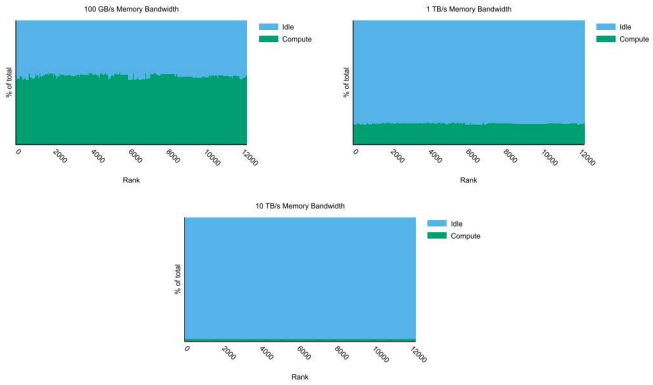


Figure 9: Percentage of time each rank spends computing versus waiting for communication to complete for three memory bandwidth values.

Figure 10 shows how the overall aggregate activity of the full system changes over time for exascale simulations varying the memory bandwidth. The significant compute sections at the beginning and end of the simulation correspond to the smooth segment of the multigrid while the largely idle section in the middle corresponds to the multigrid bottom solve, a communication heavy activity. As the memory bandwidth increases, the time spent computing decreases correspondingly, but larger and larger portions of the runtime are spent in the mostly idle bottom solve phase of the computation. These plots indicate that the explanation for the poor scaling lies in the behavior of the bottom solve.

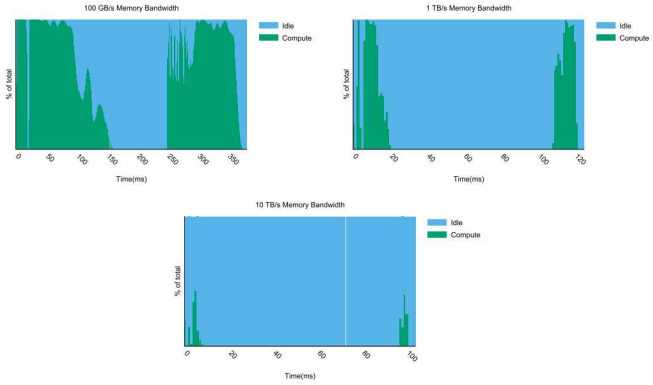
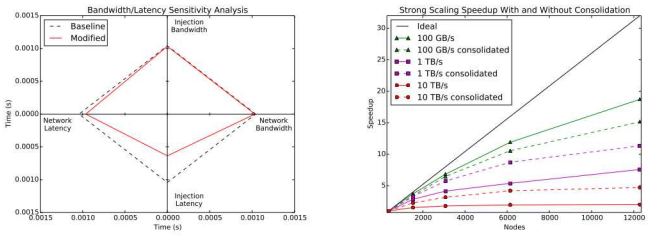


Figure 10: Percentage of time the V-cycle spends computing versus waiting for communication to complete over time for three memory bandwidth values.



(a) **(b)**

Figure 11: a) The sensivity of the multigrid V-cycle to network/injection bandwidth/latency. The dotted line represents the simulated computation time for the baseline where all parameters are fixed to representative exascale values for the network and a memory bandwidth of 10 TB/s. The intersection of the red line with each axis indicates the simulated runtime when the specified latency/bandwidth is halved/doubled keeping all other parameters fixed. b) Strong scaling speedups for the simulated V-cycle using algorithms both with and without consolidation.

For the very inefficient 12,288 node exascale simulation using 10 TB/s memory bandwidth, the sensitivity of the simulation to network and injection latencies and bandwidths was examined. These results are reported in Figre 11(a) where each axis shows the simulated time for the baseline parameters (dotted black line) compared to the simulated runtime when the indicated latency/bandwidth parameter is halved/doubled. The doubling of either injection or network bandwidth yields essentially no improvement in simulated run time; the V-cycle is not bandwidth bound for this system architecture. The latencies, conversely show improvement in simulated runtime when they are halved, with a particularly strong reduction for injection latency. Thus, we expect for the proposed exascale architecture parameters, the scaling performance is dominated by the communication-intensive bottom solve which is almost entirely latency bound.

F. Impact of Box Consolidation

Given that a large percentage of execution time is spent on the latency-bound bottom solve of the multigrid, we also briefly explore a variant of the box agglomeration technique [28] that consolidates boxes to fewer ranks as we coarsen the AMR level (but without merging them). Specifically, as we progress down the multigrid cycle, and each box is coarsened by a factor of eight (two per dimension), we migrate the boxes to one-eighth the number of ranks. This preserves the average number of grid elements each active rank is responsible for, while potentially reducing the cost of the team collectives needed for convergence detection during the bottom solve. The trade-off is the cost to migrate the coarsened boxes to the subset of ranks chosen to continue with the coarsened set of boxes. Figure 11(b) shows the impact of the box consolidation technique. For a current generation memory bandwidth of 100 GB/s the bottom solve is not a significant enough portion of the runtime to yield scaling improvements due to consolidation, while modest improvements are seen for the higher memory bandwidths expected in the exascale time frame. We will further examine the potential benefits of this optimization in greater detail in future work.

Acknowledgment

All authors from Lawrence Berkeley National Laboratory were supported by the Office of Advanced Scientific Computing Research in the Department of Energy Office of Science under contract number DE-AC02-05CH11231. Sandia National Laboratories is a multiprogram laboratory operated by Sandia Corporation, a Lockheed Martin Company, for the United States Department of Energy's National Nuclear Security Administration under contract DE-AC04- 94AL85000. This work was funded under the DOE Center for Exascale Simulation of Combustion in Turbulence (ExaCT).

References

[1] A. C. Rendleman, E. V. Beckner, M. Lijewski, W. Crutchfield, and B. J. Bell, "Parallelization of structured, hierarchical adaptive mesh refinement algorithms," *Computing and Visualization in Science*, vol. 3, no. 3, pp. 147–157, 2000. [Online]. Available: http://dx.doi.org/10.1007/PL00013544

[2] P. Colella, J. Bell, N. Keen, T. Ligocki, M. Lijewski, and B. Van Straalen, "Performance and scaling of locally-structured grid methods for partial differential equations," in *Journal of Physics: Conference Series*, vol. 78, no. 1. IOP Publishing, 2007, p. 012013.

[3] P. Kogge *et al.*, "Exascale computing study: Technology challenges in achieving exascale systems," Sept. 2008.

[4] J. D. d. S. Germain, J. McCorquodale, S. G. Parker, and C. R. Johnson, "Uintah: A massively parallel problem solving environment," in *Proceedings of the 9th IEEE International Symposium on High Performance Distributed Computing*, ser. HPDC '00. Washington, DC, USA: IEEE Computer Society, 2000, pp. 33–. [Online]. Available: http://dl.acm.org/citation.cfm?id=822085.823309

[5] L. V. Kale and S. Krishnan, "Charm++: A portable concurrent object oriented system based on c++," in *In Proceedings of the Conference on Object Oriented Programming Systems, Languages and Application*, 1993, pp. 91–108.

[6] H. Kaiser, T. Heller, B. Adelstein-Lelbach, A. Serio, and D. Fey, "Hpx: A task based programming model in a global address space," in *Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models*, ser. PGAS '14. New York, NY, USA: ACM, 2014, pp. 6:1–6:11. [Online]. Available: http://doi.acm.org/10.1145/2676870.2676883

[7] M. E. Bauer, "Legion: programming distributed heterogeneous architectures with logical regions," Ph.D. dissertation, Stanford University, 2014.

[8] G. Bosilca, A. Bouteiller, A. Danalis, T. Herault, P. Lemarinier, and J. Dongarra, "Dague: A generic distributed dag engine for high performance computing," Tech. Rep., 2010.

[9] J. Bell, A. Almgren, V. Beckner, M. Day, M. Lijewski, A. Nonaka, and W. Zhang, "BoxLib User's Guide," github.com/BoxLib-Codes/BoxLib/.

[10] C. L. Janssen, H. Adalsteinsson, S. Cranford, J. P. Kenny, A. Pinar, D. A. Evensky, and J. Mayo, "A simulator for large-scale parallel computer architectures." *IJDST*, vol. 1, no. 2, pp. 57–73, 2010.

[11] C. P. Chan, J. P. Kenny, G. Hendry, V. E. Beckner, J. B. Bell, and J. M. Shalf, "A communications simulation methodology for amr codes using task dependency analysis," in *Proceedings of the 3rd Workshop on Irregular Applications: Architectures and Algorithms*, ser. IA3 '13. New York, NY, USA: ACM, 2013, pp. 11:1–11:4. [Online]. Available: http://doi.acm.org/10.1145/2535753.2535761

[12] A. Dubey, A. Almgren, J. Bell, M. Berzins, S. Brandt, G. Bryan, D. G. P. Colella, M. Lijewski, F. Loffler, B. O'Shea, E. Schnetter, B. V. Straalen, and K. Weide, "A survey of high level frameworks in block-structured adaptive mesh refinement packages," *Journal of Parallel and Distributed Computing*, vol. 74, pp. 3217–3227, 2014.

[13] A. S. Almgren, V. E. Beckner, J. B. Bell, M. S. Day, L. H. Howell, C. C. Joggerst, M. J. Lijewski, A. Nonaka, M. Singer, and M. Zingale, "CASTRO: A New Compressible Astrophysical Solver. I. Hydrodynamics and Self-gravity," *Astrophysical Journal*, vol. 715, pp. 1221–1238, Jun. 2010.

[14] J. B. Bell, M. S. Day, A. S. Almgren, M. J. Lijewski, and C. A. Rendleman, "A parallel adaptive projection method for low mach number flows," *International Journal for Numerical Methods in Fluids*, vol. 40, no. 1-2, pp. 209–216, 2002. [Online]. Available: http://dx.doi.org/10.1002/fld.310

[15] D. Unat, C. Chan, W. Zhang, S. Williams, J. Bachan, J. Bell, and J. Shalf, "ExaSAT: An exascale co-design tool for performance modeling," *International Journal of High Performance Computing Applications*, no. 2, pp. 209–232, 2015.

[16] P. Wadler, "The essence of functional programming," in *Proceedings of the 19th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM, 1992, pp. 1–14.

[17] W. Briggs, V. Henson, and S. McCormick, *A Multigrid Tutorial, Second Edition*, 2nd ed. Society for Industrial and Applied Mathematics, 2000. [Online]. Available: http://epubs.siam.org/doi/abs/10.1137/1.9780898719505

[18] T. Hoefler and M. Snir, "Generic Topology Mapping Strategies for Large-scale Parallel Architectures," in *Proceedings of the 2011 ACM International Conference on Supercomputing (ICS'11)*. ACM, Jun. 2011, pp. 75–85.

[19] Morton, "A computer oriented geodetic data base and a new technique in file sequencing," Tech. Rep. Ottawa, Ontario, Canada, 1966.

[20] http://htor.inf.ethz.ch/research/mpitopo/libtopomap/.

[21] G. Faanes, A. Bataineh, D. Roweth, T. Court, E. Froese, B. Alverson, T. Johnson, J. Kopnick, M. Higgins, and J. Reinhard, "Cray cascade: A scalable hpc system based on a dragonfly network," in *Proceedings of the 2012 International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '12. Washington, DC, USA: IEEE Computer Society, 2012, pp. 1–9. [Online]. Available: http://dx.doi.org/10.1109/SC.2012.39

[22] B. Austin, M. Cordery, H. Wasserman, and N. J. Wright, "Performance measurements of the NERSC cray cascade system," in *Proceedings of the Cray User Group Meeting*, May 2013.

[23] E. Cuthill and J. McKee, "Reducing the bandwidth of sparse symmetric matrices," in *Proceedings of the 1969 24th National Conference*, ser. ACM '69. New York, NY, USA: ACM, 1969, pp. 157–172. [Online]. Available: http://doi.acm.org/10.1145/800195.805928

[24] J. A. George, "Computer implementation of the finite element method," Ph.D. dissertation, Stanford, CA, USA, 1971, aAI7205916.

[25] C. L. Janssen, H. Adalsteinsson, and J. P. Kenny, "Using simulation to design extremescale applications and architectures: programming model exploration," *SIGMETRICS Perform. Eval. Rev.*, vol. 38, pp. 4–8, March 2011.

[26] A. F. Rodrigues, K. S. Hemmert, B. W. Barrett, C. Kersey, R. Oldfield, M. Weston, R. Risen, J. Cook, P. Rosenfeld, E. CooperBalls, and B. Jacob, "The structural simulation toolkit," *SIGMETRICS Perform. Eval. Rev.*, vol. 38, pp. 37–42, March 2011. [Online]. Available: http://doi.acm.org/10.1145/1964218.1964225

[27] T. Nguyen, D. Unat, W. Zhang, A. Almgren, N. Farooqui, and J. Shalf, "Perilla: Metadata-based optimizations of an asynchronous runtime for adaptive mesh refinement," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, ser. SC '16, 2016.

[28] M. Emans, *Parallel Coarse-Grid Treatment in AMG for Coupled Systems*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 361–370.