AMR-aware In Situ Indexing and Scalable Querying

Xiaocheng Zou
North Carolina State University, Raleigh, NC, USA
Oak Ridge National Laboratory, Oak Ridge, TN, USA
xzou2@ncsu.edu

David A. Boyuka II
North Carolina State University, Raleigh, NC, USA
Oak Ridge National Laboratory, Oak Ridge, TN, USA
daboyuka@ncsu.edu

Dhara Desai
North Carolina State University, Raleigh, NC, USA
Oak Ridge National Laboratory, Oak Ridge, TN, USA
dadesai@ncsu.edu

Daniel F. Martin
Lawrence Berkeley National Laboratory
Berkeley, CA, USA
dfmartin@lbl.gov

Suren Byna
Lawrence Berkeley National Laboratory
Berkeley, CA, USA
sbyna@lbl.gov

Kesheng Wu
Lawrence Berkeley National Laboratory
Berkeley, CA, USA
KWu@lbl.gov

ADDITIONAL AUTHORS

1. Kushal Bansal, North Carolina State University, Raleigh, NC, USA, Oak Ridge National Laboratory, Oak Ridge, TN, USA, kbansal@ncsu.edu
2. Bin Dong, Lawrence Berkeley National Laboratory, Berkeley, CA, USA, dbin@lbl.gov
3. Wenzhao Zhang, North Carolina State University, Raleigh, NC, USA, Oak Ridge National Laboratory, Oak Ridge, TN, USA, wzhang27@ncsu.edu
4. Houjun Tang, North Carolina State University, Raleigh, NC, USA, Oak Ridge National Laboratory, Oak Ridge, TN, USA, htang4@ncsu.edu
5. Dharshi Devendran, Lawrence Berkeley National Laboratory, Berkeley, CA, USA, pdevendran@lbl.gov
6. David Trebotich, Lawrence Berkeley National Laboratory, Berkeley, CA, USA, dptrebotich@lbl.gov
7. Scott Klasky, Oak Ridge National Laboratory, Oak Ridge, TN, USA, klasky@ornl.gov
8. Hans Johansen, Lawrence Berkeley National Laboratory, Berkeley, CA, USA, hjohansen@lbl.gov
9. Nagiza F. Samatova (corresponding author), North Carolina State University, Raleigh, NC, USA, Oak Ridge National Laboratory, Oak Ridge, TN, USA, samatova@csc.ncsu.edu

Spring Simulation Multi-Conference 2016 April 3-6, Pasadena, CA, USA

ABSTRACT

Query-driven analytics on scientific datasets is one of fundamental approaches for scientific discoveries. Existing studies have explored query-driven analytics on uniform resolution meshes. However, querying on adaptive mesh refinement (AMR) data has not been explored yet. As many simulations have been lately transitioning to AMR, new methods for efficient query-driven analysis on AMR data are needed.

In this paper, we present the first work to support scalable AMR-aware analysis. We propose an AMR-aware hybrid index for supporting two common forms (i.e., spatial and value-based query selections) in query-driven analytics. To sustainably support future-scale analysis, we design an *in situ* (run-time) index building strategy with minimized performance impact to the co-located simulation. Additionally, we develop a parallel post-processing query method with an adaptive workload-balanced strategy. Our evaluation demonstrates the scalability of our *in situ* indexing and scalable querying methods up to 16,384 and 1,024 cores, respectively, using a Chombo-based benchmark. Compared to non-AMR-aware indexing and querying, we demonstrate up to 12.4x and 500x performance improvement, respectively.

Author Keywords

AMR; In Situ; Index; Query; Parallel; HPC

1. INTRODUCTION

Adaptive Mesh Refinement (AMR) [3] is an advanced numerical method that dynamically and adaptively refines the spatial accuracy of a solution where needed during the simulation run. The key factor to its success is its adaptive ability that dynamically refines simulation resolution across space and time, which yields a hierarchical, multi-level, and multi-resolution mesh (Fig. 1(a)). Many scientific simulations in various domains are thus transitioning or have transitioned to AMR. As a result, scientific analysis techniques, such as visualization, feature detection, and feature tracking, must make the transition to match.

Query-driven analysis (generally performed through SQL-like queries) is a fundamental approach for discovering scientific phenomena. For example, in climate science, detection of an extreme climate event called an Atmospheric River is expressed as a query using a spatial selection and a value constraint [5]. Furthermore, query-driven analysis is an essential approach to deal with the data deluge in various science fields. As computational capabilities are moving towards the exascale, simulations produce tremendous data. Searching for critical information in these massive data requires fast query-driven analysis. Thus, multiple efficient query-driven analysis techniques, such as ALACRITY [11], DIRAQ [14], FastBit [16], and FastQuery [7], have been developed recently.

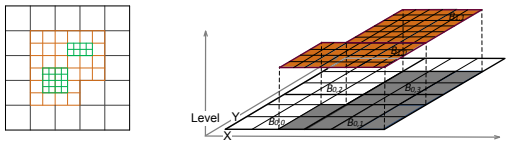
However, directly applying these existing techniques on AMR data is inefficient as they are designed for single, uniform meshes. For example, one can refine all mesh levels to a finest resolution to form a uniform grid (referred as "flattening"), allowing the direct application of existing techniques without special modification. Yet, the "flattening" approach defeats the purpose of AMR because it increases the computing, memory, and storage requirements explosively. This is untenable for large-scale scientific data with dozens of variables. Therefore, there is a need to *enable querying capabilities directly on AMR data by embracing the AMR structure.*

Developing an AMR-aware querying, however, is a non-trivial task, as it necessitates a completely new indexing methodology. Specifically, we need a new indexing method capable of handling the hierarchical, non-uniform AMR data that capture both value and space aspects at the same time. Furthermore, as demonstrated in the early work [12, 14], an effective and sustainable indexing technique towards future large-scale computation needs to operate in the context of *in situ* where the indexing runs concurrently with the simulation run. Hence, we advocate for an *in situ, scalable, and AMR-aware indexing.*

While appealing, this shift in indexing technique presents several challenges. First, as a general *in situ* algorithm, *in situ* indexing should have minimum overhead so as to not affect the simulation run. Second, the AMR-awareness further complicates this *in situ* algorithm because the AMR hierarchy is scattered across processes, and is typically both irregular and dynamically changing over time. Third, developing an AMR-aware querying (assuming indexes have been built) needs an efficient, parallel query processing strategy in order to support large-scale scientific discoveries. Specifically, the challenge centers around achieving balanced querying workload in the parallel context in presence of: 1) the irregular index read access patterns invoked by unpredictable queries; and 2) unbalanced AMR index sizes across levels.

To address all these challenges, we propose the *first* AMR-aware *in situ* indexing and scalable querying methodology for scientific exploration and analysis on the AMR data. Our contributions are as follows:

- Formally define the general problem of query-driven analysis for AMR data (Section 2). This formal definition paves the way for our systematic study of AMR indexing and querying problem.
- Design an *in situ*, query-optimized, AMR-aware indexing method (Section 3.2) that efficiently builds an AMR-specific index in a massively parallel manner.
- Develop a parallel, *AMR-index-aware, workload-balanced* query processing method (Section 3.3).
- Integrate both methods within the Chombo AMR infrastructure [1], this way we make our AMR-specific analysis services available for many existing Chombo-based applications.



(a) (b)

Figure 1. Subfigure (a) shows an example of a 3-level AMR mesh with refinement ratio 2: from the coarsest to the finest level. Subfigure (b) illustrates a 2-level AMR hierarchy, with three box types: 1) $B_{0,2}$, $B_{1,0}$, and $B_{1,1}$ are uncovered; 2) $B_{0,0}$ and $B_{0,3}$ are partially covered; and 3) $B_{0,1}$ is fully covered.

We demonstrate the scalability of our *in situ* indexing using up to 16,384 cores, and that of our querying using up to 1,024 cores with the Chombo-IO benchmark [6]. When compared to the non-AMR-aware indexing and querying, our AMR-aware indexing and querying demonstrate up to 12.4x and 500x, respectively.

2. PROBLEM STATEMENT

We now present formal definitions for a scalable, *in situ* indexing component and a parallel, post-processing querying component with respect to AMR-structured data. First, we define the *coarsening* and *refinement* operators, which will be helpful later[1]:

Definition 1 The *coarsening operator* $C : \mathbb{Z}^D \rightarrow \mathbb{Z}^D$ is defined as $C(\boldsymbol{i}, r) = (\lfloor \frac{i_0}{r} \rfloor, \lfloor \frac{i_1}{r} \rfloor, \ldots, \lfloor \frac{i_{D-1}}{r} \rfloor)$, where $r \in \mathbb{Z}^+$ is a given *refinement ratio*. Correspondingly, we define the inverse *refinement operator* $R : \mathbb{Z}^D \rightarrow \mathcal{P}(\mathbb{Z}^D)$ as $R(\boldsymbol{i}, r) = \{\boldsymbol{i'} \in \mathbb{Z}^D \mid C(\boldsymbol{i'}, r) = \boldsymbol{i}\}$. Equivalently, $R(\boldsymbol{i}, r) = \prod_{j=0}^{D-1} \{ri_j, ri_j + 1, \ldots, ri_j + r - 1\}$, where $\prod$ is the N-ary Cartesian product. These operators can be naturally extended to operate on sets of vectors.

Next, we define the components of an AMR structure.

Definition 2 A *box* is a set of integer coordinates within (inclusive) lower and (exclusive) upper corners $\boldsymbol{x}, \boldsymbol{y} \in \mathbb{Z}^D$ with $\boldsymbol{0} \leq \boldsymbol{x} < \boldsymbol{y}$, where $\boldsymbol{0}$ denotes the zero vector and $<$ ($\leq$) denotes component-wise comparison of two vectors. That is, a box $B = \{\boldsymbol{i} \in \mathbb{Z}^D \mid \boldsymbol{x} \leq \boldsymbol{i} < \boldsymbol{y}\}$.

Definition 3 An *AMR structure* Ω with $L \in \mathbb{Z}^+$ levels, domain bound $\boldsymbol{u} \in (\mathbb{Z}^+)^D$, and refinement ratio $r \in \mathbb{Z}^+$ con-

[1]Definitions 1, 3, and 4, adapted from the Chombo design document [1], were used in our previous work [17].

sists of a list of *AMR levels* $\Omega_0, \ldots, \Omega_{L-1}$. The first (coarsest) level is a box with bounds $\mathbf{0}, \boldsymbol{u}$: $\Omega_0 = \{\boldsymbol{i} \in \mathbb{Z}^D \mid \mathbf{0} \leq \boldsymbol{i} < \boldsymbol{u}\}$. For subsequent levels, $\Omega_l \subset R(\Omega_{l-1}, r)$. Finally, every Ω_l can be decomposed into a disjoint union of m_l boxes $B_{l,k}$ such that $\Omega_l = \cup_{k=0}^{m_l-1} B_{l,k}$ and $B_{l,x} \cap B_{l,y} = \emptyset$ when $x \neq y$. We denote the set of all boxes composing an AMR structure Ω as B_Ω.

In other words, an AMR structure is a hierarchy of mesh levels, with the coarsest level covering the entire computational domain, and successively finer levels covering portions of the next-coarser level. All mesh cells on a given AMR level have the same mesh spacing, which is reduced by a given *refinement ratio* relative to the next-coarser level. Additionally, in the specific case of *block-structured AMR* considered in this paper, each level's mesh can always be decomposed into a set of non-overlapping rectangular boxes. Figure 1 demonstrates these definitions.

Definition 4 A *cell* at level l of AMR structure Ω is defined as a pair $(l, \boldsymbol{i})$ with $\boldsymbol{i} \in \Omega_l$. The set of all cells in an AMR hierarchy Ω is denoted by $\sigma(\Omega) = \cup_{l=0}^{L-1}\{(l, \boldsymbol{i}) \mid \boldsymbol{i} \in \Omega_l\}$.

Definition 5 The *uncovered predicate* $U : \sigma(\Omega) \rightarrow \{true, false\}$ determines whether a given cell $c = (l, \boldsymbol{i})$ is *uncovered*, and is defined as

$$U(c) = \begin{cases} true & if\ l = L-1\ or\ \forall \boldsymbol{i'} \in \Omega_{l+1}\ C\left(\boldsymbol{i'}, r\right) \neq \boldsymbol{i} \\ false & else \end{cases}$$

Informally, we call a cell *uncovered* if no cells at the next finer level are "on top of it" (i.e., coarsen down to the given cell), or the given cell is already at the finest level.

We now give two definitions in preparation for defining the AMR query-driven analysis we aim to support.

Definition 6 A *spatial selection* over an AMR structure Ω is a box θ with corners $\boldsymbol{x}, \boldsymbol{y}$ such that $\mathbf{0} \leq \boldsymbol{x} < \boldsymbol{y} < \boldsymbol{u}$. A spatial selection is intended to be interpreted within the coordinate system of the coarsest AMR level, Ω_0. A cell $c = (l, \boldsymbol{i})$ is said to be *within* spatial selection θ iff $C\left(\boldsymbol{i}, r^l\right) \in \theta$, that is, iff its coordinates fall within the spatial selection box when coarsened down to the coarsest AMR level. We denote the set of all possible spatial selections over Ω as Θ_Ω.

Definition 7 An AMR query-driven analysis is a function $\mathcal{Q}_\Omega : \boldsymbol{I}_\mathbb{V} \times \Theta_\Omega \rightarrow \mathcal{P}(\sigma(\Omega) \times \mathbb{V})$ defined as $\mathcal{Q}_\Omega(\boldsymbol{I}, \theta) = \{(c, v) \in \sigma(\Omega) \times \mathbb{V} \mid U(c) \wedge c\ within\ \theta \wedge v \in \boldsymbol{I}\}$, where $\mathbb{V}$ is a *value domain set* in Ω and $\boldsymbol{I}_\mathbb{V}$ is a set of all intervals in $\mathbb{V}$.

That is, for a given AMR structure Ω, a spatial selection θ and a value interval $\boldsymbol{I}$ over a dependent variable with domain $\mathbb{V}$, the query-driven analysis function returns a set of cell-value pairs such that 1) all cells are *uncovered*, 2) all cells are *within* the given spatial selection, and 3) the value of the dependent variable at each cell falls within the given value interval. The reason we are interested only in uncovered cells is that covered cells generally bear less-accurate values, often computed by simply averaging down values from finer levels, and thus are not interesting or useful.

Now we consider define the query-driven analysis $\mathcal{Q}_\Omega(\boldsymbol{I}, \theta)$ with an index X as an auxiliary data structure. Before formally defining X, we introduce one more concept.

Definition 8 A *box cover finding* operation is a function $\Gamma : B_\Omega \rightarrow B_\Omega$ such that $\Gamma(B_{l,k}) = \{B_{l+1,n} \mid R(B_{l,k}, r) \cap B_{l+1,n} \neq \emptyset\}$

In other words, given a box $B_{l,k}$, the box cover finding operation returns the set of boxes on level $l+1$ such that when the given box is refined to the next fine level, all boxes comprise the refined box. B is uncovered if it is either at the finest AMR level, or all cells in B are uncovered; B is partially covered if some, but not all, of its cells are covered; and B is fully covered if all of its cells are covered. Figure 1(b) exemplifies all three types of boxes.

Now, we describe our *AMR-aware index* X as a data structure (or several data structures) that facilitates the following operations on an AMR structure Ω: 1) fast identification of all boxes within a given spatial selection θ; 2) fast computation of the box cover finding operation $\Gamma(B)$ to identify the uncovered/partially covered boxes from step 1, and which cells within these boxes are uncovered; and 3) fast selection of cells within the boxes from step 2 whose values fall within a given value interval.

These three steps correspond to the three conjuncts in the definition of query-driven analysis function $\mathcal{Q}_\Omega$ given in Definition 7: we select the cells within θ, that are uncovered, and whose values fall within $\boldsymbol{I}$. The necessity of steps 1 and 3 is straightforward: they are required to meet the user-given query constraints (spatial selection and value interval). Step 2, on the other hand, is needed to manage the non-uniformity of the AMR structure. A real-world AMR structure can be very complex, and so without the ability to quickly compute the uncovered predicate, meeting the AMR-specific query requirement of returning only uncovered matching cells is too expensive. These three operations, then, are key to AMR-aware query processing, and therefore form the central requirements of an AMR-aware index.

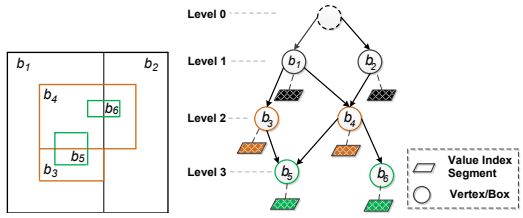
3. METHOD

We now present our proposed AMR-aware parallel, *in situ* indexing and querying methods in three parts. We first present the basic hybrid indexing structure to capture spatial and value information in the AMR hierarchy(§ Section 3.1). Then, we discuss how to extend these methods to a parallel, *in situ* context (§ Section 3.2). Finally, we discuss how to query this AMR-aware index with a workload-balanced strategy (§ Section 3.3).

3.1 A Hybrid, AMR-aware Index

To accommodate querying AMR data with both spatial and value-based constraints, we design a hybrid index data tructure. A high-level overview is given in Figure 2. Its spatial index component consists of a Directed Acyclic Graph (DAG), which organizes the AMR boxes into covering relations as some AMR frameworks, such as Chombo, lack the support of maintaining such relations. Thus, the DAG is to provide fast retrieval of all boxes within a given spatial selection. Each

vertex in the DAG represents an AMR box, and edges between vertices refer to a box-covering relationship. The juxtaposed value index component consists of many value index "segments," each covering one AMR box's data and collectively covering the whole AMR dataset. Each segment can be built from applying any existing value indexing technique, e.g., bitmap index [16], ALACRITY index [11], etc.



(a) (b)

Figure 2. Subfigure (a) shows a 3-level AMR hierarchy with boxes highlighted on each level. Subfigure (b) illustrates an AMR-aware hybrid index built from the left AMR hierarchy.

These two index components, spatial and value-based, provide the index capabilities needed for an AMR-aware index as laid out in the Problem Statement. The spatial component speeds up evaluation of spatial selections, as well as identification of uncovered cells. In contrast, the value-based component accelerates the value interval portion of queries.

Algorithm 1: AMR Index Construction

Input : A L-level AMR Ω
Output: AMR index, X

```
1  //Phase I
2  X = create_root_box(physical_domain(Ω))
3  X.add_covering_boxes(Ω_0)
4  for ℓ = {0, ..., L − 2} do
5      for every box B ∈ Ω_ℓ do
6          for f = {0, ..., |Ω_{ℓ+1}| − 1} do
7              if B.refine ∩ Ω_{ℓ+1}[f] ≠ ∅ then
8                  B.add_covering_boxes(f)
9  //Phase II
10 for every box B during the traversal of X do
11     if B is not fully covered then
12         B.vidx = build_value_index(B.data)
13 return X
```

As an initial step, we give a basic (serial, non-in situ) algorithm for building a hybrid AMR-aware index in Algorithm 1. *Phase I* (lines 2 to 8) builds the DAG by finding the box-covering relationship between boxes in adjacent AMR levels (lines 5 to 8). *Phase II* (lines 10 to 12) builds a value index for each non-fully-covered box, after which the value index segment is attached to its corresponding box (line 12).

3.2 *In Situ*, AMR-aware Indexing

We now explain how to build the hybrid index in a parallel, *in situ* context; to do so, however, it is necessary to first understand the context in which our indexing routine operates. As an AMR simulation advances its computation to each new time step, it will dynamically regrid (some of) the mesh, potentially generating new mesh levels, by tagging cells using error estimation criteria. During the regridding, a load-balancing routine is invoked to assign of AMR boxes to processes in a intelligent manner. This results in a distributed environment with a balanced workload for the simulation to compute its numerical solutions.

Our indexing is invoked at the end of a given time step, when AMR data is available for indexing. Regarding the resources for building indexes, the index generation uses the same resources available to the simulation. Particularly, every process has AMR data (e.g., boxes and data) with approximate size, and also has the *whole AMR boxes* for efficient computation purpose. This is observed in simulation infrastructures, such as Chombo [1].

We provide a high-level overview of our three-phase *in situ* indexing in Algorithm 2. In the *spatial index building* phase, a scattered DAG is generated on each process. The *value index building* phase generates value indexes for local AMR data and attaches them to their corresponding boxes. Finally, all processes write their local indexes to an aggregated index.

Algorithm 2: *In situ* AMR Indexing

Input : A L-level global AMR hierarchy, Ω
Input : Distributed AMR data on process p, Ω_p
Output: Persisted AMR index, X

```
1  all processes (in parallel) do
2      for ℓ = {0, ..., L − 2} do //I. Spatial Index Building
3          for B_{ℓ,k} ∈ Ω_{ℓ,p} do //Ω_{ℓ,p} is local boxes on level ℓ
4              g = modified_binary_search(B_{l,k}, 𝕏_{ℓ+1}) for
                i = {g.lb, ..., g.ub} do
5                  if B_{ℓ,k}.refine ∩ Ω_{ℓ+1}[i] ≠ ∅ then
6                      B_{ℓ,k}.add_covering_boxes(i)
7      for ℓ = {0, ..., L − 1} do //II. Value Index Building
8          H = [0, ...m_{ℓ,p} − 1] //mapping array
9          M_ℓ = {Ω_{ℓ,p}[0]}
10         for i = {1, ..., |Ω_{ℓ,p}| − 1} do
11             j = |M_ℓ| − 1
12             if mergeable(Ω_{ℓ,p}[i], M_ℓ[j]) then
13                 M_ℓ[j] = Ω_{ℓ,p}[i] ∪ M_ℓ[j]
14             else
15                 append Ω_{ℓ,p}[i] to M_ℓ
16             update all i in H to |M_ℓ| − 1
17         Ω_{ℓ,p} = M_ℓ
18         G_ℓ = sync_to_global_mapping(H)
19         if ℓ > 0 then
20             update_covering_boxes(M_{ℓ−1}, B_{ℓ−1}, G_ℓ)
21         build_value_index_on_each_level(M_ℓ, G_ℓ)
22     write_index_collectively(M) //III. Index Write
```

Spatial Index Building

When scaling the basic indexing method given in Algorithm 1 to an *in situ*, large-scale context, we identify the box cover finding operation Γ, the essence of which is to find covering boxes between any two consecutive levels, as a performance bottleneck. Specifically, the cost of running the two-level for

loop used to implement Γ (lines 5 to 8) in Algorithm 1 is considerably high as the number of boxes becomes large. Thus, our effort at this phase primarily focuses on optimizing this operation in an *in situ* context.

Without loss of generality, assuming we have two successive AMR level boxes on any process p: $\Omega_{\ell,p}$ and $\Omega_{\ell+1}$, where $\Omega_{\ell,p}$ is the ℓ-level data that are assigned to process p, and $\Omega_{\ell+1} = \cup_{j=0}^{m_{\ell+1}-1} B_{\ell+1,j}$, where $m_{\ell+1}$ is total number boxes on level $\ell+1$. Note that $\Omega_{\ell+1}$ is available to any process because it is observed in AMR libraries, such as Chombo.

Toward pruning the full scan of $\Omega_{\ell+1}$ to a subset scan, we take advantage of two insights. First, the small ends (bottom-left points) of all boxes at any level are commonly sorted in lexicographic order in AMR frameworks; let sequence $\mathbb{X}_\ell = \{\boldsymbol{x}_0, ..., \boldsymbol{x}_{m_\ell-1}\}$ represent the small ends in this order. Second, if two boxes are overlapping, the interval of these two boxes on each dimension (particularly, the first dimension we consider in this optimization) needs to overlap as well.

Taking the above two properties together, we expect to find a range (denoted as $g = [lb, ub]$) in $\Omega_{\ell+1}$ for any given coarse-level box $B_{\ell,k}$, such that any $\ell+1$-level box within the range could potentially overlap with $B_{\ell,k}$, and any box out of the range never overlaps with $B_{\ell,k}$. Searching for ub and lb is performed by a slightly-modified standard binary search on the sorted sequence $\mathbb{X}_\ell$. Although, the asymptotic complexity of our optimized method remains the same as that of the two for-loop operation, the practical run time is much smaller due to the pruned search space.

Building Value Index

As before, the second indexing phase concerns building the value-based portion of the hybrid index. However, this time, our parallel, in situ algorithm bears the additional optimization of performing *box merging* to improve index quality.

Back in Algorithm 1, we built one value index segment per AMR box. However, when faced with, e.g., millions of boxes (and thus millions of value index segments), such a highly "fragmented" index would produce poor performance [14].

Thus, we "de-fragment" the indexes to yield a query-optimized AMR-aware index. Our main idea is to merge the indexes belonging to contiguous boxes on the same AMR level to reduce the number of index segments. The question, however, is to determine whether boxes are mergeable or not, because merging arbitrary (non-contiguous) boxes would lead to indexing non-continuous data with holes. Thus, we define two boxes, B_1 and B_2, are mergeable *iff* the resultant box B_r, such that $B_r = B_1 \cup B_2$ and $B_r \setminus B_1 \setminus B_2 = \emptyset$. Note that, we exclude the fully covered boxes from the merging procedure to avoid unnecessary computation. Also, we merge the boxes on the same process to avoid expensive communications between processes.

The pseudocode of our scalable box merging approach for an AMR level is shown from lines 8 to 17 in Algorithm 2. A box array M_ℓ temporarily keeps the currently merged boxes during the entire merge procedure (line 9). A mapping array H is used as an array-based union-find structure to store the mapping from original boxes to merged boxes (line 8). For instance, $H[i] = j$ indicates the original box i becomes the jth box in the merged box array, M_ℓ. The search for mergeable boxes occurs only on local boxes. If there are two boxes B_j in the M_ℓ and B_i in the $\Omega_{\ell,p}$, are detected as mergeable boxes, then the jth box in the array M is updated to the merged box of B_j and B_i, meaning B_i fuses into B_j (line 13). Meanwhile, all boxes that were mapped to box i are updated to mapping to the new box j (line 16). If these two boxes are not mergeable, B_i just simply appends to the end of the M_ℓ (line 15). Similarly, all boxes that were mapped to box i now map to the new array index of box i (line 16). Note that the mergeable determination only occurs between a current box in $\Omega_{\ell,p}$ and the last box in M_ℓ (line 9) because the small ends of boxes are in sorted order.

After the box merging, we must go back and update the DAG computed in the first phase, substituting in the newly-merged boxes. To do so, we build a global mapping structure from original to merged boxes by synchronizing all the local mappings generated by each process (line 18). Then, each merged box's cover relation is re-established by converting the union of covering box sets of original boxes to a new covering box set using the global mapping (line 20).

Similarly to the Algorithm 1, we then conclude this phase by building the actual value indexes for each (now-merged) box level by level. However, this time, each process concurrently generates value indexes on only its *local* data. This independent process-by-process index generation thus does not incur any communication among processes, but requires a last phase to collect the distributed index segments into a global shared index structure.

Finally, every process first collects the metadata of the global index (the new merged boxes), which consists of the write offset and the total write size, by performing the MPI_scan and MPI_Allreduce operations. All the processes then collectively write a global index.

3.3 Scalable AMR-index-aware Querying

We now discuss using the hybrid AMR-aware index for executing queries consisting of a spatial selection θ and a value constraint $\boldsymbol{I}$, in parallel, post-processing context. A query's output is a list of cell-value tuples $(c = (l, \boldsymbol{i}), v)$, in which cells are all uncovered as stated in Section 2

Our basic querying strategy is a three-stage AMR box pruning process, applied level-by-level. First, an input "candidate" set of boxes is reduced to only those intersecting θ (the initial candidate set contains just the root box). Next, value based index segment are used to select cells matching the value constraint. After this, the DAG is consulted again to eliminate any covered cells from the result. Finally, we continue recursion to the next level, forming the new candidate set by following the covering relationship from current level boxes that intersect θ, until all levels are visited. Note that θ is refined accordingly when the query processing advances to a new level, in order to match the coordinate system.

We parallelize the basic query approach by partitioning the workload. An initial approach would be to partition θ equally

Algorithm 3: Parallel AMR-aware Query Process

Input : Query constraints: θ and $\boldsymbol{I}$
Output: A collection of tuples, T

1 **all** *processes assigned with boxes* ***(in parallel)*** **do**
2 $RR, L = load\_refine\_ratios\_and\_level\_num()$
3 $B = load\_root\_box()$
4 **for** $\ell = \{-1,\ ...,\ L-1\}$ **do**
5 $B_s = process\_spatial\_selection(B, \theta)$
6 $B_p = distribute\_boxes(B_s)$
7 **if** $\ell \neq L-1$ **then**
8 $CB_p = load\_covering\_boxes(B_p)$
9 $B = aggregate\_boxes(CB_p)$
10 **for** *every box* $b \in B_p$ **do**
11 $idxMeta = load\_index\_metadata(b)$
12 $R = value\_processing(idxMeta, \boldsymbol{I})$
13 $filter\_covered\_cells(b, b.covering\_boxes, R)$
$T = T \cup P$
14 $\theta = refine(\theta, \ell == -1\ ?\ 1 : RR[\ell])$
15 **return** Υ

across all p processes. While a reasonable approach for uniform grids, the adaptive nature of AMR would likely result in a highly-skewed workload under this strategy. For example, one process might have boxes only to a middle level, whereas another process might have boxes to the finest level. This would result in severe load imbalance.

To deal with this, we adopt a more balanced workload distribution strategy, wherein processes communicate to evenly distribute candidate box sets for each level. This strategy is realized in Algorithm 3. The *aggregate_boxes* function performs an MPI all-to-all operation to make all boxes to be processed available to every process (line 9). Then, the *distribute_boxes* function assigns t/p boxes to each process, where t is the total number of boxes at one level (line 6). This effectively balances the number of boxes on each process, which is especially in cases where some processes have very large number of covered boxes, but other processes do not. Note that the final step of Algorithm 3, querying the index segments on lines 13 and 14, is treated generically here, as this task is handled by the particular value-based indexing technique employed.

4. RESULTS

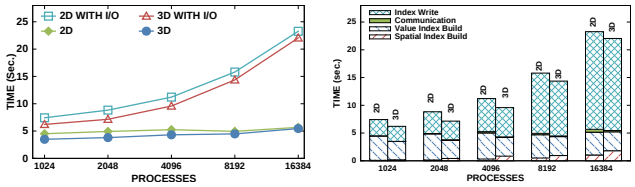
4.1 Experimental Setup

We conduct all evaluations on the "Edison" supercomputer at the National Energy Research Scientific Computing Center (NERSC). We implement our indexing and query framework based on the Chombo block-structured AMR framework [1], a popular framework used by many scientific applications. We use the HDF5 I/O library for writing our AMR index because not only is HDF5 one of widely-used I/O frameworks, but it also has a hierarchical data model which fits the AMR structure. Last, we leverage ALACRITY, a lightweight value indexing technique, as our value index component.

4.2 AMR-aware Indexing Evaluation

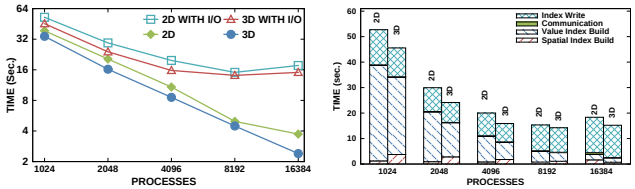
Scalability of AMR-aware Indexing

Performance of *in situ* indexing has to scale well as the indexing phase will run concurrently with distributed simulations that are already using large numbers of processors. To show the impact of our indexing, we have performed both strong and weak scalability studies. We conduct our experiments using the Chombo I/O benchmark, which provides a real *in situ* context to our indexing [6]. We configure the benchmark to perform 2D and 3D simulations, which are common in many applications. In the weak scaling experiment, each MPI process is assigned ≈40MB and ≈35MB of data in size for the 2D and 3D runs, respectively, by Chombo's load balancing function. In the strong scaling experiment, the size of total produced data is kept constant at ≈325GB and ≈285GB for entire 2D and 3D runs, respectively. Note that we repeat the value index building ten times, simulating indexing for ten data variables.



(a) (b)

Figure 3. Weak scaling performance of the AMR-aware indexing with ≈40MB and 35MB data size per-process (total 650GB and 570GB) for 2D and 3D runs, respectively.

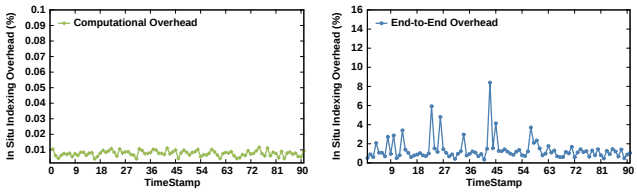


(a) (b)

Figure 4. Strong scaling performance of the AMR-aware indexing with total ≈ 325GB and 285GB data size for 2D and 3D runs, respectively.

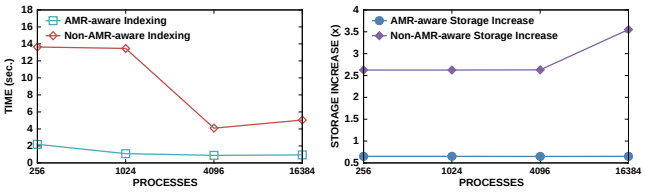
We show the execution time of indexing with weak scaling in Figure 3, increasing the number of processes from 1K to 16K using 2D and 3D configurations. Without the index write time included in the execution time, we observe a relatively constant performance. A slight increase at larger scales is due to the spatial index building time. In the box cover finding step, the narrowed search space (a subset of the fine-level boxes) grows as we double the problem size and the number of processes in this experiment. As shown in Fig. 3 (a), slightly increased communication cost during the index building phase also contributes the small increase in the execution time. Overall, the computation cost of our AMR-aware index building scales well with the increased number of processes. The size of the index varies between 60% and 110% of the original data, with compression turned on. We also show the I/O time in writing the indexes in Figure 3, which increases as the problem size increases, as expected.

We show the strong scaling performance in Figure 4. We observe that the indexing time becomes nearly half when the



(a) (b)

Figure 5. The overhead of the AMR-aware, *in situ* indexing in a BISICLES run for consecutive 90 time-steps. Subfigure (a) and (b) show percentage of indexing (CPU) and end-to-end (CPU + I/O) time added on original BISICLES time, respectively.



(a) (b)

Figure 6. Performance (subfigure (a)) and storage increase (subfigure (b)) comparison between our AMR-aware *in situ* indexing and a non-AMR-aware *in situ* indexing, *flattening*.

number of processes is doubled. As observed with the weak scaling test, the time for building the spatial index does not reduce at the same pace as that for building the value index component as shown in Fig. 4 (b). The search space on the fine-level boxes in the box cover finding remains the same with scaling the number of processes becomes twice. Despite that, we still observe a decrease in time spent building the spatial index, as the number of boxes involved in the box cover finding per process is reduced by half when the number of processes doubles. The communication cost, is almost negligible due to the total box mapping size being relatively small. From these scaling studies, we observe that our value index build component has nearly perfect scaling due to our communication-avoiding strategy.

Overhead of In Situ Indexing

To evaluate the overhead of *in situ* indexing, we use an existing Chombo-based AMR ice sheet modeling code, BISICLES [8]. We integrate our indexing algorithm directly into a trunk version of BISICLES. We conduct this experiment simulating the continental ice sheet of Antarctica, where the simulation begins with a 768x768 coarse mesh and generates AMR refined levels up to 6 levels using a refinement ratio of 2. Our indexing algorithm is invoked at every time step for the variable "dThicknessDT" (representing ice thickness change) during the simulation run, producing one index file per time step. Meanwhile, after every time step, the simulation writes a subset of its data into an HDF5 plot file for visualization and analysis purposes. We run the simulation with 1500 processes.

We show the computational and the end-to-end overheads of index generation on the simulation in Figure 5. The end-to-end overhead includes both the computational and the I/O overhead. We observe that the overheads are negligible. The computation overhead is in the range of 0.01% (shown in 5(a)), and the end-to-end overhead is typically around 1% (5(b)). The spikes in the end-to-end overhead is due to the inconsistent I/O performance. Overall, the overhead of our indexing is small, which is necessary for generating indexes in an *in situ* fashion.

Effectiveness of AMR-awareness in Indexing

We now demonstrate the effectiveness of our AMR-aware *in situ* indexing by comparing a non-AMR-aware *in situ* indexing, which needs *flattening* of the AMR structure to a uniform mesh at the finest resolution. We use ALACRITY indexing on the flattened uniform grid. The major difference between these two indexing is that our indexing operates directly on the AMR structure, as opposed to the *flattening* approach, which runs on an AMR-refined uniform mesh.

Figure 6 shows our AMR-aware indexing consistently outperforms the *flattening* method for two metrics: total indexing time, representing the *in situ* performance overhead, and storage requirement increase (index size divided by the original AMR size), indicating the run-time memory footprint requirement. Our AMR-aware indexing approach runs achieves a maximum of 12.4x speedup over non-AMR-aware approach. At 16K scale, our approach outperforms existing technique by $\approx$ 5.5x. The storage overhead with the traditional non-AMR-aware approach is obviously significantly higher. At 16K scale, the storage overhead is $\approx$ 6x more than our approach.

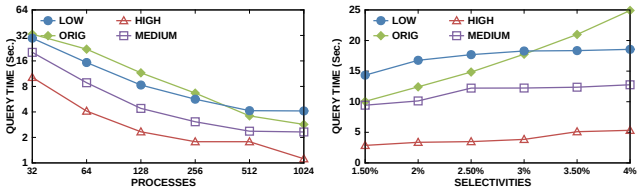
4.3 AMR-index-aware Querying Evaluation

We now evaluate the scalable AMR-aware querying based on indexes produced by experiments described in Section 4.2 All queries we execute in this section have a spatial selection and a value interval constraint. For the purpose of demonstrating scalability, we instruct our query engine to return a list of cells, rather than tuples. Time reported in this section is the maximum querying time across all processes.

Querying scalability

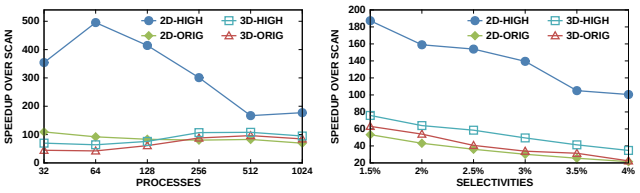
We evaluate performance of querying 2D data in two aspects: increasing process number from 32 to 1024 (in 2x increments) with a fixed querying selectivity of 1%, and increasing the selectivity with a fixed number of processes (i.e., 256 processes). Note that using up to 1024 processes as our querying demonstration is sufficient as the analysis scale is often much smaller than the data producing scale. Furthermore, to demonstrate the effectiveness of our index defragmentation, we conduct the same queries for four different levels of box merging. From the coarsest to the finest level, they are *original aggregation*, in which one value index segment associates with one original AMR box; *low, medium, and high aggregations*, in which one value index segment is in association with at most two, four, and unlimited original boxes along each dimension, respectively.

Figures 7 report the query performance over the four index aggregation levels for 2D datasets. In the fixed-selectivity experiments (subfigures 7(a)), we observe the query performance continuously improves with increasing process number. As the number of available boxes to be processed by



(a) (b)

Figure 7. Scalable query performance with a fixed query selectivity of 1% (subfigure 7 (a)) and a fixed process number of 256 (subfigure 7 (b)) at four index aggregation levels.



(a) (b)

Figure 8. Speedup of our AMR-aware querying over the scan approach for 2D and 3D AMR.

each process saturates at 512 processes, the improvement stalls, showing that the querying for this dataset scales up to 512 processes. In the fixed-concurrency experiments (subfigure 7 (b)), the query time grows as the increase of the query selectivity, which indicates the workload on each process increases as the result of the increasing querying coverage. We note that lines with low, medium, and high index aggregations become slightly flat when the selectivity increases. This is because the additional boxes retrieved by the increased selectivity are handled concurrently by processes with light workload, which leads to an unchanged overall query time. We also observe that in most cases, as the index aggregation level increases, the query speed is improving (up to 5.4x improvement compared to *original aggregation*). As we increase the index aggregation level (in other words, decrease the number of value index segments), the number of reads invoked during the query processing decreases, resulting in lower time.

Effectiveness of AMR-index-aware Querying

We now demonstrate the advantage of our AMR-aware querying method compared to a non-AMR-aware approach, referred as *scan* approach. As the name suggests, in the *scan* approach, each process reads AMR data into memory and performs a sequential scan level by level to verify the conditions of a given query. In order to filter the covered cells from the temporary results, each process performs a box cover-finding operation by using the two for-loop approach shown in Algorithm 1.

As shown in Figure 8, we observe that the new AMR-aware query approach is up to 500x faster than the *scan* approach, which justifies our effort of developing an AMR-aware index. Surprisingly, a key factor in slowing the *scan* approach is not from loading the entire AMR data as we expected. Instead, it is the box cover-finding operation, which takes most time. This demonstrates the effectiveness of our optimized box cover-finding operation.

5. RELATED WORK

Indexing is frequently a key strategy employed to speed up query processing over large datasets. Indexing can commonly be split into *spatial* and *value* indexing, in support of spatial- and value-based constraints in queries, respectively. Yet, indexing and query techniques for AMR data are virtual unexplored in the literature.

More generically (i.e., not AMR-specific), numerous value indexing techniques in scientific community have been developed. FastBit [16] is a state-of-the-art bitmap indexing library used by many scientific applications, offering WAH bitmap compression as its key feature to overcome explosive index size growth under high-cardinality scientific data. Recent work has demonstrated storage light-weight index alternatives, such as ALACRITY or hyperdyadic tree indexes [4, 11]. These techniques further have been shown to operate in a parallel and/or *in situ* context [7, 9, 12, 14, 15]. Unfortunately, none of these value indexing works are AMR-aware. Nonetheless, our proposed AMR-aware indexing and query approach leverages these existing work, seamlessly integrating any existing value indexing technique.

Spatial indexing, in contrast, deals with accelerating queries over geometry and objects in a coordinate space. The R-tree and its variants [2, 10] are among the most common spatial index structures; these function by recursively decomposing the entire spatial domain into a tree of nested bounding volumes, with spatial objects then collected into the leaf nodes. Though seemingly applicable to an AMR grid hierarchy, it turns out that the one-to-many tree-based structure of these and related spatial indexes is not a perfect match. Instead, AMR boxes exhibit a many-to-many *covering* relationship. Though methods exist in field of visualization to convert a many-to-many hierarchy to one-to-many [13], these substantially increase the number of tree nodes. However, managing the multiplicity of nodes (AMR boxes) already represents a scalability challenge. Instead, we propose the use of a more general DAG spatial structure to model this type of hierarchy.

6. CONCLUSION

In this work, we formally define, and then solve, the scalable query-driven analysis problem on block-structured AMR. Our proposed method consists of two parts, a massive parallel, *in situ* indexing and a scalable post-processing query. We minimize the performance disturbance of our *in situ* indexing by exploiting AMR boxes' lexicographical order in the AMR library infrastructure. Further, we optimize the index quality using a parallel, decentralized, user-controllable box merging approach. On the query side, we develop an AMR-index-aware querying with a workload balancing approach.

Our results show that our AMR-aware indexing and querying is up to 12.4x and 500x faster than non-AMR-aware indexing and querying. We show scalability up to 16,384 and 1,024 cores on the Edison supercomputer for our indexing and querying, respectively, demonstrating the viability of our AMR-aware analysis framework for large-scale, parallel scientific exploration. Additionally, we measure the overhead of our *in situ* indexing within the BISICLES ice sheet modeling

code, demonstrating its impact on the simulation run to be negligible.

7. ACKNOWLEDGMENT

We would also like to thank the National Energy Research Scientific Computing Center and Oak Ridge National Laboratory for the use of resources. Oak Ridge National Laboratory is managed by UTBattelle for the LLC U.S. D.O.E. under Contract DE-AC05-00OR22725. Support for this work was provided by U.S. Department of Energy, Office of Science, Advanced Scientific Computing Research and the U.S. National Science Foundation (Expeditions in Computing and EAGER programs). Work at the Lawrence Berkeley National Laboratory was supported by the Director, Office of Science, Office of Advanced Scientific Computing Research, of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231.

REFERENCES FORMAT

References must be the same font size as other body text. While this document provides several examples of how to cite common types of sources, it does not provide rules on how to cite all types of sources. Therefore, if you have a source that is not included in the examples below, SCS suggests that you find the example that is most similar to your source and use that format.

REFERENCES

1. Adams, M., Colella, P., Graves, D. T., Johnson, J., Keen, N., Ligocki, T. J., Martin, D. F., McCorquodale, P., Modiano, D., Schwartz, P., Sternberg, T., and Straalen, B. V. Chombo software package for AMR applications-design document. *Lawrence Berkeley National Laboratory Technical Report LBNL-6616E* (2000).
2. Beckmann, N., Kriegel, H., Schneider, R., and Seeger, B. *The R*-tree: An efficient and robust access method for points and rectangles*. 1990.
3. Berger, M. J., and Oliger, J. Adaptive mesh refinement for hyperbolic partial differential equations. *Journal of Computational Physics* (1984).
4. Boyuka II, D. A., Tang, H., Bansal, K., Zou, X., Klasky, S., and Samatova, N. F. The hyperdyadic index and generalized indexing and query with pique. In *Proceedings of the 27th International Conference on Scientific and Statistical Database Management* (2015).
5. Byna, S., Wehner, M. F., and Wu, K. J. Detecting atmospheric rivers in large climate datasets. In *Proceedings of the 2nd international workshop on Petascal data analytics: challenges and opportunities*, ACM (2011).
6. Carns, P., Latham, R., Ross, R., Iskra, K., Lang, S., and Riley, K. 24/7 characterization of petascale I/O workloads. In *Cluster Computing and Workshops* (2009).
7. Chou, J., Wu, K., and Prabhat. FastQuery: A parallel indexing system for scientific data.
8. Cornford, S. L., Martin, D. F., Graves, D. T., Ranken, D. F., Le Brocq, A. M., Gladstone, R. M., Payne, A. J., Ng, E. G., and Lipscomb, W. H. Adaptive mesh, finite volume modeling of marine ice sheets. *Journal of Computational Physics* (2013).
9. Dong, B., Byna, S., and Wu, K. Parallel query evaluation as a scientific data service. In *Cluster Computing (CLUSTER), 2014 IEEE International Conference on*, IEEE (2014).
10. Guttman, A. *R-trees: A dynamic index structure for spatial searching*. ACM, 1984.
11. Jenkins, J., Arkatkar, I., Lakshminarasimhan, S., Shah, N., Schendel, E. R., Ethier, S., Chang, C. S., Chen, J. H., Kolla, H., Klasky, S., and Samatova, N. F. Analytics-driven lossless data compression for rapid in-situ indexing, storing, and querying. In *Proc. Database and Expert Systems Applications (DEXA)* (2012).
12. Kim, J., Abbasi, H., Chacon, L., Docan, C., Klasky, S., Liu, Q., Podhorszki, N., Shoshani, A., and Wu, K. Parallel in situ indexing for data-intensive computing. In *Large Data Analysis and Visualization (LDAV), 2011 IEEE Symposium on* (2011).
13. Kreylos, O., Weber, G. H., Bethel, E., Shalf, J. M., Hamann, B., and Joy, K. I. Remote interactive direct volume rendering of AMR data. *Lawrence Berkeley National Laboratory* (2002).
14. Lakshminarasimhan, S., Boyuka II, D. A., Pendse, S. V., Zou, X., Jenkins, J., Vishwanath, V., Papka, M. E., and Samatova, N. F. Scalable in situ scientific data encoding for analytical query processing. In *Proceedings of the HPDC 2013* (2013).
15. Lakshminarasimhan, S., Zou, X., Boyuka Ii, D. A., Pendse, S. V., Jenkins, J., Vishwanath, V., Papka, M. E., Klasky, S., and Samatova, N. F. DIRAQ: Scalable in situ data-and resource-aware indexing for optimized query performance. *Cluster Computing* (2014).
16. Wu, K. FastBit: An efficient indexing technology for accelerating data-intensive science. *Journal of Physics: Conference Series 16* (2005).
17. Zou, X., Wu, K., Boyuka, D., Martin, D. F., Byna, S., Tang, H., Bansal, K., Ligocki, T. J., Johansen, H., and Samatova, N. F. Parallel in situ detection of connected components in adaptive mesh refinement data. In *Cluster, Cloud and Grid Computing (CCGrid)* (2015).