Local–Global Merge Tree Computation with Local Exchanges

Arnur Nigmetov
nigmetov@tugraz.at
Graz University of Technology
Graz, Austria

Dmitriy Morozov
dmorozov@lbl.gov
Lawrence Berkeley National Laboratory
Berkeley, California

ABSTRACT

A merge tree is a topological summary of a real-valued function on a graph. Merge trees can be used to find stable features in the data, report the number of connected components above any threshold, or compute other topological descriptors. A local–global merge tree provides a way of distributing a merge tree among multiple processors so that queries can be performed with minimal communication. While this makes them efficient in massively parallel setting, the only known algorithm for computing a local–global merge tree involves global reduction.

Motivated by applications in cosmological simulations, we consider a restricted version of the problem: we compute a local–global tree down to a threshold fixed by the user. We describe two algorithms for computing such a tree via only local exchanges between processors. We present a number of experiments that show the advantage of our method on different simulations.

ACM Reference Format:
Arnur Nigmetov and Dmitriy Morozov. 2019. Local–Global Merge Tree Computation with Local Exchanges. In *The International Conference for High Performance Computing, Networking, Storage, and Analysis (SC '19), November 17–22, 2019, Denver, CO, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3295500.3356188

1 INTRODUCTION

Topological methods play an important role in data analysis. By examining the connectivity of scalar fields across a range of thresholds, they identify stable features in the data and create indexing schemes that allow users to quickly answer complicated queries. This paper studies *merge trees* [7, 19]. Given a scalar field $f : X \to \mathbb{R}$, a merge tree describes the structure of connected components in all superlevel sets $f^{-1}[a, \infty)$ of the field. As one varies the threshold a from ∞ to $-\infty$, the connected components appear, grow, and merge together, thus forming a tree; see Figure 1. Such a tree implicitly labels connected components in every superlevel set and makes it possible to efficiently answer queries about the data: (1) how many connected components are there at level a? (2) what is the volume of a connected component at level b that contains a point $x \in X$? (3) how many connected components at level b exist at level a?

Merge trees are a popular tool in topological data analysis and visualization. In particular, they can be used to analyze combustion processes [5, 15], in function simplification [3]. The merge trees that appear in scientific applications are often very large [4], requiring supercomputers to compute and analyze them.

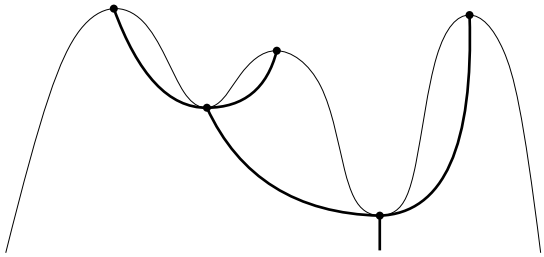

Figure 1: Scalar function $f : \mathbb{R} \to \mathbb{R}$ and its merge tree.

Our work is motivated by cosmological simulations, where the field f represents the density of the matter in the universe. In these simulations, merge trees are especially useful for the so-called *halo finding* problem: identifying dense clumps of matter and computing their properties, e.g., total mass. The halo finding problem can be formulated as identifying connected components in the superlevel set $f^{-1}[\rho, \infty)$ that have at least one point with value greater than ρ'. This information is especially easy to recover from a merge tree. Moreover, the branching structure of these trees also helps to identify subhalos, i.e., subclusters of mass, whose distribution is useful for identifying the type of object the halo represents.

Cosmological simulations [1] are some of the largest computations that run on modern supercomputers. They rely on advanced techniques, such as adaptive mesh refinement (AMR), to produce high fidelity results that adapt to the distribution of data. The individual snapshots of the simulations are dozens of terabytes (and there are thousands of time steps in a typical run). To analyze the results of these simulations, one has to develop efficient distributed algorithms that can take advantage of the massively parallel hardware. If one can identify halos in every time step of the simulation, without slowing the simulation down significantly, it is possible to feed these results back into the simulations to model physical phenomena that cannot be simulated from the first principles. Accordingly, it is necessary to compute merge trees *in situ* [12] with a running simulation, by working with its internal data representation (e.g., AMR) directly in memory, without saving data to disk.

The difficulty with computing topological information in parallel is that topology by definition captures global connectivity, while parallelization thrives on locality. In the context of merge trees, this problem has motivated a line of work on *local–global merge trees*, which we review in the next section. Although the existing algorithms can compute local–global merge trees *in situ* with the simulation, the computation involves a global reduction (an exchange of data following a butterfly network). This reduction is a

SC '19, November 17–22, 2019, Denver, CO, USA

ACM ISBN 978-1-4503-6229-0/19/11…$15.00
https://doi.org/10.1145/3295500.3356188

major bottleneck, making it impossible to compute a tree for every time step of the simulation. Furthermore, the existing algorithms only work with flat domains, i.e., they cannot process AMR data.

In this paper, we address these two problems. We introduce two algorithms for computing local–global trees that take advantage of the additional structure of the problem. In cosmology, one is only interested in density above a certain minimal threshold ρ — there cannot be dense clumps of matter at densities below it. So restricting the data to this superlevel set $f^{-1}[\rho, \infty)$ captures all the information relevant for halo finding. At the same time, this subset consists of many separate connected components, which obviates the need for a global reduction. Accordingly, our algorithms rely only on local exchanges to augment the local data with all the necessary global information. The result is an algorithm that is two orders of magnitude faster than its predecessor. This speedup, together with the ability to process AMR data, makes it possible to analyze the individual time steps at the same rate that a simulation produces them.

2 BACKGROUND AND RELATED WORK

Merge trees. Let $G = (V, E)$ be a graph, and $f\colon V \to \mathbb{R}$ a real-valued function on its vertices $V = \{v_1, \dots, v_n\}$. We assume, without loss of generality, that the vertices are ordered by the function value:

$$f(v_1) < f(v_2) < \cdots < f(v_n).$$

For $a \in \mathbb{R}$, we denote by G_a the subgraph induced by the vertices with function value at least a. We call this subgraph the *superlevel graph at a*. As a varies from $f(v_n)$ to $f(v_1)$, the connected components of the superlevel graph G_a appear and merge together. The merge tree of f records how this happens; see Figure 2.

By definition, the *merge tree* of f is the graph $T_f = (V, E_T)$ that has an edge $(v_i, v_j) \in E_T$, with $i < j$, if and only if v_i and v_j belong to the same connected component C of $G_{f(v_i)}$, and there is no vertex $v_k \in C$ such that $f(v_i) < f(v_k) < f(v_j)$.

Intuitively, each vertex v_i in the merge tree T_f represents the connected component of v_i in the superlevel graph $G_{f(v_i)}$. The edges represent the nesting relationship between the connected components. We note that the merge tree T_f is not a subgraph of G. If G is disconnected, then T_f is not a tree but a forest. Even though in this paper we are interested exactly in the case when T_f consists of many relatively small connected components, we simplify the language and call it a merge tree.

Triplet merge trees. Throughout the paper we rely on the *triplet merge tree* representation, introduced by Smirnov and Morozov [20]. This representation transforms the merge tree into a directed tree of its subtrees. Specifically, a directed edge (v_i, v_j) with label v_k belongs to the triplet merge tree if and only if vertex v_i has the highest function value of all the vertices in its connected component of G_a for $a \in [f(v_i), f(v_k))$, and v_i falls into the same connected component as v_j in $G_{f(v_k)}$. See Figure 2 on the right for an example of a triplet merge tree.

The authors of [20] show how to compute the triplet merge tree directly, bypassing the computation of the ordinary merge tree first. They show that doing so is faster in practice than computing the ordinary merge tree. Furthermore, the computation can be run in parallel in shared memory by merging edges independently, in a lock-free manner.

The authors introduce a procedure $\textsc{Merge}(T_1, T_2, E)$ that we rely on in our paper. Given a function $f : V \to \mathbb{R}$ on the vertices of a graph G, let f_1 and f_2 be its restrictions to two disjoint subsets V_1 and V_2, with $V_1 \sqcup V_2 = V$. If T_1 and T_2 are the triplet merge trees of f_1 and f_2, and E is the set of edges of G connecting the two subsets, then $\textsc{Merge}(T_1, T_2, E)$ returns the triplet merge tree of f.

Grids. Since merge trees are often computed for data on d-dimensional grids, it is necessary to convert a grid to a graph, to fit with the definitions given so far. There are multiple ways to do this. Throughout the paper we use the Freudenthal triangulation [11]: two grid cells with indices $x = (x_1, \dots, x_d)$ and $y = (y_1, \dots, y_d)$ are connected with an edge if and only if either all differences $(x_i - y_i)$ are in $\{0, 1\}$ or all of them are in $\{-1, 0\}$. See Figure 3.

Adaptive Mesh Refinement. Modern simulations rely on advanced techniques to accurately and efficiently represent the simulated phenomena. One such technique, adaptive mesh refinement (AMR), uses different resolution in different regions of the domain, to efficiently allocate computational resources.

An AMR mesh consists of a set of grids, each on its own level l_i, with higher levels corresponding to finer grids. For each grid, we know its refinement r_i, relative to the base level. A single cell on level $i-1$ is covered by $(r_i/r_{i-1})^d$ cells on level i. For each grid, we also know the bounds of the region that it covers in the domain. Cells of a grid on level i that are covered by a cell of a grid on level $i+1$ are called *masked*, and their values must be ignored. A cell cannot be partially masked, and regions of the grids on the same level are disjoint (but two grids on the same level can be adjacent).

Once again we must convert an AMR mesh into a graph. Given such a mesh, let vertices of G be all the cells that are not masked. We define the *inner* edges of G by fixing the graph structure inside a single grid using the Freudenthal link. What remains is to define the *outer* edges that connect different grids. If two adjacent grids are on the same level, then we use the same Freudenthal link to connect their boundary cells.

If one grid is on a higher level, then we expand it by one layer of *ghost cells*, and extend the graph to include them. Every ghost cell is contained in the unique cell of the grid at one level lower. Replacing the ghost cell by the cell that contains it, we replace the edges connecting boundary cells and ghost cells with the edges connecting the boundary cells of the finer grid with the cells of the coarser grid. Multiple boundary–ghost edges can map to the same fine–coarse edge. See Figure 4 for an illustration.

Parallel setup. We are interested in the setting, where the domain of the function $f : V \to \mathbb{R}$ is distributed among different processors. Specifically, we assume there is a global graph $G = (V, E)$ with a function $f : V \to \mathbb{R}$ and a partition of its vertices $V = V_1 \sqcup \cdots \sqcup V_b$. Every V_i is kept in a block i, which also stores $f\big|_{V_i}$ and all the edges that contain at least one vertex in V_i, $E_i = \{(u, v) \mid u \in V_i \text{ or } v \in V_i\}$. In the context of AMR meshes, the partition of the vertices corresponds to the individual grids that constitute the AMR mesh: each V_i is the set of vertices that belong to a single grid.

Our terminology of blocks comes from our use of the DIY library [16] — a data-parallel library built on top of MPI — where

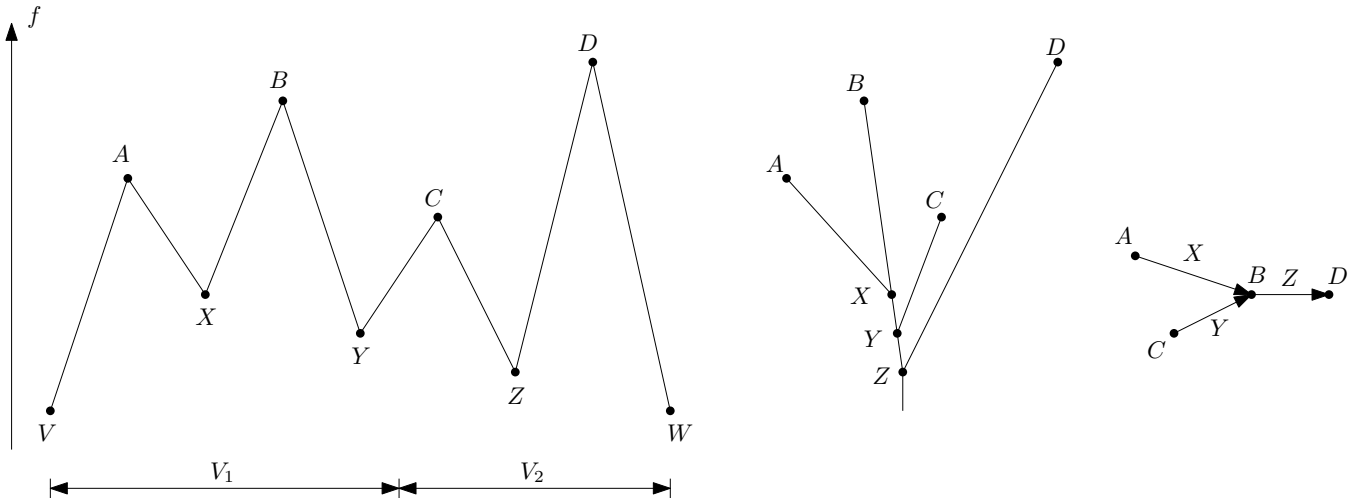

Figure 2: Function f on a graph (left), its merge tree (center), and its triplet merge tree (right).

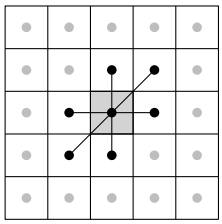

Figure 3: Freudenthal link of a vertex in a two-dimensional grid.

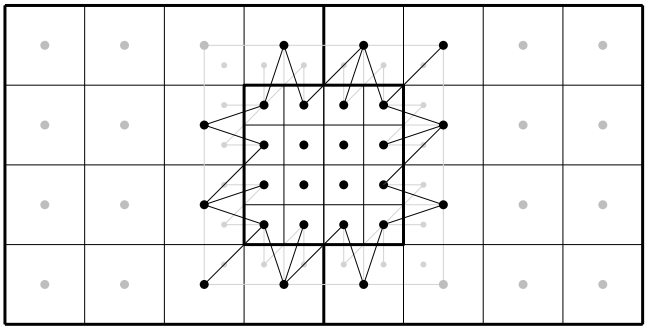

Figure 4: An AMR mesh with three grids: two at level 1, and one at level 2, with refinement $r_2 = 2$. The outer edges of the finer grid are shown. They are constructed by adding a layer of ghost vertices (in light gray) to the finer grid, extending the graph to include them, and remapping the ghost vertices to the coarser grid.

the main unit of data is called a *block*. In the simplest situation, one block is assigned to one processor, the configuration we use in this paper.

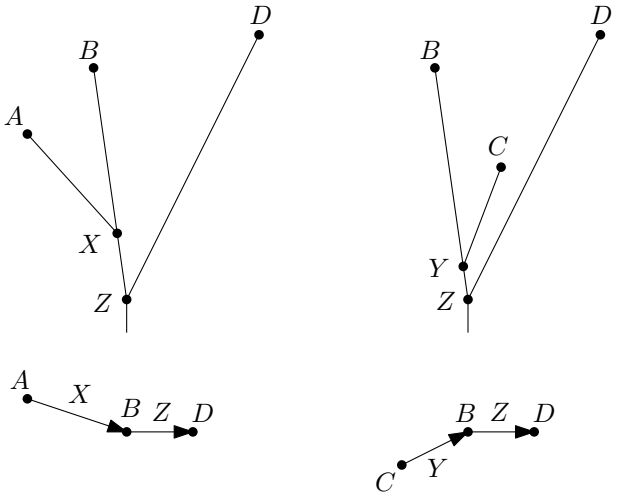

Figure 5: Local–global trees for the blocks V_1, on the left, and V_2, on the right, in Figure 2.

We define a graph H whose vertices are blocks $B = \{1, \ldots, b\}$. An edge connects two blocks i and j if and only if there is an edge between some pair of vertices $u \in V_i$ and $v \in V_j$ in G.

Our algorithms rely on local communication between blocks. Every block i maintains the list of neighbors N^i, with which it is able to communicate. During the execution of our algorithms, the neighborhood of a block may change: block i maintains its neighborhood N^i, by adding or removing blocks.

Local–global merge trees. Morozov and Weber [17] introduced the *local–global merge tree* representation, in which every block stores detailed information about its local vertices, together with just enough global information to answer global queries. Significantly, no processor has to store the full tree, and the queries can be answered with minimal communication. The local–global representation is especially well-suited for *in situ* parallel analysis [10, 17].

The local–global tree is defined with respect to a subset of vertices of the full domain. It is simpler to define in the triplet representation. Suppose the full triplet merge tree is denoted T_f. Given a subset $W \subseteq V$, the (triplet) local–global tree with respect to W, T_f^W consists of the union of all the paths in T_f from the vertices of W to the root. T_f^W may contain vertices outside of W, if they lie on some path from a vertex in W to the root. These extra vertices provide the eponymous global information. Below, we call the operation of keeping only the paths from the local vertices to the root, i.e., going from T_f to T_f^W, *sparsification*.

Given a function $f : V \to \mathbb{R}$, where V are the vertices of the global graph G, and a partition of its vertices $V_1 \sqcup \dots V_b$, the goal is for each block to compute $T_f^{V_i}$. See Figure 5 for an illustration of the local–global trees for the function in Figure 2 partitioned into two blocks. Although not apparent from this tiny example, an important feature of the local–global trees is that they are significantly smaller than the full global tree; in fact, they are only a little larger than the strictly local trees, while being substantially more versatile [17].

The crucial advantage of the local–global tree is that once computed, each block can determine completely locally what connected component of the full merge tree T_f any one of its local vertices belongs to *on any level*. This basic question is central to a large number of queries: assigning unique labels to connected components at a given level, computing volumes of connected components, etc. Once the local–global representation is computed, these queries can be answered locally without any communication between the blocks, except to report the results.

Morozov and Weber [17] give an algorithm to compute a local–global tree via a global reduction between the processors. Although the resulting representation is efficient for querying, the global reduction itself can be expensive. In this paper, we consider a restricted version of the problem that motivates replacing the global reduction with a local exchange, greatly improving the performance.

Restricted problem. Given a threshold $\rho \in \mathbb{R}$, we are interested in the local–global merge tree of G_ρ, the superlevel graph. In many applications, all the interesting content of a given function lies above such a threshold, and as we show below, such a restriction significantly improves computation.

We use the following terminology throughout the paper. Given a threshold $\rho \in \mathbb{R}$, we refer to the vertices v with $f(v) < \rho$ as *low* and those with with $f(v) \geq \rho$ as *high*. By definition, G_ρ is the graph obtained from G by removing all low vertices. Our goal is to compute the local–global merge tree of f on G_ρ in the sense of [17].

We note that because the block graph H is constructed independently of the threshold ρ, it depends solely on G: it can happen that there is no edge in G_ρ between blocks i and j, but they are connected in H.

2.1 Related Work

The traditional way to compute merge trees in serial relies on a variation of the classic Kruskal's algorithm [9]. We note that merge trees are different from contour trees, which track level sets, rather than superlevel sets. Carr et al. describe an algorithm to compute a contour tree from two merge trees [7].

Considerable work has been done on computing merge trees in shared memory [6, 8, 13, 20], and it remains an active research direction. These results are interesting, but ultimately independent of the distributed setting we are interested in. All of those algorithms can be combined with the techniques described in this paper.

Pascucci and Cole-McLaughlin [19] describe an algorithm suitable for using in distributed memory. It performs a global reduction, merging trees from different processors in pairs. This approach has two ultimate problems: the reduction doesn't scale past a small number of processors, and the final global tree is difficult to work with, since for large data sets it does not fit in the memory of a single node.

Two independent approaches address this limitation. Morozov and Weber [17] introduce local–global trees, described in the previous subsection, that perform a swap reduction (rather than the merge reduction of [19]). Most importantly, instead of computing one large global tree, they compute small local–global trees, one per processor, which are convenient for subsequent analysis, including for deriving local–global contour trees [18].

The second approach relies on the restricted version of the problem, described in the previous subsection. Landge et al. [14] describe an algorithm to compute merge trees in a way similar to the neighborhood-growing Simple algorithm we describe in Section 3.2. The principal difference is that their algorithm performs a fixed number of rounds, prescribed a priori as input, and as such only guarantees that features below a certain size are computed correctly. In contrast, our algorithm adapts the number of rounds to the problem, increasing them as needed, until the correctness of the output (the solution to the restricted problem) is guaranteed. Additionally, besides computing local–global trees and processing AMR data, we also introduce the Components algorithm, in Section 3.3, which our experiments in Section 4 show performs better than the Simple algorithm.

3 ALGORITHMS

We describe two algorithms, Simple and Components. Both algorithms are implemented using DIY library [16] and follow the bulk synchronous parallel (BSP) model [21]. The main unit of data is a block, which knows its neighbor blocks and can communicate with them. The algorithms work in multiple rounds, expanding the neighborhoods with which they communicate, until they see enough of the domain to be certain that they have constructed the complete local–global trees. The difference between the algorithms is in how much data they send to other blocks.

The first algorithm, Simple, sends its full original tree to each block in its neighborhood. The second algorithm, Components, splits the local tree into connected components and grows each such subtree independently, sending it only to those blocks that have trees that connect to it.

3.1 Computing outer edges

Since we are working with a subset of the function above the given threshold ρ, the first step in both algorithms is to find active vertices that belong to the subgraph G_ρ. Each block must also identify its

outer edges that connect its vertices to other blocks. Because each block has only access to its local data, some extra care is required to identify the outer edges. An edge (u, v) in the full graph G may connect vertex u active in the local block (i.e., $f(u) \geq \rho$) to a remote inactive vertex v (with $f(v) < \rho$). To prune such edges, Algorithm 1 exchanges the outer edges between the neighboring blocks and takes their intersection. This exchange ensures that the remaining edges have both vertices above the threshold ρ.

Algorithm 1 ComputeOuterEdges

```
function ComputeOuterEdges(ρ)
    for all block i do
        ▷ use ρ to identify high vertices.
        V^i ← {unmasked high vertices of block i}
        E^i ← {outer edges from block i}
        for all j ∈ N^i do
            E^i_j ← {edges of E^i that end in block j}
            Send E^i_j to j
    Exchange
    for all block i do
        E^i_R ← ∅                          ▷ set of edges received by block i
        for all j ∈ N^i do
            Receive E^j_i from j
            E^i_R ← E^i_R ∪ E^j_i
        E^i ← E^i ∩ E^i_R
```

In order to compute the outer edges locally, we use the AMR mesh to determine the geometry of the neighboring grids. Each block has the list of all of its neighboring grids; this is the graph H described in the previous section. Any outer edge, by definition, connects a local vertex to a vertex of one of these neighbors. We assume the geometry of the neighboring grids — their extents, level, and refinement — is known to the block. This allows each block to compute all of its outer edges in the full graph G. In Algorithm 1, E^i denotes the set of all outer edges for a block i.

3.2 Algorithm Simple

Algorithm 2 Simple.

```
ComputeOuterEdges(ρ)                    ▷ Algorithm 1
for all block i do
    InitSimple(i)                       ▷ Algorithm 3
while not all blocks are done do
    for all block i do
        SendSimple(i)                   ▷ Algorithm 4
    Exchange
    for all block i do
        ReceiveSimple(i)                ▷ Algorithm 5
```

The initialization (Algorithm 3) in the Simple algorithm (Algorithm 2) includes computing the local tree T^i on active vertices V_i. We sparsify the tree with respect to the boundary vertices of the block i. We save the original tree and the original neighbor blocks in variables T^i_o and N^i_o. We use them later in the algorithm, after the tree T^i and the neighborhood N^i grow as a result of the computation. Recall that N^i denotes the set of blocks with which block i can communicate; it is initialized with the neighbors of block i in the graph H.

Algorithm 3 Initialization in algorithm Simple.

```
function InitSimple(i)
    T^i ← local tree on block i
    Sparsify(T^i)
    N^i ← {j : (i, j) ∈ H}
    T^i_o ← T^i                         ▷ T^i_o will never change
    N^i_o ← N^i                         ▷ N^i_o will never change
```

After initialization we start the communication process. In each round, every block sends (Algorithm 4) its original local tree T^i_o, its outer edges E^i, and its original neighborhood N^i_o to all blocks in its current neighborhood N^i that have not received this information in previous rounds.

Algorithm 4 Sending phase of algorithm Simple.

```
1: function SendSimple(i)
2:     for all r ∈ N^i do
3:         if r not marked as processed then
4:             Send (T^i_o, E_i, N^i_o) to r
5:             Mark r as processed
```

In the receiving phase (Algorithm 5), every block receives local trees of its neighbors T^j_o and merges them into its current tree T^i. The edges E^j are either contained in the current neighborhood N^i, or they connect a vertex from N^i with a vertex of a block outside of N^i. The former edges are used in the merge process; the latter edges are accumulated in the variable $\tilde{E}$, which stores the outer edges of the expanded neighborhood.

Consider an edge (u, v) from $\tilde{E}$. Its vertex u is contained in the tree T^i after we merged the corresponding tree T^j_o into it. Its vertex v belongs to a block with which the current block i has not communicated yet. If there is a path in graph G_ρ connecting vertex u to the original local tree T^i_o, then block i is not done: our algorithm must continue to the next round. If there is no such path to any edge in $\tilde{E}$, then we have the complete local–global tree for the block i.

Figure 6 illustrates an example. In the first round, two of the central block's local connected components leave the neighborhood. The block sends its local tree to the eight neighboring blocks. In the second round, because the connected component in the top part of the central figure leaves the neighborhood, the block is not done. It sends its tree to the next layer of neighboring blocks in H. After the third round, all the connected components of the central block are contained entirely in the known neighborhood N^i, and the central block declares itself done.

3.3 Algorithm Components

The first algorithm, Simple, sends the entire local tree to all of its neighbors; each block expands its neighborhood using only the

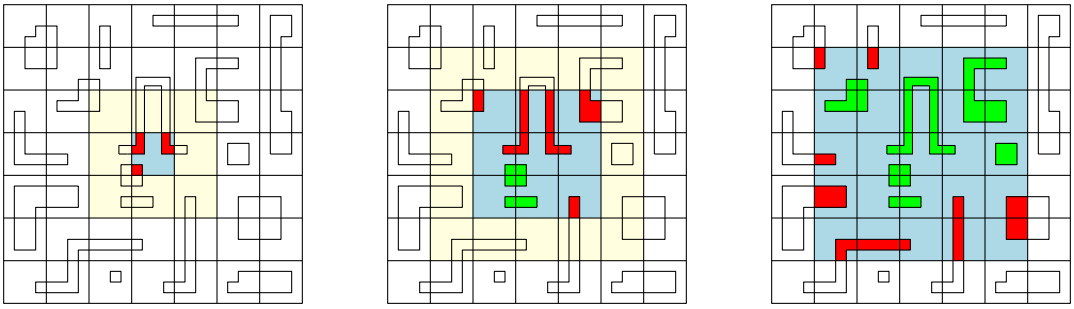

Figure 6: Three rounds of the Simple algorithm from the point of view of the central block. Its communication neighborhood is highlighted in yellow. The neighborhood whose trees are known to it is highlighted in blue. Components in green are contained entirely in the blue neighborhood. Components in red leave it. Once every local component is green, the block is complete.

Algorithm 5 Receiving phase of algorithm Simple.

```
1:  function ReceiveSimple(i)
2:      Ẽ ← ∅
3:      Ñ^i ← ∅
4:      for all (T_o^j, E_j, N_o^j) in incoming do
5:          T^i ← merge(T^i, T_o^j, E^j)
6:          Ẽ ← Ẽ ∪ {edges of E_j that do not end in a block of N^i}
7:          Ñ^i ← Ñ^i ∪ N_o^j
8:      done_i ← True
9:      for all (u, v) ∈ Ẽ do
10:         ▷ outgoing edge connected to the original tree
11:         if u or v is connected in T^i to T_o^i then
12:             done_i ← False
13:             Break
14:     N^i ← N^i ∪ Ñ^i                    ▷ Expand the neighborhood
```

structure of the graph H. The second algorithm, Components (Algorithm 6), is more selective. It handles each connected component of the local tree separately and sends the corresponding subtree only to the blocks with a tree connected to it.

Algorithm 6 Components.

```
ComputeOuterEdges(ρ)                          ▷ Algorithm 1
for all block i do
    InitComponents(i)                         ▷ Algorithm 7
while Not all blocks are done do
    for all block i do
        SendComponents(i)                     ▷ Algorithm 8
    Exchange
    for all block i do
        ReceiveComponents(i)                  ▷ Algorithm 9
```

In the initialization function (Algorithm 7), each block i computes its local tree T^i and breaks it up into connected components, $T^i_1, \dots, T^i_{n_i}$. The set of outer edges E^i is split into $E^i_1, \dots, E^i_{n_i}$, where edges E^i_j emanate from the vertices of the tree T^i_j.

Each component also maintains three sets A^i_k, B^i_k and C^i_k:

- Set A^i_k stores the blocks, to which the component T^i_k must be sent. These are the blocks that we know contain a tree connected to T^i_k. We initialize A^i_k using the set of edges E^i_k.
- Set B^i_k stores the blocks, to which we have already sent the tree T^i_k.
- Set C^i_k stores the component trees T^a_b that are connected to the tree T^i_k in the global tree. We identify each component tree by its highest vertex, v^a_b. We initialize C^i_k using the outer edges E^i_k, after modifying Algorithm 1 to send with each outer edge the identifier of the component, to which its local vertex belongs.

Algorithm 7 spells out the initialization details.

Algorithm 7 Initialization in algorithm Components.

```
function InitComponents(i, ρ)
    T^i ← local tree in block i
    T^i_1, ..., T^i_{n_i} ← connected components of T^i
    for k = 1 ... n_i do
        E^i_k ← outgoing edges from T^i_k to neighboring blocks
        A^i_k ← { i } ∪ { a | ∃e ∈ E^i_k that ends in block a }
        B^i_k ← { i }
        C^i_k ← { v^i_k } ∪ { v^a_b | ∃e ∈ E^i_k that ends at T^a_b }
```

In the sending phase (Algorithm 8), a block iterates over its connected components T^i_k. If there are blocks with trees connected to T^i_k, but to which we haven't yet sent this tree — in other words, if $A^i_k - B^i_k$ is not empty — the block sends the component T^i_k with its edges to each of these blocks. Afterwards, all blocks in A^i_k are added to B^i_k to mark them as processed. Hence, each component is sent at most once to any block. If the receiving phase of the previous round added new components to the set C^i_k, then we send the set C^i_k to all blocks A^i_k, connected to the tree T^i_k, since they are connected to them as well.

The bulk of the work happens in the receiving phase (Algorithm 9). First, block i collects all the incoming trees T^a_b and merges them into its growing tree T^i. Then, we need to update the sets of

Algorithm 8 Sending phase in algorithm Components.

```
1: function SendComponents(i)
2:     for k = 1 ... n_i do
3:         for all blocks r in A^i_k − B^i_k do
4:             Send (T^i_k, E^i_k) to r
5:         B^i_k ← A^i_k                ▷ mark blocks in A^i_k as processed
6:         if C^i_k changed in the previous round then
7:             for all blocks r in A^i_k do
8:                 Send C^i_k to r
```

connected trees C_k^i, which we do by identifying which incoming trees T_b^a connect to which component T_k^i of the block's tree. The second for-loop takes care of this update.

Algorithm 9 Receiving phase in algorithm Components.

```
1: function ReceiveComponents(i)
2:     for all incoming (T^a_b, E^a_b) do
3:         T^i ← merge(T^i, T^a_b, E^a_b)
4:     for k = 1 ... n_i do
5:         Ĉ^i_k ← C^i_k
6:         for all incoming C^a_b do
7:             if T^a_b and T^i_k are connected in T^i then
8:                 C^i_k ← C^i_k ∪ C^a_b
9:     for k = 1 ... n_i do
10:        C^i_k ← ⋃ { C^i_m | m = k or T^i_k, T^i_m are connected in T^i }
11:        A^i_k ← { a | ∃ v^a_b ∈ C^i_k }
12:    N^i ← ⋃_{j=1}^{n^i} A^i_j
13:    done_i ← (∀k = 1 ... n_i : C^i_k = Ĉ^i_k)
```

After the set of connected trees C_k^i is computed, we put into the set A_k^i the identifiers of all blocks a such that at least one of their component trees T_b^a is in C_k^i. Finally, in the last line of Algorithm 9, we determine if the block is finished by whether any set of connected components C_k^i has changed. We take a logical-and of all such results via a global reduction (`MPI_Allreduce`): as long as a single block is not finished, the algorithm continues.

Optimization. In order to update the sets of connected trees C_k^i more effectively, we use the connectivity information provided by the merge tree T^i. If several components T_b^a get connected after calling the merge procedure, then their union has a unique deepest vertex — it is the vertex $\hat{v}_b^a$ that has the maximum function value among all vertices v_b^a. The converse also holds: If components T_b^a and T_d^c have the same deepest vertex in T^i, then they are connected.

We take advantage of these observations and implement the connectivity tests in Lines 7 and 10 of Algorithm 9 as follows. We create a hash table K that maps the vertices v_b^a identifying different components of T^i to sets of block identifiers. We iterate over all received components T_b^a and compute the deepest vertex v of v_b^a in the current tree T^i. Then we add all the component identifiers from C_b^a to the set $K[v]$ in the hash table.

After running a similar loop over all local components T_j^i, we have the hash table with the correct values of sets C_j^i. All that remains is to iterate once again over all local components T_j^i, find their deepest vertex v in T^i and set the C_j^i to the set $K[v]$. This optimization is effective because finding the deepest vertex in a triplet merge tree is straightforward and computationally efficient.

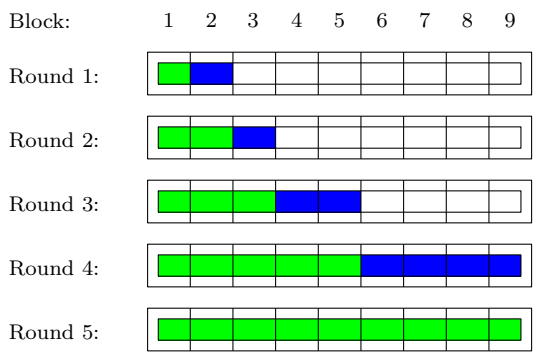

Figure 7: The Components algorithm's execution on nine blocks with a single component that spans all of them. Colors indicate the status of the component from the point of view of the left-most block: green means the component has been merged into the block's tree; blue means the block knows about this component, but has not received it yet.

Illustration. Figure 7 highlights a key feature of the Components algorithm: the components grow exponentially (thus the algorithm finishes in logarithmic number of rounds, in the size of the largest component). The example in the figure illustrates a linear chain of blocks, with a single component across all of them. The components that have been merged into the tree of the left-most block are shown in green; their blocks are in the set B_1^1. The components known to the left-most block, but with which it has not communicated yet, are shown in blue. The green and blue components make up the set C_1^1. The block of the blue components are in the difference $A_1^1 - B_1^1$.

Because during each round a block sends all the components known to it, that are connected to a particular component, the sets C_j^i grow exponentially. For example, after round 3, block 5 knows about all nine connected components. After it communicates with block 1, between rounds 3 and 4, block 1 learns about all nine components as well. It communicates with them between rounds 4 and 5, and thus finishes its local tree in round 5.

Figure 8 illustrates the execution of the Components algorithm on the same example as Figure 6. Its salient point is that, in contrast to the Simple algorithm, only components connected to some local component are communicated to the block. This translates in substantial reduction in communication, computation, and memory usage that are responsible for the better experimental performance in Section 4.

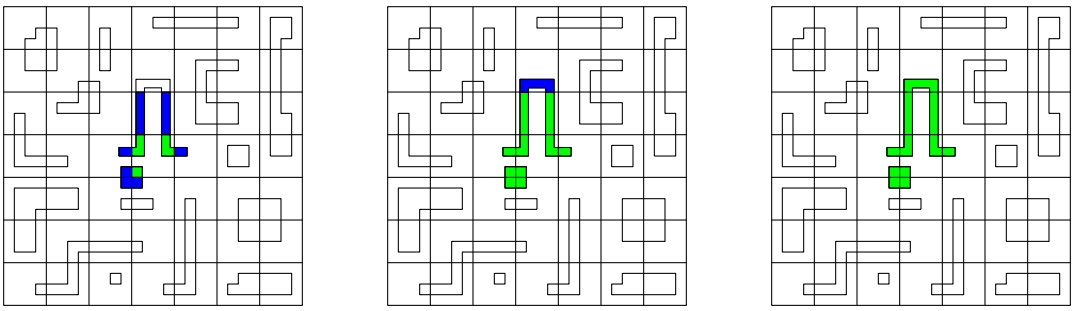

Figure 8: Execution of the Components algorithms from the point of view of the central block, on the same input as Figure 6. The color scheme is the same as in Figure 7: green components have been merged into the block's tree; blue components are known to the block, but have not been received yet.

3.4 Correctness

We sketch why our algorithms correctly compute the local–global trees, forgoing the formal proofs for lack of space.

Observation 1. If the domain of the function f is disconnected, then the merge tree of f is the disjoint union of the merge trees of f restricted to each connected component. The component merge trees can be computed independently from each other.

Observation 2. Suppose that the domain of the function f is connected and distributed among blocks $0, 1, \dots, n$. If blocks $1, \dots, n$ send their merge trees T^i and all outer edges E^i to block 0, then, after merging the received trees into T^0 in *arbitrary order*, T^0 contains the correct global merge tree of f.

Observation 1 is immediate from the definition of the merge tree. Observation 2 follows from the results of [20]. We can think of outer edges as the glue that connects different pieces of the global tree together: if we have all components of the tree and glue each outer edge, we get the complete merge tree.

In both algorithms, blocks always send merge trees with all outer edges that belong to them, and in the receiving phase they merge every tree they received into their local tree T^i. Observations 1 and 2 imply that correctness is a consequence of the following statement.

Statement 1. If at the end of round t there exist blocks i and j such that there is a path in G_ρ connecting $v \in V^i$ and $u \in V^j$, and block i did not send the tree that contains v to block j, then the algorithm does not terminate at round t.

Simple algorithm. Statement 1 is immediate for this algorithm. Formal proofs follow by induction on the number of rounds.

Lemma 1. At the beginning of communication round t:

$$N^i = \{\, j \mid d_H(i,j) \le t \,\},$$

where d_H denotes the graph distance in the graph H of blocks. Block i sends its tree to block j at round t if and only if $d_H(i,j) = t$.

To prove Statement 1, suppose there is a block j that needs the original tree of block i, but it has not received it. This means that there is a vertex x in V^i and vertex $y \in V^j$ that are connected by a path in G_ρ. If block i has not sent its tree to block j, then there is an edge $e = (u, v)$ on this path that goes from one of the processed blocks of N^i to a block to which the original tree T^i_o has not been sent. This edge forces the termination condition to be false in the receiving phase.

The fact that Simple algorithm terminates at some point also follows from Lemma 1: in the worst case (when the domain is connected and covers all blocks), the neighborhood of each block contains all blocks after at most diam(H) rounds.

Components algorithm. We define $\mathcal{H}$ to be the graph whose vertices are all local component trees,

$$V(\mathcal{H}) = \left\{\, T^i_j \mid i = 1, \dots, n, j = 1, \dots, n_i \,\right\};$$

T^a_b and T^c_d are connected in $\mathcal{H}$ if and only if there is an edge in G_ρ between some vertices of T^a_b and T^c_d. By $d_{\mathcal{H}}(T^a_b, T^c_d)$ we denote the graph metric on $\mathcal{H}$, i.e., the length of the shortest path between T^a_b and T^c_d. The following lemma summarizes some basic properties of the Components algorithm.

Lemma 2.

(1) Sets A^i_k, B^i_k and C^i_k only grow.
(2) If block i receives (C^a_b, v^a_b), then it has already received the corresponding component tree T^a_b. So its deepest vertex v^a_b is present in T^i.
(3) In all rounds, C^i_k is a connected subset of the tree graph $\mathcal{H}$, A^i_k and B^i_k are connected subsets of the block graph H.
(4) After t rounds of communication, every component tree T^a_b such that $d_{\mathcal{H}}(T^i_j, T^a_b) \le 2^t$ is contained in C^i_j.

The last statement of the lemma, Item 4, is the only one not immediate from the algorithm description, but it can be proven by induction. Figure 7 gives a hint why it is true.

The information flows in the Components algorithm as follows: each tree T^i_k has its own set C^i_k that grows exponentially. This set is always a connected subset of the component of T^i_k in $\mathcal{H}$. Each time a new element is added to C^i_k, it is reported to all blocks that contain at least one tree in C^i_k. The information spreads along paths in graph $\mathcal{H}$, each time to a larger distance.

Proof of Statement 1. Suppose that component T^i_k declares itself done at the end of communication round t, and there are blocks that need T^i_k, but have not received it yet. This means that there exists a local component tree T^k_m such that (a) block k, to which T^k_m

belongs, has not received T_j^i, so $k \notin B_j^i$; (b) T_m^k and T_j^i belong to the same connected component of the global merge tree.

The last condition implies that there is a path in graph $\mathcal{H}$ that connects T_j^i and T_m^k. Let us denote this path as

$$T_j^i \to T_{b_1}^{a_1} \to \cdots \to T_{b_r}^{a_r} \to T_m^k.$$

We can assume, without loss of generality, that all blocks $a_1, \ldots, a_r$ have already received T_j^i. Because there is an edge between $T_{b_r}^{a_r}$ and T_m^k in graph $\mathcal{H}$, there is an outer edge between them in G_ρ. Therefore, v_m^k is added to $C_{b_r}^{a_r}$ during initialization.

Since we assumed tree T_j^i was not sent to block k, it follows tree $T_{b_r}^{a_r}$ was not sent to tree T_j^i: otherwise, by the end of round t, the tree T_j^i would be aware of T_m^k, and then it cannot declare itself done before sending itself to block k. This implies that component $T_{b_r}^{a_r}$ is not done at round t, since it is now aware of T_j^i. So the algorithm continues. The Components algorithm eventually terminates because in each iteration at least one C_k^i grows. □

4 EXPERIMENTS

Experiments were performed on two machines of the National Energy Research Scientific Computing Center (NERSC), *Edison* and *Cori*. *Edison* is a Cray XC30 supercomputer with 5,586 nodes, each node with 24 CPU cores and 64 GB of RAM. Each node has two sockets with an Intel Ivy Bridge 12-core processor. *Cori* is a Cray XC40 supercomputer with two types of nodes, Haswell and KNL. We used both types of nodes. There are 9,688 KNL nodes, each with a single-socket 68-core Intel Xeon Phi 7250 processor, and 2,388 Haswell nodes, each with two sockets, each with a 16-core Intel Xeon E5-2698 v3 processor. The code was compiled with GCC compiler (version 7.3.0).

Cosmology. We used data sets from cosmological simulations of the Nyx project [1]. The function f is the (normalized) density of the matter distribution for different values of cosmological redshift z given on a 3-dimensional grid. We ran experiments for $z = 2$, comparing the running times of the two algorithms. We do not include the IO time, because in the experiments the data was read from disk and written back to it, while in real application the code is going to be used *in situ*, with data residing in the memory. So we show only the essential computation, including local initialization and communication phase. Our experiments used data sets of sizes 512^3, 1024^3, 2048^3, and 8192^3.

In Figure 9, we compare Simple and Components algorithms on a single 2048^3 dataset, running on *Cori* Haswell nodes. The figure illustrates the average of five runs, for each data point, with the standard deviation shown as error bars. As the figures illustrate, Components algorithm runs approximately 1.5 times faster than Simple algorithm. In Components algorithm, blocks send only those component trees that are needed. Even though maintaining the A, B and C sets introduces overhead, it is minimal compared to the reduced communication and computation. We conclude that for such single-grid data Components algorithm is always preferable. The only advantage of Simple is its simplicity, which allows for fast reference implementation.

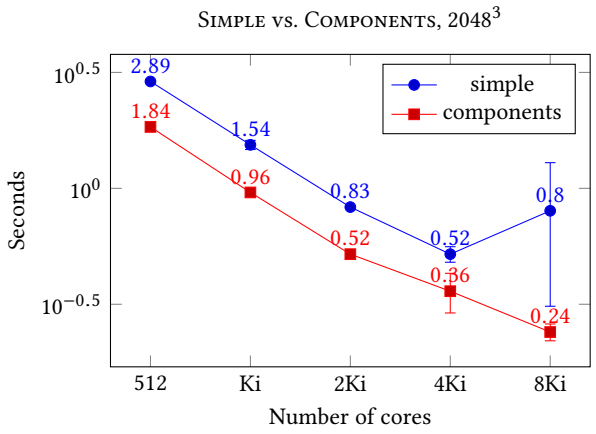

Figure 9: Strong scaling of Simple and Components, averaged over five runs, with standard deviation error bars, on a 2048^3 snapshot of a Nyx simulation. (*Cori*, Haswell)

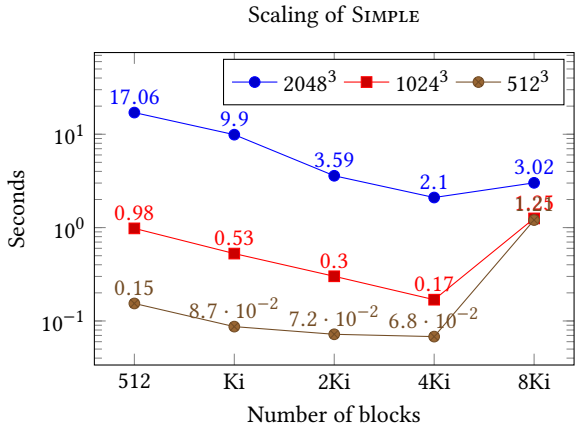

Figure 10: Strong and weak scaling of algorithm Simple. (*Edison*)

Figure 9 highlights a crucial disadvantage of Simple algorithm: for sufficiently high core counts, the blocks become small and components that previously fit inside a neighborhood of a block, now leave its bounds, triggering the second round of the algorithm. Because all blocks have to participate in each round of Simple algorithm, the second round is responsible for the increased communication and running time of Simple algorithm on 8Ki cores. (In contrast, in Components algorithm, only a few blocks are involved in the second round.) The increased communication is also responsible for a considerably larger variability in running times, as indicated by the error bars.

In order to see how each of the algorithms scales, we plot the timings for all 3 data sets: Figure 10 is for Simple algorithm and Figure 11 for Components algorithm. We observe that Simple algorithm stops scaling by the time the number of blocks increases from 4096 to 8192. The reason is that at this threshold, a single round of communication becomes insufficient. When we go to the

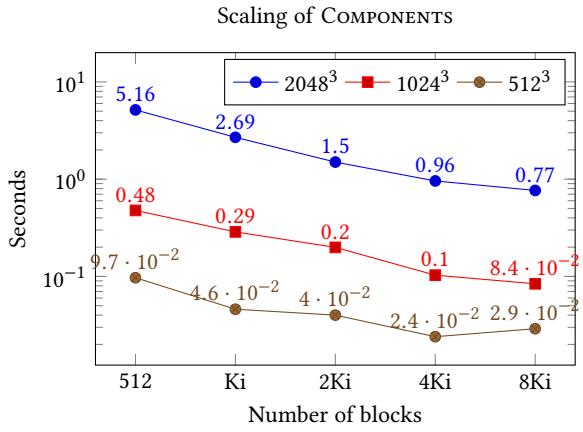

Figure 11: Scaling of Components algorithm on different data sets. (*Edison.*)

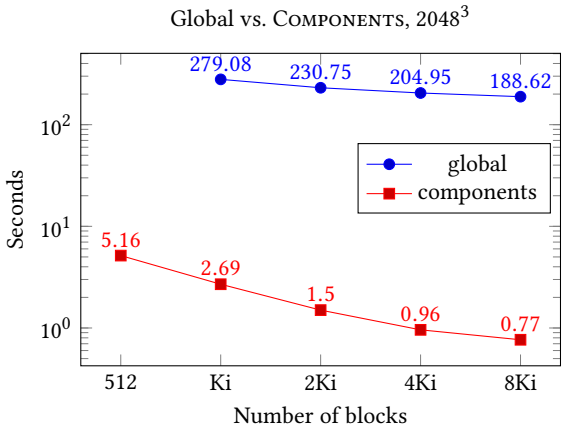

Figure 12: Comparison with the old (global) algorithm on a snapshot of a Nyx simulation of size 2048^3. (*Edison.*)

second round, instead of communicating with 26 neighbors, each block must communicate with 98 neighbors. The strong scaling of Components algorithm is better, but not perfect. For 2048^3 input, going from 512 to 4096 blocks, we get strong scaling efficiency of 67%; it drops to 42% going from 512 to 8192 blocks. As for weak scaling, when we go from 512^3 data set on 512 cores to 1024^3 data set on 4096 cores, the running time of both Components and Simple algorithm stays practically the same. The ratios are 1.09 and 1.06. As number of processors gets larger, and we run out of useful work, the ratios get worse: e.g., going from 1024^3 data set on 1024 processors to 2048^3 data set on 8192 processors increases the running time roughly by a factor of 2.7.

We also compare our code with the existing code for local–global merge tree computation, to demonstrate the practical efficiency of our results. This comparison might seem unfair: the existing code (implementation of [20] and [17]) must compute a much larger tree on the whole graph G, so it will inevitably run much slower. However, in the cosmological calculations, the merge trees are

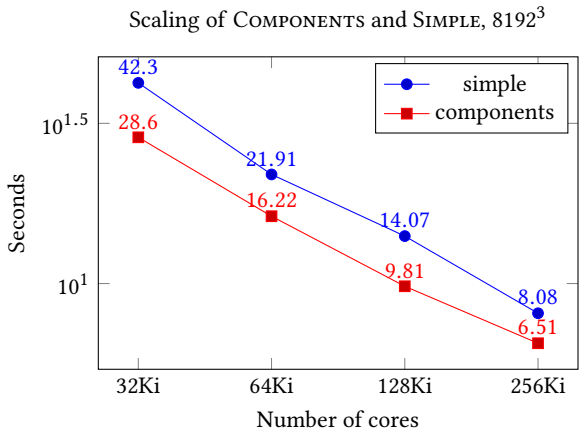

Figure 13: Comparison of the algorithms on 8192^3 file with fixed number of blocks. (*Cori*, KNL.)

analyzed at a given threshold ρ. For example, it is necessary to identify regions with high density and to integrate the density on each component individually. From the application's point of view, the two codes accomplish exactly the same task: we compare the time it takes to compute the integral using the old and the new codes. Figure 12 illustrates the results for the 2048^3 data set. Our algorithm outperforms the global reduction by two orders of magnitude: while the global reduction requires more than 3 minutes, our algorithm finishes computation in 3 seconds or less. Furthermore, the missing timing for the old code on 512 cores is because the available RAM is not enough to accommodate the intermediate trees during the reduction.

In order to evaluate our algorithms on a data set used in the state-of-the-art simulations, we applied our code to a 8192^3 dataset. These datasets are stored in the internal file format of Nyx, which in turn uses AMReX [22] library. The decomposition of the domain is predefined: we are bound to use 262,144 blocks. These experiments were performed on *Cori*, using the KNL nodes, but without using hyper-threading: we assigned one MPI process per core and ran the algorithms on 32768, 65536, 131072, and 262144 cores. The timings of Simple and Components algorithms for one input are plotted in Figure 13. Components algorithm is about 20-30% faster than Simple, and the two algorithms exhibit similar scaling behavior.

Figure 14 illustrates the performance of our algorithms on a 3-level AMR mesh, with a fixed number of blocks (roughly 24K). The coarsest level has resolution of 512^3; at the finest level this translates into effective resolution of 2048^3. Components algorithm is 5-6 times faster than Simple algorithm across all core counts, although the scaling of both slows down at higher core counts, as we run out of work. The explanation for Simple algorithm's poor performance is that in the multilevel AMR even small components span larger neighborhoods, because blocks at higher levels of the AMR hierarchy are smaller in the "absolute" coordinates of the lower levels. In the experiments of Figure 14, Simple algorithm has to use two rounds of communication.

Combustion. Figure 15 shows the performance of the Components algorithm on a snapshot of a combustion simulation [2],

Scaling of Components and Simple on multilevel AMR data

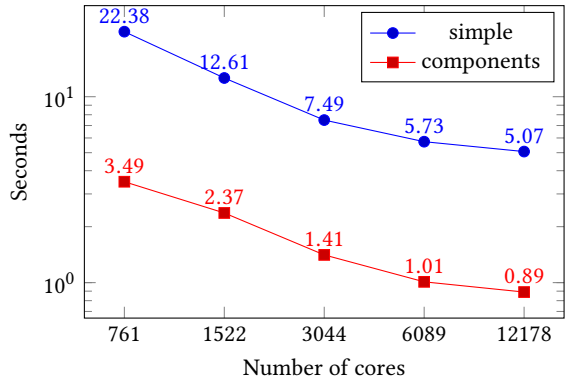

Figure 14: Comparison of the algorithms on multilevel AMR data with fixed number of blocks. (*Cori*, KNL.)

Scaling of Components on a snapshot of a combustion simulation

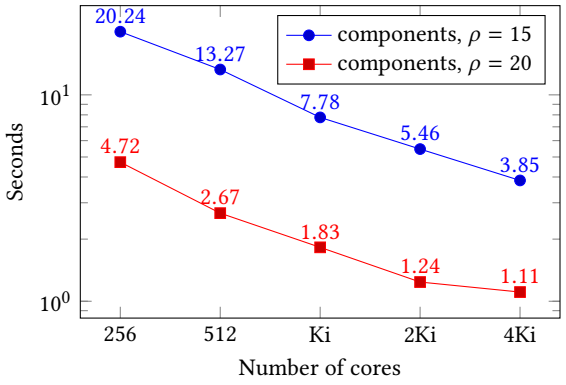

Figure 15: Scaling of Components on a snapshot of a combustion simulation (multilevel AMR data) for two different thresholds ρ, with fixed number of blocks. (*Cori*, Haswell.)

another domain that motivates the restricted problem. (Landge et al. [14] study combustion simulations, albeit on different data sets.) Our input data is a 4-level AMR grid, with the resolution of $64^2 \times 512$ at the coarsest level and $512^2 \times 4096$ at the finest level. In the simulation, as the flame is propagating through a premixed hydrogen-air bath, turbulent mixing processes are beginning to dominate those due to molecular transport. The local structure of the flame is distorted in a fundamental way. A signature of these changes is the overall fuel consumption rate, which is our input function. Pockets of intense fuel consumption, separated by regions of total extinction become distorted and mixed more uniformly; the propagating flame surface is transformed into a distributed burning region that exhibits an entirely different macroscopic behavior, putting us in the setting of many connected components above the threshold of interest.

As Figure 15 shows, the Components algorithm scales well while the data is large enough, compared to the number of cores used. The strong scaling is above 65% going from 256 to 1024 cores and above 40% going from 256 to 2048 cores. As we increase the core count to 4096, we run out of work: the overall time is dominated by communication, rather than computation, and the algorithm stops scaling.

5 CONCLUSION

Components algorithm, despite its bookkeeping overhead, performs better than Simple algorithm. Moreover, Components algorithm is optimal in its memory consumption because it sends only the minimum necessary amount of data: sending subtrees only to those blocks that they are connected to. Components algorithm processes 8192^3 data set on *Cori* KNL in roughly 6.5 seconds, using 262,144 cores. It takes Nyx simulation roughly 70 seconds to process a single time step, using 524,288 cores (with four hyper-threads per core). This means that it is practical to run our code *in situ* with the simulation, processing every time step of the simulation.

We finally reiterate the key assumption of this work: both our algorithms rely on the restricted graph G_ρ having a large number of relatively small components. If this assumption is not true, then distant blocks must exchange information, and the advantage of locality is lost.

ACKNOWLEDGEMENTS

The authors are grateful to Zarija Lukić, Hannah Ross, Marc Day, Weiqun Zhang, and Ann Almgren for their help with the data used in different experiments. This work 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, through the grant "Scalable Data-Computing Convergence and Scientic Knowledge Discovery," and by the use of resources of the National Energy Research Scientific Computing Center (NERSC). We rely on the AMR functionality in the DIY library, implemented with the support of the RAPIDS SciDAC Institute for Computer Science and Data. A. Nigmetov was partially supported by Austrian Science Fund (FWF) grant number P29984-N35 and is grateful to Michael Kerber and Johannes Wallner for organizational support, and to Marina Zyubina for her help with illustrations.

REFERENCES

[1] Ann S Almgren, John B Bell, Mike J Lijewski, Zarija Lukić, and Ethan Van Andel. Nyx: A massively parallel AMR code for computational cosmology. *The Astrophysical Journal*, 765(1):39, 2013.

[2] A J Aspden, M S Day, and J B Bell. Towards the distributed burning regime in turbulent premixed flames. *Journal of fluid mechanics*, 871:1–21, July 2019.

[3] Ulrich Bauer, Carsten Lange, and Max Wardetzky. Optimal topological simplification of discrete functions on surfaces. *Discrete & Computational Geometry*, 47(2):347–377, 2012.

[4] Janine C Bennett, Hasan Abbasi, Peer-Timo Bremer, Ray Grout, Attila Gyulassy, Tong Jin, Scott Klasky, Hemanth Kolla, Manish Parashar, Valerio Pascucci, et al. Combining in-situ and in-transit processing to enable extreme-scale scientific analysis. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, page 49. IEEE Computer Society Press, 2012.

[5] Janine C Bennett, Vaidyanathan Krishnamoorthy, Shusen Liu, Ray W Grout, Evatt R Hawkes, Jacqueline H Chen, Jason Shepherd, Valerio Pascucci, and Peer-Timo Bremer. Feature-based statistical analysis of combustion simulation data. *IEEE transactions on visualization and computer graphics*, 17(12):1822–1831, 2011.

[6] Peer-Timo Bremer, Gunther H. Weber, Julien Tierny, Valerio Pascucci, Marcus S. Day, and John B. Bell. Interactive exploration and analysis of large scale simulations using topology-based data segmentation. *IEEE Transactions on Visualization and Computer Graphics*, 17(9):1307–1325, 2011.

[7] Hamish Carr, Jack Snoeyink, and Ulrike Axen. Computing contour trees in all dimensions. *Computational Geometry—Theory and Applications*, 24(2):75–94, 2003.

[8] Hamish A Carr, Gunther H Weber, Christopher M Sewell, and James P Ahrens. Parallel peak pruning for scalable smp contour tree computation. In *2016 IEEE 6th Symposium on Large Data Analysis and Visualization (LDAV)*, pages 75–84. IEEE, 2016.

[9] Thomas Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction To Algorithms*. MIT Press, 2009.

[10] Sean Ahern (ed.). Scientific Discovery at the Exascale: Report from the DOE ASCR 2011 Workshop on Exascale Data Management, Analysis, and Visualization, February 2011.

[11] Hans Freudenthal. Simplizialzerlegungen von beschrankter flachheit. *Annals of Mathematics*, 43(3):580–582, 1942.

[12] Brian Friesen, Ann Almgren, Zarija Lukić, Gunther Weber, Dmitriy Morozov, Vincent Beckner, and Marcus Day. In situ and in-transit analysis of cosmological simulations. *Computational Astrophysics and Cosmology*, 3(1):4, 2016.

[13] C Gueunet, P Fortin, J Jomier, and J Tierny. Task-based augmented merge trees with fibonacci heaps. In *2017 IEEE 7th Symposium on Large Data Analysis and Visualization (LDAV)*, pages 6–15, October 2017.

[14] A G Landge, V Pascucci, A Gyulassy, J C Bennett, H Kolla, J Chen, and P Bremer. In-Situ feature extraction of large scale combustion simulations using segmented merge trees. In *SC '14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1020–1031, November 2014.

[15] Ajith Mascarenhas, Ray W Grout, Peer-Timo Bremer, Evatt R Hawkes, Valerio Pascucci, and Jacqueline H Chen. Topological feature extraction for comparison of terascale combustion simulation data. In *Topological Methods in Data Analysis and Visualization*, pages 229–240. Springer, 2011.

[16] Dmitriy Morozov and Tom Peterka. Block-parallel data analysis with diy2. In *2016 IEEE 6th Symposium on Large Data Analysis and Visualization (LDAV)*, pages 29–36. IEEE, 2016.

[17] Dmitriy Morozov and Gunther Weber. Distributed merge trees. In *ACM SigPlan notices*, volume 48, pages 93–102. ACM, 2013.

[18] Dmitriy Morozov and Gunther Weber. Distributed Contour Trees. In Peer-Timo Bremer, Ingrid Hotz, Valerio Pascucci, and Ronald Peikert, editors, *Topological Methods in Data Analysis and Visualization III*. Springer International Publishing, 2014.

[19] Valerio Pascucci and Kree Cole-McLaughlin. Parallel computation of the topology of level sets. *Algorithmica*, 38(1):249–268, 2003.

[20] Dmitriy Smirnov and Dmitriy Morozov. Triplet merge trees. In *Topological Methods in Data Analysis and Visualization V: Theory, Algorithms, and Applications (TopoInVis'17)*, 2019.

[21] Leslie G Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, August 1990.

[22] Weiqun Zhang, Ann Almgren, Vince Beckner, John Bell, Johannes Blaschke, Cy Chan, Marcus Day, Brian Friesen, Kevin Gott, Daniel Graves, Max Katz, Andrew Myers, Tan Nguyen, Andrew Nonaka, Michele Rosso, Samuel Williams, and Michael Zingale. AMReX: a framework for block-structured adaptive mesh refinement. *Journal of Open Source Software*, 4(37):1370, May 2019.

Appendix: Artifact Description/Artifact Evaluation

SUMMARY OF THE EXPERIMENTS REPORTED

We ran two algorithms for merge tree computation on NERSC Edison and Cori (Haswell, KNL) machines, using DIY library version 3.5.dev1.

ARTIFACT AVAILABILITY

Software Artifact Availability: Some author-created software artifacts are NOT maintained in a public repository or are NOT available under an OSI-approved license.

Hardware Artifact Availability: There are no author-created hardware artifacts.

Data Artifact Availability: Some author-created data artifacts are NOT maintained in a public repository or are NOT available under an OSI-approved license.

Proprietary Artifacts: No author-created artifacts are proprietary.

List of URLs and/or DOIs where artifacts are available:

N/A

BASELINE EXPERIMENTAL SETUP, AND MODIFICATIONS MADE FOR THE PAPER

Relevant hardware details: NERSC Edison, Cori

Compilers and versions: GCC v7.3.0