Scalable Algorithms for Distributed-Memory Adaptive Mesh Refinement

Akhil Langer‡, Jonathan Lifflander‡, Phil Miller‡, Kuo-Chuan Pan*, Laxmikant V. Kale‡, Paul Ricker*
‡Department of Computer Science, *Department of Astronomy
University of Illinois at Urbana-Champaign
{alanger, jliffl2, mille121, kpan2, kale, pmricker}@illinois.edu

***Abstract*—This paper presents scalable algorithms and data structures for adaptive mesh refinement computations. We describe a novel mesh restructuring algorithm for adaptive mesh refinement computations that uses a constant number of collectives regardless of the refinement depth. To further increase scalability, we describe a localized hierarchical coordinate-based block indexing scheme in contrast to traditional linear numbering schemes, which incur unnecessary synchronization. In contrast to the existing approaches which take $O(P)$ time and storage per process, our approach takes only constant time and has very small memory footprint. With these optimizations as well as an efficient mapping scheme, our algorithm is scalable and suitable for large, highly-refined meshes. We present strong-scaling experiments up to 2k ranks on Cray XK6, and 32k ranks on IBM Blue Gene/Q.**

I. Introduction

Traditional algorithms and programming models must be reconsidered as we move forward toward the exascale era. Algorithmic tradeoffs and design decisions oriented toward bulk synchronous programming models are often infeasible in this regime, especially for irregular applications, where the structure of computation and communication is a function of input data. Algorithms that require a growing number of collective communication operations as the input size increases are inherently non-scalable because as the problem is scaled, synchronization will dominate.

Many newer paradigms decompose work into medium-grain tasks or objects that have an affinity to data instead of treating processors as fundamental first-class entities. The underlying runtime system then handles the placement and construction of these tasks during the execution, allowing the programmer to concentrate on the higher-level algorithmic design decisions. This methodology enables many runtime optimizations, but iteration times are often much shorter in duration in this execution model, so reducing synchronization points becomes paramount for achieving high performance.

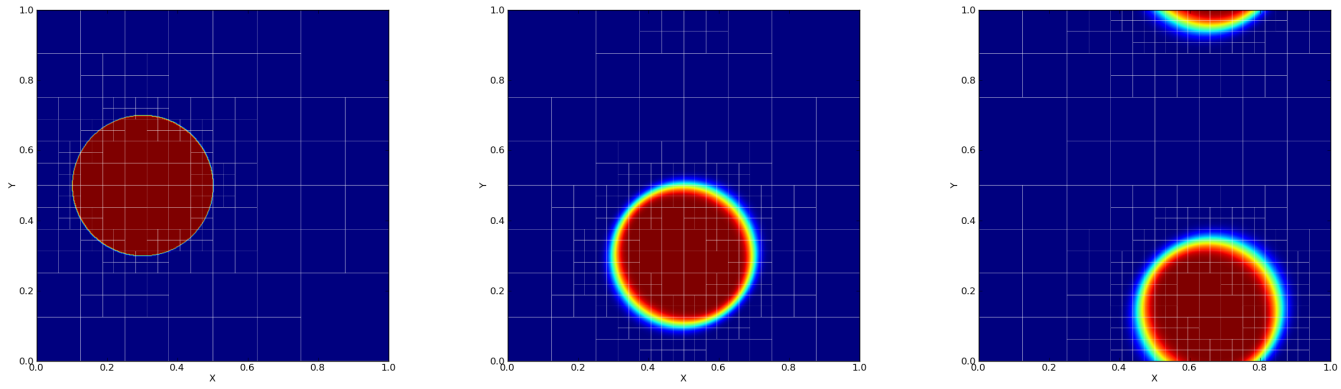
Adaptive mesh refinement is important as we look toward the future because it is a efficient method for simulating differential equations on very large meshes that arise in many domains (e.g. numerical cosmology, mantle convection modeling, global atmospheric modeling, etc.). For locating neighboring blocks and for maintaining the mesh structure invariant, traditional parallel AMR implementations store the tree information on each process. This takes $O(P)$ or more memory for storing the tree structure and $O(\log P)$ time for locating neighbors. Parallel AMR computations are often overdecomposed to obtain high performance (each processor is responsible for multiple blocks), but the mesh restructuring phases of traditional implementations require a sequence of $\mathcal{O}(d)$ rounds of collective communication, where d is the depth of mesh refinement. These collectives also require $\mathcal{O}(P)$ memory on each processor to store the results. As problem size and dynamic range increase, these costs pose a scalability bottleneck.

To address this problem, we abstract the AMR computation as a distributed collection of parallel objects that can expand and contract over time without any synchronization. This enables several advantages: first, we can define the decision-making process for the restructuring phase in terms of point-to-point communication among near-neighbor objects. Second, the collective communication is no longer proportional to the depth of recursive refinement. Instead, our remeshing algorithm's only form of global communication is a single barrier-like operation taking $\mathcal{O}(\log P)$ time to detect consensus on refinement levels among the tasks, keeping the amount of synchronization constant. The barrier communicates no data, and thus consumes a negligible amount of memory. Finally, finer grain sizes can be efficiently used to enable more overlap between communication and computation.

The primary contributions of this paper are as follows:

- A hierarchical coordinate-based block indexing scheme that takes $O(\#\text{blocks}/P)$ memory per process for storing the tree and $O(1)$ time for locating the neighboring blocks.
- We describe a parallel mesh restructuring algorithm that operates in terms of near-neighbor communication among individual blocks, and a single synchronization-only collective, avoiding the expense in time and memory of previous algorithms (§ III-B).
- We define a locally computable mapping scheme from blocks to processors that maintains a fairly even distribution of work over the course of execution so that explicit rebalancing can be performed rarely or eliminated entirely (§ III-D).
- We provide benchmark results for our methods up to thousands of cores of two modern supercomputer architectures, Cray XK6 and IBM Blue Gene/Q (§ IV-C).
- We analyze the performance and scalability of the new methods presented, and their dependence on the underlying primitives they use (§ IV-D).

Complete working code for the algorithms and benchmark



(a) (b) (c)

Fig. 1: An example simulation of the motion of a circular fluid advected by a constant velocity field. Each panel from left to right shows the fluid density after 2, 572, and 1022 iterations. The white squares show how the AMR mesh evolves over time (Min-depth = 4 and Max-depth=6). § IV-A contains a detailed description of the benchmark.

described in this paper are available online [1].

II. Related Work

We use a block-structured AMR scheme that has similar refinement structure to [2], [3]. PARAMESH [4], Burstedde et al [5] implement a design that requires each process to store the mesh structure, which requires $O(p)$ memory per process and $O(\log p)$ time per lookup of a neighboring leaf block. Burstedde et al. [6] describe a distributed AMR strategy that uses a parallel prioritized ripple propagation algorithm for mesh restructuring, causing the number of communication rounds to grow with the number of refinement levels. Each round involves message exchanges between processors and an equivalent of a system reduction to indicate the beginning of the balancing at the next level of refinement. This approach is also limited because it does not allow coarsening of sibling quadrants that are distributed across seperate processors. The SAMRAI framework [7] incurs significant overhead creating a 'communication schedule' during remeshing that also involves multiple collective communication rounds.

Bangerth at el. [8] describe a scalable design for the parts of a parallel AMR calculation *other* than the mesh generation/restructuring effort. They explicitly delegate maintenance and distribution of the mesh structure to an 'oracle' which must be able to answer queries akin to what the algorithms described in this paper provide, for which they give `p4est` [9] as an example. They describe a hierarchical mesh point identification scheme as a requirement of said oracles that matches the one we describe for mesh blocks. Our mapping scheme explicitly uses these identifiers to generate a roughly balanced mapping of mesh blocks to processors.

Our bitvector indexing and mapping scheme pushes the simplicity benefits of the forest-of-trees decomposition used by `p4est` [9], [10] into each tree, creating a completely uniform representation of element identity. Rather than splitting the elements along a Peano-Hilbert space-filling curve (or Z-order curve) [11] to load balance, which requires expensive collective communication and synchronization and imposes substantial memory usage requirements (at least one entry per rank on every rank), we use these identifiers to generate a mapping that provides a large degree of natural balance. Where that is insufficient, the Charm++ [12] runtime system underlying our implementation provides efficient object migration, hash-based lookup of migrated objects, and application-transparent forwarding and new-location notification when migrations occur. These support a wide variety of dynamic load-balancing and mapping schemes which can be applied in concert with our algorithms.

Load balancing has been studied extensively for AMR, ranging from using graph partitioning techniques [13], [14], [15], to other more AMR-specific methods [16]. We focus on building a locally computatable mapping strategy for mesh blocks that localizes work and evenly distributes it without any synchronization or periodic redistribution of work.

III. Algorithm Description

Traditional AMR algorithms are designed in terms of processors that contain many blocks. In contrast, blocks in our design are first-class entities that operate as if each resides on its own virtual processor (§ III-A). As the computation proceeds, refinement and coarsening operations expand and contract the collection of blocks. The refinement decisions are made locally by each block and then are propagated to affected neighboring blocks recursively to keep blocks within one refinement level of their neighbors (§ III-B). Because remeshing is constrained to occur at discrete points in simulation time, we use a scalable termination detection mechanism (§ III-C) to globally determine when all refinement decisions have been finalized. Besides this, blocks synchronize with each other only by the communication of boundary cells, and otherwise execute completely asynchronously.

Each block is addressed by its location in the refinement tree. The underlying runtime system provides direct communication between arbitrary blocks. We describe a mapping from

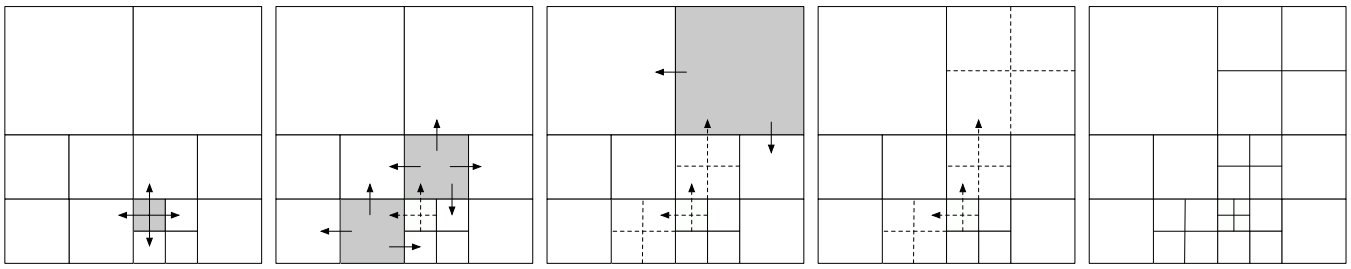


Fig. 2: Propagation of refinement decision messages, based on local error criteria and near-neighbor communication. Shaded blocks have concluded that they must refine, and send messages (solid arrows) accordingly (a-c). The path and effects of this rippling message chain are shown by dashed lines and arrows (b-d). Eventually, all the blocks reach a consensus state (e).

block addresses to processors that provides reasonable load balance and locality under the dynamic workload evolution that AMR presents (§ III-D). This avoids the need for explicitly redistributing the load during the computation.

A. Distributed Parallel Objects

To obtain high performance, AMR implementations typically partition work into k blocks for p processor cores, where $k > p$. Existing algorithms and implementations treat processors as fundamental first-class entities that explicitly manage $\frac{k}{p}$ blocks. However, the computation is local to each block or between neighboring blocks, so processor-centricity obscures the fundamental character. Our design treats each block as the basic element of a medium-grained parallel execution. Each block is expressed as an uniquely addressable object within a parallel collection that encapsulates data and methods. By taking a dynamic collection of blocks as our fundamental entity, we enable straightforward expression of the new algorithms described later in this section.

Each block in our design is a virtual endpoint of communication. Instead of addressing messages to a system rank, each message is addressed to an object that is managed by the runtime. The runtime ensures that each message is delivered to the appropriate processor where the object currently resides. Directly addressing blocks requires that they have distinct, processor-independent names that can be efficiently mapped (and possibly remapped) to a host processor. This requirement turns out to lead to other algorithmic improvements relative to existing implementations (§ III-D) and it takes only $O(N/P)$ memory per process where N is the total number of blocks and P is the total number of processes.

The block-centric formulation of our design offers several algorithmic advantages: firstly, the updates on a block's zones can begin as soon as it receives the necessary halo data from that block's neighbors. Secondly, the computation of each block's update steps can overlap with communication for all the other blocks on the same processor. Finally, a great deal of implementation complexity is spared in the application code.

Our novel algorithm relies on efficient, asynchronous messages between block objects. Each block can send a message to another block by remotely invoking a method on it with some associated data. The data is sent as a message to the appropriate processor by the runtime and executed in turn on the targeted block. Messages can be sent to currently nonexistent objects: because the block-to-processor mapping is deterministic given the block's unique address, messages can be simply buffered by the runtime on the processor where the block will be dynamically constructed. This behavior allows us to limit the amount of synchronization that is required in our algorithm.

B. Mesh Restructuring Decision Algorithm

During the course of execution, the simulated domain is expected to evolve such that some zones require finer resolution to obtain accurate results, while other zones can be safely simulated more coarsely. Like other AMR implementations, we currently make these adjustments periodically between steps of the simulation. The defining features of our algorithm are that it uses only point-to-point messages between spatially-neighboring blocks to communicate remeshing decisions, and that it synchronizes through a lightweight termination detection mechanism (§ III-C) only to determine when all blocks have reached consensus on their remeshing decisions.

When a block reaches the point in simulation time at which mesh resolution is to be reconsidered, it must decide whether it will *refine*, *stay* at its current resolution, or *coarsen* before subsequent time steps. Each block can assume as a precondition that all of its neighbors and siblings (i.e. its communication partners) start off at a refinement depth that differs from its own by at most one. To minimize the overall computational load, every block should be coarsened as much as possible. The requirement for accuracy means that any block's decision to refine or maintain its resolution will constrain its neighbors and siblings to maintain or increase their own resolution.

Figure 2 illustrates an example of how this process might proceed. Part (a) shows that a single block decides to refine (shown as shaded) based on its local error estimate and all the other blocks locally decide to maintain their current resolution. The shaded block sends messages (drawn as solid arrows) to its communication partners indicating that it intends to increase its refinement depth, and they must adjust accordingly to keep the invariant of at most one level of difference between neighbors. Parts (b) and (c) depict how this decision's effect ripple out to nearby blocks, with affected blocks downstream

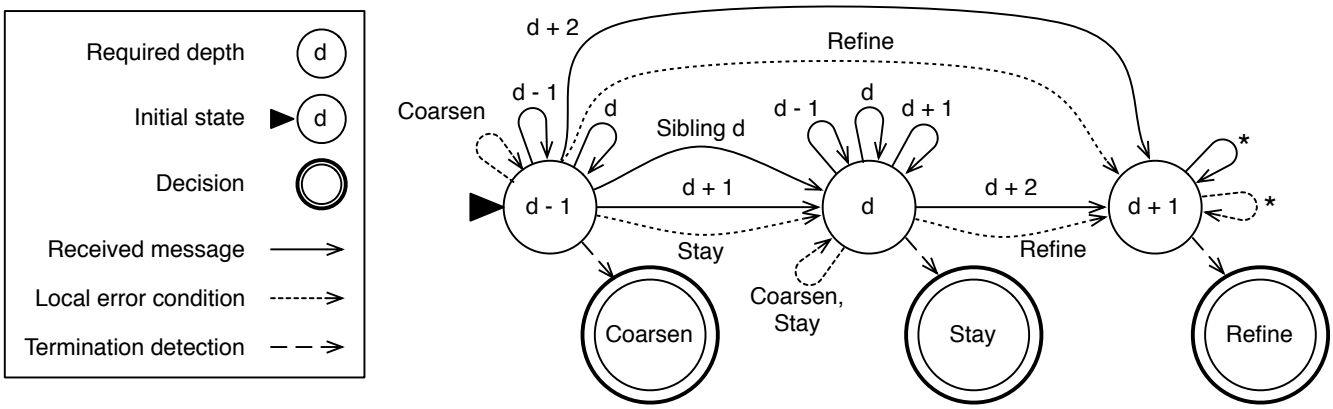


Fig. 3: The finite state machine describing each block's decision process during the mesh restructuring algorithm. A block's decision can change as a result of receiving messages from neighbors or siblings and as a result of evaluating its local error condition. When termination is detected all decisions are finalized.

(those whose resolution changes) shaded, and the path of affected blocks shown by dashed lines and arrows.

The overall algorithm that each block executes can be described by the finite state machine illustrated in Figure 3. Each d state represents a possible refinement depth for the block relative to its current depth. All of the blocks move from a d state to a decision state when termination detection indicates that they have reached consensus. The primary transitions from one state to another are driven by the receipt of messages from neighbors and siblings indicating their intended depth. Each time a block moves from one d state to another, it sends messages to each of its communication partners indicating the state that it has entered, possibly causing them to transition and communicate as well. Although blocks will try to coarsen themselves by default, any stimulus (message or local error condition) indicating a need for higher resolution will take precedence. This can be seen in the state machine's monotonic flow from coarser states toward more refined states.

Each block's machine is initialized to a state that would have it coarsen (indicated by the large triangle) as soon as its execution passes the previous cycle of remeshing decision-making. Because the blocks do not execute in lock step with one another, a block may receive messages that advance its state machine to $d+1$ and thereby constrain its decision even before it has finished timestepping to the remeshing point. This allows for a small optimization in which a block need not evaluate its local error condition if its neighbors' decisions dictate that it must refine. If a block does finish timestepping while in a state other than $d+1$, it evaluates its local error condition and follows the appropriate transition as indicated by the dotted arrows.

Note that there are no transitions that move into the $d-1$ state from another state. As a result, no block will ever send a message indicating its own intention to coarsen, and no block will receive a message indicating that a less-refined neighbor wishes to change to level $d-2$. Thus, there are no $d-2$ transitions in the state machine.

After all the decisions are finalized, blocks are created or destroyed as a result. A block that has decided to coarsen (in concert with its siblings) sends its downsampled data to its parent block and then destroys itself. A block that has decided to refine constructs four new child blocks and send a quarter of its data to each of them.

C. Termination Detection

Because refinement decisions are determined and further propagated based on distributed mesh data, detecting the global property of consensus requires termination detection. Termination is the state when no messages are in flight and all processes are idle. Many different varieties of algorithms for detecting termination are well-established in the literature [17].

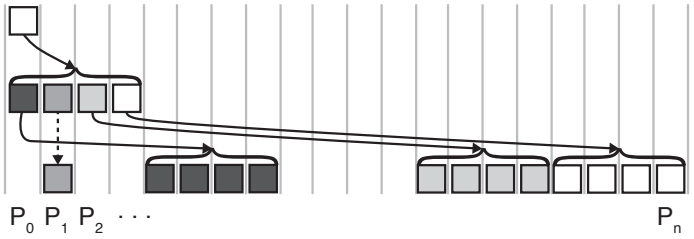
For this application, we use a wave-based four-counter termination detection algorithm that propagates waves of total send and receive message counts up and down a spanning tree that includes all the processors. When the send and receive message counts for two consecutive waves are identical, termination is detected [18]. Because waves are only propagated when a processor is otherwise idle, two identical consecutive counts indicate that no messages are in flight that could spawn more work. Only propagating waves when a processor is otherwise idle heavily reduces the number of waves that are ever started, because any busy processor will block the progression up the spanning tree. For AMR, the delay time between the last block reaching its decision and termination detection is low (empirical results are in § IV).

D. Block-to-processor Mapping and Load Balancing

In AMR, the collection of objects expands and contracts unpredictably over time, causing dynamic load imbalances to arise. Synchronized redistribution of blocks is expensive because of the high frequency of growth and shrinkage. Hence, it is important to consider locality and load balance when initially placing new objects. Because we seek to limit synchronization, the seeding function must be locally computable on any processor and deterministic, so that addressing a block is inexpensive and possible before block construction.

Input: $blockAddress$, a `bitvector` of length $2d$
$nproc$, the total number of processors
$prime$ = `0x9e37fffffffc0001`
Output: The processor $proc$ it is mapped to
let `bitvector` $base$ = $blockAddress[1:c]$;
let `int` $basePE$ = ($prime$ * $base$) >> (64 - lg($nproc$));
let `bitvector` $remainder$ = $blockAddress[c+1:2d]$;
return $(basePE + remainder)\ \%\ nproc$

(a) Mapping algorithm



(b) Mapping visualization: in the example shown, the block on P_1 chooses not to refine.

Fig. 4: The description and visualization of our block-to-processor mapping algorithm, which maps all the descendents from a base block to distinct processors (until they wrap around).

Each block's address is a bit-vector b that represents its location in the distributed refinement tree. A block of depth d in a quad-tree (oct-tree, respectively) will require an address of $2d$ ($3d$) bits, in which each pair (triplet) of bits $b_{2d+1}b_{2d}$ maps the block to a sector of the tree at depth d relative to its parent. In other words, it describes the path taken by a recursive traversal through the tree from a nominal root to the block in question. The function we define takes this sequence of bits and maps it to a host processor.

The AMR computation initially starts with a collection of blocks at a constant depth c. Because block refinements tend to be spatially correlated, we initially seed the set of blocks with a uniform random distribution (using an inexpensive hashing function) over the set of processors. The mapping function takes the first c bits, applies the hash, and uses this value as the base processor for this block and all its descendents. The remaining $d - c$ bits are then used as an integer offset from the base processor. The algorithm is detailed in Figure 4a.

By treating the $d - c$ bits as an integer offset from the base processor, all the descendents of a base block will be mapped to different processors until this offset wraps around. Figure 4b visualizes this effect.

IV. Experimental Results

A. Advection Benchmark

To empirically test our AMR remeshing algorithm, we benchmark a finite-difference simulation of advection, described by the following hyperbolic partial differential equation:

$$\frac{\partial u}{\partial t} + v\nabla u = 0 \tag{1}$$

The advection equation is common in chemistry and describes the advection of a tracer along with the fluid. The density (or concentration) u is the conserved quantity with a bulk motion speed v. In our simulation, v is held constant. We solve the advection equation using a first-order upwind method in two-dimensional space. Although our algorithm is applied to a first-order scheme, our AMR framework can easily be adapted to higher-order multi-dimensional schemes and other hyperbolic problems.

We initialize the simulation with a circular region of density $u = 2$, ambient density $u = 1$, and bulk velocity $v = 1$, with periodic boundary conditions (Figure 1). The error is estimated using the second derivative of the density u, as described by Löhner [19].

B. Experimental Setup

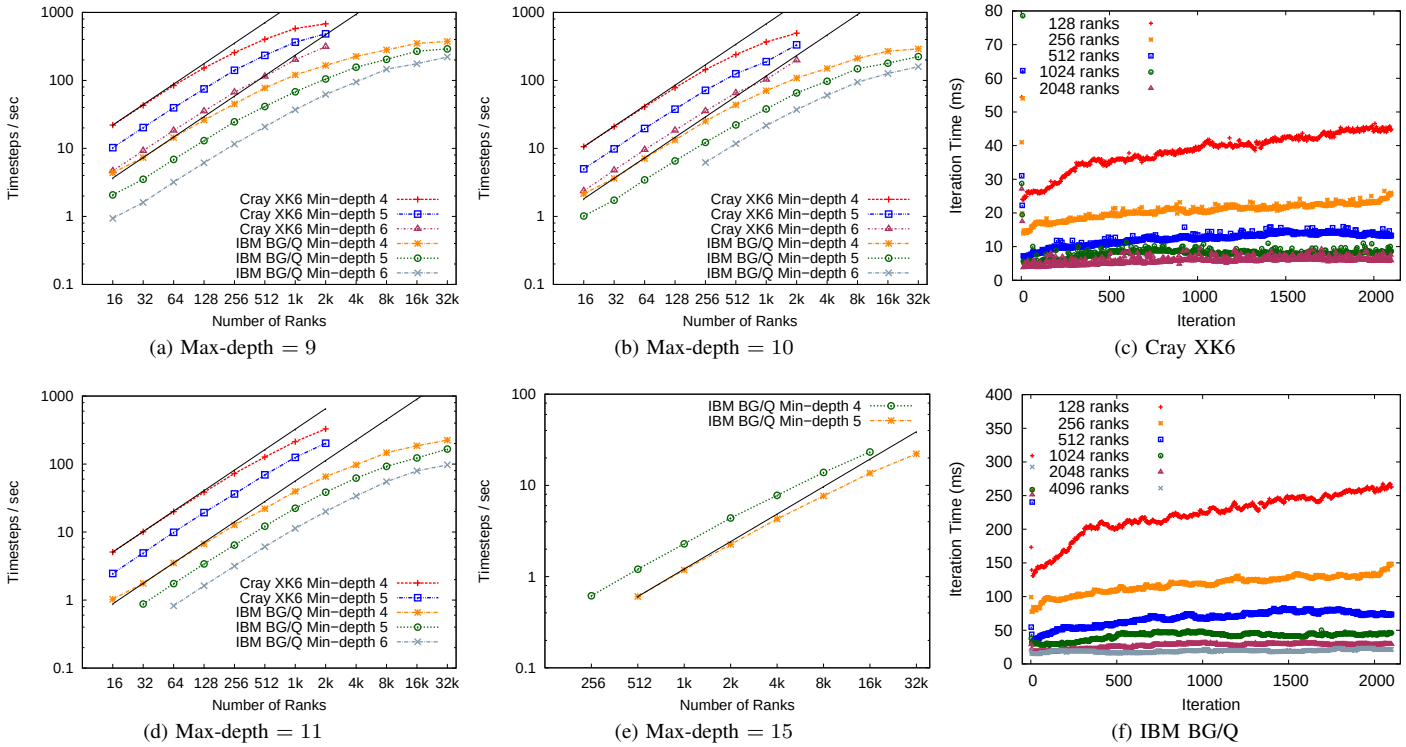
The experiments were performed on two systems: Cray XK6 'Titan' and IBM Blue Gene/Q 'Vesta'. Each node of Titan consists of one sixteen-core 2.2GHz AMD 'Bulldozer' processor and 32GB DDR3 memory. Only the CPU part of Titan was used for our runs, with no GPU acceleration. Nodes are connected by the Gemini interconnection network with a 9.8GB/s peak bandwidth per Gemini chip. Our experiments ran with 16 ranks on each node of Titan. Each node of Vesta consists of one 1.6 GHz PowerPC A2 processor with 16 application cores supporting 4-way simultaneous multithreading and 16 GB DDR3 memory. Our experiments ran with 32 ranks on each node of Vesta, using 2-way SMT per core.

Our code uses the Charm++ runtime system [12], which supports dynamic collections of parallel objects in the form of *chare arrays* [20]. We used the Gemini machine layer in Charm++ for Cray XK6 and the PAMI (Parallel Active Messaging Interface) machine layer for BG/Q. Our code was compiled with the GNU compiler suite version 4.6.2 on Titan and version 4.4.6 for Vesta (Blue Gene/Q release BGQ-V1R1M1-120628).

C. Overall Performance and Scalability

Our benchmark application performs relatively little calculation in each time step, and remeshes in a cycle of every two timesteps. We chose this configuration in order to both highlight and stress test the remeshing algorithm.

Our benchmark results can be seen in Figure 5, which plots the time taken for each cycle over the course of a run. As one would expect, step times scale down with processor count. The upward trend in cycle time seen on the smaller runs can be attributed to slowly-growing load imbalance as the highest-resolution zones shift across the problem domain. The smaller runs are more severely impacted by this because of the effect of blocks descended from different roots being mapped to overlapping processors. At larger scales, where the root blocks are more widely spread over the whole system, this overlap effect diminishes. An explicit dynamic load balancing



(a) Max-depth = 9 (b) Max-depth = 10 (c) Cray XK6

(d) Max-depth = 11 (e) Max-depth = 15 (f) IBM BG/Q

Fig. 5: Columns 1 & 2: timesteps per second strong scaling on Cray XK6 and IBM BG/Q with various minimum depths; column 3: the duration in milliseconds for each cycle with a max-depth = 10 (each composed of two timesteps and a remeshing operation).

mechanism could be run periodically to mitigate this effect. However, in the current work, we have found this to not provide sufficient benefits.

An overall view of our code's strong scaling behavior can be seen in Figure 5. We depict strong scaling curves, each representing a dynamic range of refinement. A minimum depth of 4 represents a coarsest mesh dimension of 256^2, which quadruples to 512^2 and 1024^2 at depths 5 and 6 respectively. The black lines indicate ideal scaling on each machine relative to the performance of a whole single node. The ideal scaling lines for Blue Gene/Q are drawn through the 32-core point to reflect the higher-performance 2-way symmetric multi-threaded mode in which that and all larger runs were performed.

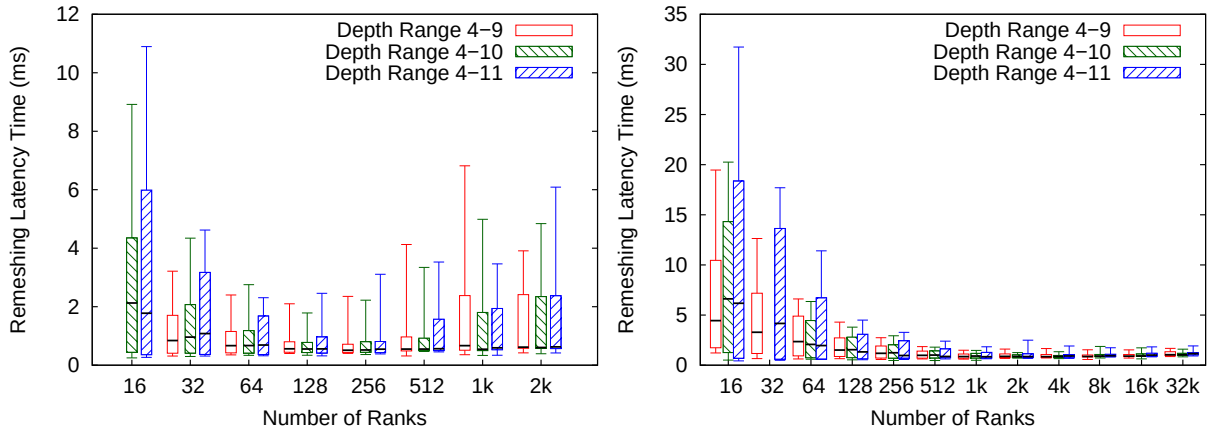
When scaling from 16 ranks on Cray XK6 with a minimum depth of 5 and maximum depth of 11 (as shown in Figure 5d), we are able to achieve 80% parallel efficiency up to 1024 ranks (up to 30 ms/cycle), and 64% parallel efficiency at 2048 ranks (up to 19 ms/cycle). Note that we run with a wide range of refinement levels and increasing dynamic range does not reduce efficiency. Instead, for a given minimum depth, increasing the maximum depth actually increases efficiency: on 2048 ranks our code attains 36% parallel efficiency with a depth ranging from 5–9 and increases to 64% parallel efficiency with a depth ranging from 5–11. Although our remeshing scheme requires deeper propagation with a wider depth range, it does not dominate and the increase in work leads to an increase in efficiency.

On IBM BG/Q, scaling from 64 ranks to 2048 with a depth range of 6–11, our code achieves 76% parallel efficiency (up to 182 ms/cycle), and when it's pushed to the limit of machine size to 32768 ranks (one rack of BG/Q at SMT 2), it attains 23% parallel efficiency (up to 28 ms/cycle). Upon scaling from 512 to 32k ranks and allowing the depth range to vary from 5–15, we get much higher efficiencies of 99%, 95%, 65%, 55% at 2k, 8k, 16k and 32k ranks, respectively (Figure 5e).

D. Remeshing Performance

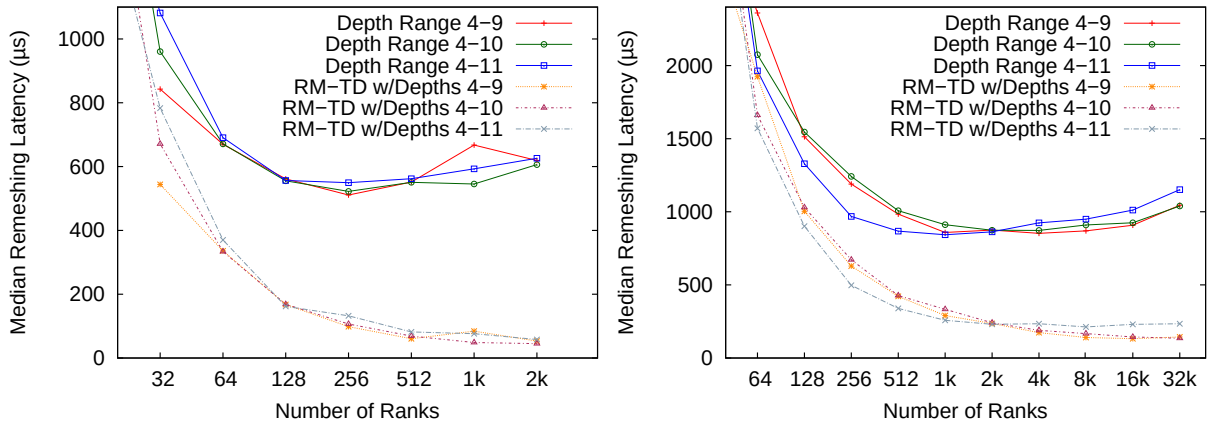
In Figure 6, we graph the distribution of remeshing latencies, that is the time interval between the last processor beginning remeshing and the start of the next timestep. This measures the duration spent in remeshing with no overlapping computation. The general trend is that remeshing latency scales down with the number of processors out to a strong scaling limit (about 256 ranks on XK6, and 1024 ranks on BG/Q).

Its performance is actually bounded by two different factors: the communication necessary to make all of the remeshing decisions, and the delay in synchronizing through termination detection after consensus is reached. The first factor dominates at low processor counts, but scales downward as it gets distributed over a larger number of processors. This is easiest to see in Figure 6b, where the maximum, 95th percentile, and median times all descend smoothly from 16 ranks to 1024 ranks. To examine the cross-over behavior into the second factor dominance, Figure 7 shows the trend in median



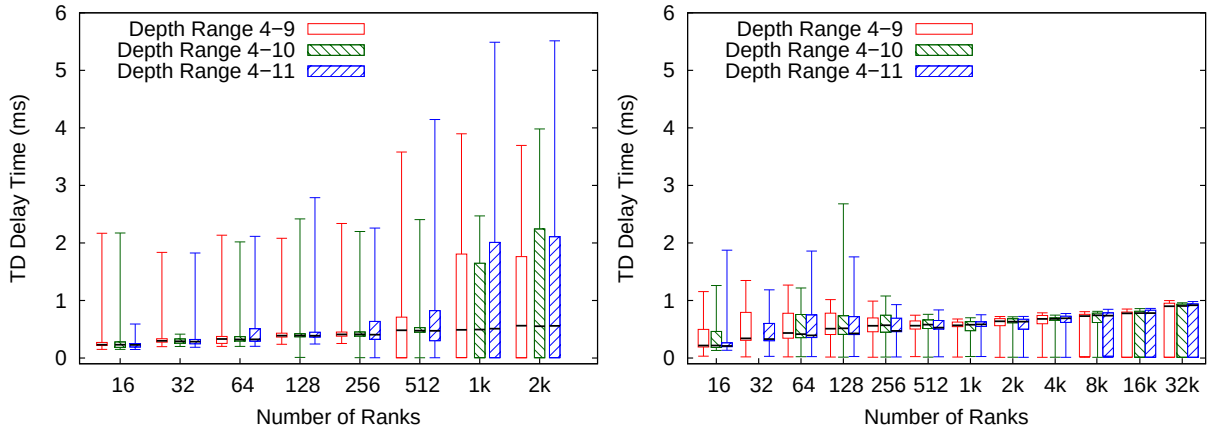
(a) Cray XK6 (b) IBM BG/Q

Fig. 6: The remeshing latency in milliseconds: the non-overlapped delay that remeshing causes in the computation, i.e. the time from the end of non-remeshing work on the last processor to the beginning of the next timestep. The vertical lines stretch between the minimum and maximum values; the box spans between the 5th and 95th percentile; the horizontal line spanning the box indicates the median.



(a) Cray XK6 (b) IBM BG/Q

Fig. 7: The median remeshing latencies (mid-points from Figure 6) in microseconds are graphed as the upper three solid lines. The latency scales down with processor count until it becomes synchronization bound by termination detection. The lower three dotted lines represent the difference between the median remeshing latency and median termination detection delay, demonstrating that remeshing time is dominated by termination at larger scales.



(a) Cray XK6 (b) IBM BG/Q

Fig. 8: The delay time in milliseconds for termination detection. This is measured as the duration between the last work unit executed on any core and the start of the next timestep. The vertical lines stretch between the minimum and maximum values; the box spans between the 5th and 95th percentile; the horizontal line spanning the box indicates the median.

remeshing times for each set of runs in solid lines, starting from a slightly higher core count to make the slow increase at larger scales apparent.

The wave-based termination detection algorithm as described in § III-C has a theoretical upper-bound *delay time* that scales logarithmically with the number of processors, because it uses broadcasts and reductions over a spanning tree. We measure the delay time as the time interval between the last processor processing an application message and it receiving a broadcast indication that consensus has been reached. This duration is graphed in Figure 8. The median remeshing times at larger scale approach a constant offset above the median termination detection delay times as shown by the dotted lines in Figure 7. This demonstrates that termination accounts for the slight trend upward in remeshing latency at larger scales.

Overall, these trends show that our remeshing algorithm is not limited by the performance of collective data exchange and has no readily apparent dependence on the depth of the refinement.

V. Conclusion and Future Work

Efficient mesh restructuring is an important problem in AMR computation both for its use in scientific applications and for the challenges it poses as parallel computing reaches to ever larger scales. Existing approaches incur a number of scalability bottlenecks in their use of collective communication to organize, adapt, and distribute the work of the simulation.

We presented a scheme that eliminates the need to store the tree structure on each process. Traditional algorithms combine the data exchange necessary to adjust the mesh to match the evolving problem domain with the synchronization inherent in collective communication operations. By invoking these routines multiple times, they incur substantial overhead. We show that by reformulating the communication asynchronously, the synchronization is mostly obviated, and the necessary synchronization can be attained through a separate, less-costly mechanism. In the process, we also eliminate the per-processor memory expense of collectives. As we push our applications past their current scalability limits, we can see that problem formulation in terms of local data-flow dependences and minimal synchronization is essential.

Existing AMR implementations explicitly redistribute work frequently to obtain good load balance, and often incur high data movement and synchronization costs to do so. This takes substantial time and energy, and does not directly advance the computation itself. We take a different approach, examining a mechanism to map the workload of an AMR computation to processors of a parallel machine in a roughly load-balanced fashion without imposing additional interaction between processors.

Acknowledgements

The authors were supported by grants MITRE Research Agreement No. 81990, NSF ITR-HECURA-0833188, NSF OCI-0725070. This research used resources of the Oak Ridge Leadership Computing Facility located in the Oak Ridge National Laboratory, which is supported by the Office of Science of the Department of Energy under Contract DE-AC05-00OR22725. This research used resources of the Argonne Leadership Computing Facility at Argonne National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under contract DE-AC02-06CH11357.

References

[1] "AMR algorithm and benchmark source code," 2012. Source code and scripts available at `git://charm.cs.illinois.edu/benchmarks/amr.git`.

[2] D. DeZeeuw and K. G. Powell, "An adaptively refined cartesian mesh solver for the euler equations," *JCP*, vol. 104, p. 56, 1993.

[3] P. MacNeice, K. M. Olson, C. Mobarry, R. de Fainchtein, and C. Packer, "Paramesh: A parallel adaptive mesh refinement community toolkit," *Computer Physics Communications*, vol. 126, pp. 330–354, 2000.

[4] M. Parashar, X. Li, and S. Chandra, *Advanced Computational Infrastructures for Parallel and Distributed Adaptive Application*. Wiley-Interscience, 2009.

[5] T. Tu, D. O'Hallaron, and O. Ghattas, "Scalable parallel octree meshing for terascale applications," in *Supercomputing, 2005. Proceedings of the ACM/IEEE SC 2005 Conference*, pp. 4–4, IEEE, 2005.

[6] C. Burstedde, O. Ghattas, M. Gurnis, G. Stadler, E. Tan, T. Tu, L. Wilcox, and S. Zhong, "Scalable adaptive mantle convection simulation on petascale supercomputers," in *High Performance Computing, Networking, Storage and Analysis, 2008. SC 2008. International Conference for*, pp. 1–15, IEEE, 2008.

[7] A. Wissink, R. Hornung, S. Kohn, S. Smith, and N. Elliott, "Large scale parallel structured AMR calculations using the SAMRAI framework," in *Supercomputing, ACM/IEEE 2001 Conference*, p. 22, November 2001.

[8] W. Bangerth, C. Burstedde, T. Heister, and M. Kronbichler, "Algorithms and data structures for massively parallel generic adaptive finite element codes," *ACM Trans. Math. Softw.*, vol. 38, pp. 14:1–14:28, Jan. 2012.

[9] 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*, vol. 33, no. 3, pp. 1103–1133, 2011.

[10] C. Burstedde, O. Ghattas, M. Gurnis, T. Isaac, G. Stadler, T. Warburton, and L. Wilcox, "Extreme-scale AMR," in *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '10, (Washington, DC, USA), pp. 1–12, IEEE Computer Society, 2010.

[11] C. Faloutsos and S. Roseman, "Fractals for secondary key retrieval," in *Proceedings of the eighth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, pp. 247–252, ACM, 1989.

[12] L. Kalé and S. Krishnan, "CHARM++: A Portable Concurrent Object Oriented System Based on C++," in *Proceedings of OOPSLA'93* (A. Paepcke, ed.), pp. 91–108, ACM Press, September 1993.

[13] L. Oliker and R. Biswas, "PLUM: Parallel load balancing for adaptive unstructured meshes," *Journal of Parallel and Distributed Computing*, vol. 52, no. 2, pp. 150 – 177, 1998.

[14] Ü. Çatalyürek, E. Boman, K. Devine, D. Bozdag, R. Heaphy, and L. Riesen, "Hypergraph-based dynamic load balancing for adaptive scientific computations," in *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International*, pp. 1 –11, March 2007.

[15] K. Schloegel, G. Karypis, and V. Kumar, "A unified algorithm for load-balancing adaptive scientific simulations," in *Supercomputing, ACM/IEEE 2000 Conference*, p. 59, November 2000.

[16] H. deCougny, K. Devine, J. Flaherty, R. Loy, C. zturan, and M. Shephard, "Load balancing for the parallel adaptive solution of partial differential equations," *Applied Numerical Mathematics*, vol. 16, no. 12, pp. 157 – 182, 1994.

[17] J. Matocha and T. Camp, "A taxonomy of distributed termination detection algorithms," *Journal of Systems and Software*, vol. 43, no. 3, pp. 207 – 221, 1998.

[18] A. B. Sinha, L. V. Kale, and B. Ramkumar, "A dynamic and adaptive quiescence detection algorithm," Tech. Rep. 93-11, Parallel Programming Laboratory, Department of Computer Scie nce , University of Illinois, Urbana-Champaign, 1993.

[19] R. Löhner, "Finite elements in CFD: What lies ahead," *International Journal for Numerical Methods in Engineering*, vol. 24, no. 9, pp. 1741–1756, 1987.

[20] O. S. Lawlor and L. V. Kalé, "Supporting dynamic parallel object arrays," *Concurrency and Computation: Practice and Experience*, vol. 15, pp. 371–393, 2003.