Theory Comput Syst (2011) 48: 269–296
DOI 10.1007/s00224-009-9242-2

Optimal Cache-Oblivious Mesh Layouts

Michael A. Bender · Bradley C. Kuszmaul · Shang-Hua Teng · Kebin Wang

Published online: 27 October 2009

Abstract A *mesh* is a graph that divides physical space into regularly-shaped regions. Meshes computations form the basis of many applications, including finite-element methods, image rendering, collision detection, and N-body simulations. In

M.A. Bender was supported in part by NSF Grants CCF 0621439/0621425, CCF 0540897/05414009, CCF 0634793/0632838, CNS 0627645, and CCF 0937822 and by DOE Grant DE-FG02-08ER25853.
B.C. Kuszmaul was supported in part by the Singapore-MIT Alliance, NSF Grant ACI-0324974, and DOE Grant DE-FG02-08ER25853.
S.-H. Teng was supported in part by NSF grants CCR-0311430 and ITR CCR-0325630.

M.A. Bender (✉)
Department of Computer Science, Stony Brook University, Stony Brook, NY 11794-4400, USA
e-mail: bender@cs.sunysb.edu

M.A. Bender · B.C. Kuszmaul
Tokutek, Inc., 1 Militia Drive, Lexington, MA 02421, USA
url: http://www.tokutek.com

B.C. Kuszmaul
Computer Science and Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, MA 02139, USA
e-mail: bradley@mit.edu

S.-H. Teng
Computer Science Department, University of Southern California, 941 Bloom Walk, Los Angeles, CA 90089-0781, USA
e-mail: shanghua@usc.edu

S.-H. Teng
Akamai Technologies, Inc., Cambridge, MA 02124, USA

K. Wang
Computer Science Department, Boston University, Boston, MA 02215, USA
e-mail: kwang@cs.bu.edu

one important mesh primitive, called a *mesh update*, each mesh vertex stores a value and repeatedly updates this value based on the values stored in all neighboring vertices. The performance of a mesh update depends on the layout of the mesh in memory. Informally, if the mesh layout has good data locality (most edges connect a pair of nodes that are stored near each other in memory), then a mesh update runs quickly.

This paper shows how to find a memory layout that guarantees that the mesh update has asymptotically optimal memory performance for any set of memory parameters. Specifically, the cost of the mesh update is roughly the cost of a sequential memory scan. Such a memory layout is called *cache-oblivious*. Formally, for a d-dimensional mesh G, block size B, and cache size M (where $M = \Omega(B^d)$), the mesh update of G uses $O(1 + |G|/B)$ memory transfers. The paper also shows how the mesh-update performance degrades for smaller caches, where $M = o(B^d)$.

The paper then gives two algorithms for finding cache-oblivious mesh layouts. The first layout algorithm runs in time $O(|G| \log^2 |G|)$ both in expectation and with high probability on a RAM. It uses $O(1 + |G| \log^2(|G|/M)/B)$ memory transfers in expectation and $O(1 + (|G|/B)(\log^2(|G|/M) + \log|G|))$ memory transfers with high probability in the cache-oblivious and disk-access machine (DAM) models. The layout is obtained by finding a fully balanced decomposition tree of G and then performing an in-order traversal of the leaves of the tree.

The second algorithm computes a cache-oblivious layout on a RAM in time $O(|G| \log|G| \log\log|G|)$ both in expectation and with high probability. In the DAM and cache-oblivious models, the second layout algorithm uses $O(1 + (|G|/B) \log(|G|/M) \min\{\log\log|G|, \log(|G|/M)\})$ memory transfers in expectation and $O(1 + (|G|/B)(\log(|G|/M) \min\{\log\log|G|, \log(|G|/M)\} + \log|G|))$ memory transfers with high probability. The algorithm is based on a new type of decomposition tree, here called a *relax-balanced decomposition tree*. Again, the layout is obtained by performing an in-order traversal of the leaves of the decomposition tree.

Keywords Cache-oblivious · Decomposition tree · Fully-balanced decomposition tree · Geometric separator · Mesh layout · Relax-balanced decomposition tree

1 Introduction

A *mesh* is a graph that represents a division of physical space into regions, called *simplices*. Simplices are typically triangular (in 2D) or tetrahedral (in 3D). They are *well shaped*, which informally means that they cannot be long and skinny, but must be roughly the same size in any direction. Meshes form the basis of many computations such as finite-element methods, image rendering, collision detection, and N-body simulations. Constant-dimension meshes have nodes of constant-degree.

In one important mesh primitive, each mesh vertex stores a value and repeatedly updates this value based on the values stored in all neighboring vertices. Thus, we view the mesh as a weighted graph $G = (V, E, w, e)$ ($w : V \to \mathbb{R}$, $e : E \to \mathbb{R}^+$). For each vertex $i \in V$, we repeatedly recompute its weight w_i as follows:

$$w_i = \sum_{(i,j)\in E} w_j e_{ij}.$$

We call this primitive a *mesh update*. Expressed differently, a mesh update is the sparse matrix-vector multiplication, where the matrix is the (weighted) adjacency matrix of G, and vectors are the vertex weights.

On a *random access machine (RAM)* (a flat memory model), a mesh update runs in linear time, regardless of how the data is laid out in memory. In contrast, on a modern computer with a hierarchical memory, how the mesh is laid out in memory can affect the speed of the computation substantially. This paper studies the *mesh layout problem*, which is how to lay out a mesh in memory, so that mesh updates run rapidly on a hierarchical memory.

We analyze the mesh layout problem in the *disk-access machine (DAM) model* [2] (also known as the *I/O-model*) and in the *cache-oblivious (CO) model* [17]. The DAM model is an idealized two-level memory hierarchy. These two levels could represent L2 cache and main memory, main memory and disk, or any other pair of levels. The small level (herein called *cache*) has size M, and the large level (herein called *disk*) has unbounded size. Data is transferred between the two levels in blocks of size B; we call these *memory transfers*. Thus, a memory transfer is a cache-miss if the DAM represents L2 cache and main memory and is a page fault, if the DAM represents main memory and disk.

A memory transfer has unit cost. The objective is to minimize the number of memory transfers. Focusing on memory transfers, to the exclusion of other computation, frequently provides a good model of the running time of an algorithm on a modern computer. The *cache-oblivious model* is essentially the DAM model, except that the values of B and M are unknown to the algorithm or the coder. The main idea of cache-obliviousness is this: If an algorithm performs an asymptotically optimal number of memory transfers on a DAM, but the algorithm is not parameterized by B and M, then the algorithm also performs an asymptotically optimal number of memory transfers on an arbitrary unknown, multilevel memory hierarchy.

The cost of a mesh update in the DAM and cache-oblivious models depends on how the mesh is laid out in memory. An update to a mesh $G = (V, E)$ is just a graph traversal. If we store G's vertices arbitrarily in memory, then the update could cost as much as $O(|V| + |E|) = O(|G|)$ memory transfers, one transfer for each vertex and each edge. In this paper we achieve only $\Theta(1 + |G|/B)$ memory transfers. This is the cost of a sequential scan of a chunk of memory of size $O(|G|)$, which is asymptotically optimal.

Our mesh layout algorithms extend earlier ideas from VLSI theory. Classical VLSI-layout algorithms turn out to have direct application in scientific and I/O-efficient computing. Although these diverse areas may appear unrelated, there are important parallels. For example, in a good mesh layout, vertices are stored in (one-dimensional) *memory locations* so that most mesh edges are short; in a good VLSI layout, graph vertices are assigned to (two-dimensional) *chip locations* so that most edges are short (to cover minimal area).

1.1 Results

We give two algorithms for laying out a constant-dimension well-shaped mesh $G = (V, E)$ so that updates run in $\Theta(1+|G|/B)$ memory transfers, which is $\Theta(1+|V|/B)$ since the mesh has constant degree.

Our first layout algorithm runs in time $O(|G|\log^2|G|)$ on a RAM both in expectation and with high probability.[1] In the DAM and cache-oblivious models, the algorithm uses $O(1+(|G|/B)\log^2(|G|/M))$ memory transfers in expectation and $O(1+(|G|/B)(\log^2(|G|/M)+\log|G|))$ memory transfers with high probability. The layout algorithm is based on decomposition trees and fully balanced decomposition trees [7, 24]; specifically, our mesh layout is obtained by performing an in-order traversal of the leaves of a fully-balanced decomposition tree. Decomposition trees were developed several decades ago as a framework for VLSI layout [7, 24], but they are well suited for mesh layout. However, the original algorithm for building fully-balanced decomposition trees is too slow for our uses (it appears to run in time $O(|G|^{\Theta(b)})$, where b is the degree bound of the mesh). Here we develop a new algorithm that is faster and simpler.

Our second layout algorithm, this paper's main result, runs in time $O(|G|\log|G|\log\log|G|)$ on a RAM both in expectation and with high probability. In the DAM and cache-oblivious models, the algorithm uses $O(1+(|G|/B)\log(|G|/M)\min\{\log\log|G|,\log(|G|/M)\})$ memory transfers in expectation and $O(1+(|G|/B)(\log(|G|/M)\min\{\log\log|G|,\log(|G|/M)\}+\log|G|))$ memory transfers with high probability.

The algorithm is based on a new type of decomposition tree, which we call a *relax-balanced decomposition tree*. As before, our mesh layout is obtained by performing an in-order traversal of the leaves of a relax-balanced decomposition tree. By carefully relaxing the requirements of decomposition trees, we can retain asymptotically optimal mesh updates, while improving construction by nearly a logarithmic factor.

The mesh-update guarantees require a *tall-cache assumption* on the memory system that $M=\Omega(B^d)$, where d is the dimension of the mesh. We also show how the performance degrades for small caches, where $M=o(B^d)$. If the cache only has size $O(B^{d-\epsilon})$, then the number of memory transfers increases to $O(1+|G|/B^{1-\epsilon/d})$.

In addition to the main results listed above, this paper has contributions extending beyond I/O-efficient computing. First, our algorithms for building fully-balanced decomposition trees are faster and simpler than previously known algorithms. Second, our relax-balanced decomposition trees may permit some existing algorithms based on decomposition trees to run more quickly. Third, the techniques in this paper yield simpler and improved methods for generating k-way partitions of meshes, earlier shown in [23]. More generally, we cross-pollinate several fields, including I/O-efficient computing, VLSI layout, and scientific computing.

2 Geometric Separators and Decomposition Trees

In this section we review the geometric-separator theorem [27], which we use for partitioning constant-dimensional meshes. We then review decomposition trees [24]. Finally, we show how to use geometric separators to build decomposition trees for well shaped meshes.

[1] For input size N and event E, we say that E occurs *with high probability* if for any constant $c>0$ there exists a proper choice of constants defining the event such that $\Pr\{E\}\geq 1-N^{-c}$.

2.1 Geometric Separators

A finite-element mesh is a decomposition of a geometric domain into a collection of interior-disjoint *simplices* (e.g., triangles in 2D and tetrahedra in 3D), so that two simplices can only intersect at a lower dimensional simplex. Each simplicial element of the mesh must be *well shaped*. Well shaped means that there is a constant upper bound to the aspect ratio, that is, the ratio of the radius of the smallest ball containing the element to the radius of the largest ball contained in the element [33].

A *partition* of a graph $G = (V, E)$ is a division of G into disjoint subgraphs $G_0 = (V_0, E_0)$ and $G_1 = (V_1, E_1)$ such that $V_0 \cap V_1 = \emptyset$, and $V_0 \cup V_1 = V$. G_0 and G_1 is a *β-partition* of G if they are a partition of G and $|V_0|, |V_1| \leq \beta|V|$. We let $E(G_0, G_1)$ denote the set of edges in G crossing from V_0 to V_1, and $E(v, G_1)$ denote the set of edges in G connecting vertex v to the vertices of G_1. For a function f, $G = (V, E)$ has a *family of (f, β)-partitions* if for each subset $S \subseteq V$ and induced graph $G_S = (V_S, E_S)$, graph G_S has a β-partition of $G_{S_0} = (V_{S_0}, E_{S_0})$ and $G_{S_1} = (V_{S_1}, E_{S_1})$ such that $|E_S - E_{S_0} - E_{S_1}| \leq f(|V_S|)$.

The following separator theorem of Miller, Teng, Thurston, and Vavasis [27] shows that meshes can be partitioned efficiently:

Theorem 1 (Geometric Separators [27]) *Let $G = (V, E)$ be a well shaped finite-element mesh in d dimensions ($d > 1$). For constants ϵ ($0 < \epsilon < 1$) and $c(\epsilon, d)$ depending only on ϵ and d, a $(f(N) = O(N^{1-1/d}), (d + 1 + \epsilon)/(d + 2))$-partition of G can be computed in $O(d|G| + c(\epsilon, d))$ time with probability at least $1/2$.*

The separator algorithm from [27] works as follows. First, project the coordinates of the vertices of the input graph G onto the surface of a unit sphere in $(d + 1)$-dimensions. The projection of each point is independent of all other input points and takes constant time. Sample a constant number of points from all projected points uniformly at random. Compute a centerpoint of the sampled points. (A *centerpoint* of a point set in d-dimensions is a point such that every hyperplane through the centerpoint divides the point set approximately evenly, i.e., in the ratio of d to 1 or better.) Rotate and then dilate the sampled points. Both the rotation and dilation are functions of the centerpoint and the dimension d. Choose a random great circle on this unit sphere. (A *great circle* of a sphere is a circle on the sphere's surface that evenly splits the sphere.) Map the great circle back to a sphere in the d-dimensional space by reversing the dilation, the rotation, and the projection. Now use this new sphere to divide the vertices and the edges of the input graph.

Now more mechanics of the algorithm. Mesh G is stored in an array. Each vertex of G is stored with its index (i.e., name), its coordinates, and all of its adjacent edges, including the index and coordinates of all neighboring vertices. (This mesh representation means that each edge is stored twice, once for each of the edge's two vertices.)

To run the algorithm, scan the vertices and edges in G after obtaining the sphere separator. During the scan, divide the vertices into two sets, G_0, containing the vertices inside the new sphere and G_1, containing the vertices outside the sphere. Mark an edge as "crossing" if the edge crosses from G_0 to G_1. Verify that the number of

crossing edges, $|E(G_0, G_1)|$, is $O(|G|^{1-1/d})$, and if not, repeat. The cost of this scan is $O(|G|/B + 1)$ memory transfers.

The geometric separator algorithm has the following performance:

Corollary 2 *Let $G = (V, E)$ be a well shaped finite-element mesh in d dimensions ($d > 1$). For constants ϵ ($0 < \epsilon < 1$) and $c(\epsilon, d)$ depending only on ϵ and d, the geometric-separator algorithm finds an $(f(N) = O(N^{1-1/d}), (d+1+\epsilon)/(d+2))$-partition of G. The algorithm runs in $O(|G|)$ on a RAM and uses $O(1 + |G|/B)$ memory transfers in the DAM and cache-oblivious models, both in expectation and with probability at least $1/2$. With high probability, the geometric-separator algorithm completes in $O(|G| \log |G|)$ on a RAM and uses $O(1 + |G| \log |G|/B)$ memory transfers in the DAM and cache-oblivious models.*

Proof A linear scan of G takes time $O(|G|)$ and uses an asymptotically optimal number of memory transfers. We expect to find a good separator after a constant number of trials, and so the expectation bounds follow by linearity of expectation. The probability that after selecting $c \lg |G|$ candidate separators, none are good is at most $1/2^{c \lg |G|} = |G|^{-c}$. Thus, with high probability, the geometric separator algorithm completes in $O(|G| \log |G|)$ on a RAM and uses $O(1 + |G| \log |G|/B)$ memory transfers in the DAM and cache-oblivious models. The separator algorithm is cache-oblivious since it is not parameterized by B or M. □

2.2 Decomposition Trees

A *decomposition tree* T_G of a graph $G = (V, E)$ is a recursive partitioning of G. The root of T_G is G. Root G has left and right children G_0 and G_1, and grandchildren G_{00}, G_{01}, G_{10}, G_{11}, and so on recursively down the tree. Graphs G_0 and G_1 partition G, graphs G_{00} and G_{01} partition G_0, and so on. More generally, a node in the decomposition tree is denoted G_p ($G_p \subset G$), where p is a bit string representing the path to that node from the root. We call p the *id* of G_p. We say that a decomposition tree is *β-balanced* if for all siblings $G_{p0} = (V_{p0}, E_{p0})$ and $G_{p1} = (V_{p1}, E_{p1})$ in the tree, $|V_{p0}|, |V_{p1}| \leq \beta |V_p|$. We say that a decomposition tree is *balanced* if $\beta = 1/2$. For a function f, *graph G has an f decomposition tree* if for all (nonleaf) nodes G_p in the decomposition tree, $|E(G_{p0}, G_{p1})| \leq f(|V_p|)$. A β-balanced f decomposition tree is abbreviated as an (f, β)-decomposition tree.

For a parent node G_p and its children G_{p0} and G_{p1}, there are several categories of edges. *Inner edges* connect vertices that are both in G_{p0} or both in G_{p1}. *Crossing edges* connect vertices in G_{p0} to vertices in G_{p1}. *Outgoing edges* of G_{p0} (resp. G_{p1}) connect vertices in G_{p0} (resp. G_{p1}) to vertices in neither set, i.e., to vertices in $G - G_p$. *Outer edges* of G_{p0} (resp. G_{p1}) connect vertices in G_{p0} (resp. G_{p1}) to vertices in $G - G_{p0}$ (resp. $G - G_{p1}$); thus an outer edge is either a crossing edge or an outgoing edge. More formally, $\mathsf{inner}(G_{p0}) = E(G_{p0}, G_{p0})$, $\mathsf{crossing}(G_p) = E(G_{p0}, G_{p1})$, $\mathsf{outgoing}(G_{p0}) = E(G_{p0}, G - G_p)$, and $\mathsf{outer}(G_{p0}) = E(G_{p0}, G - G_{p0})$.

We build a decomposition tree T_G of mesh G recursively. First we run the geometric separator algorithm on the root G to find the left and right children, G_0 and G_1. Then we recursively build the decomposition tree rooted at G_0 and then the decomposition tree rooted at G_1. (Thus, the right child of T_G is not processed until the whole left subtree is built.)

The decomposition tree is encoded as follows. Each leaf node G_q for T_G stores the single vertex v and the bit string q (the root-to-leaf path). The leaf nodes of T_G are stored contiguously in an array L_G. The bit string q contains enough information to determine which nodes (subgraphs) of T_G contain v—specifically any node $G_{\hat{q}}$, where $\hat{q}$ is a prefix of q (including q). As mentioned earlier, each vertex is stored along with its coordinates, adjacent edges, and coordinates of all neighboring vertices in G. (Recall that each edge is therefore stored twice, once for each of the edge's vertices.) Each edge e in G is a crossing edge for exactly one node in the decomposition tree T_G. In T_G, each edge e also stores the id p of the tree node G_p for which e is a crossing edge. The bit strings on nodes and edges therefore contains enough information to determine which edges are crossing, inner, and outer for which tree nodes. Specifically, $e \in \mathsf{crossing}(G_p)$. Let $\hat{p}$ be a prefix of p that is strictly shorter ($p \neq \hat{p}$); then $e \in \mathsf{inner}(G_{\hat{p}})$. Let $\tilde{p}$ be bit string representing a node in T_G where p is a strictly shorter prefix of $\tilde{p}$ ($p \neq \tilde{p}$). Then $e \in \mathsf{outer}(G_{\tilde{p}})$. If $\tilde{p}0$ and $\tilde{p}1$ represent nodes in T_G, then $e \in \mathsf{outgoing}(G_{\tilde{p}0})$ or $e \in \mathsf{outgoing}(G_{\tilde{p}1})$.

Thus, decomposition tree T_G is laid out in memory by storing the leaves in order in an array L_G. We do not need to store internal nodes explicitly because the bit strings on nodes and edges encode the tree structure.

Here are a few facts about our layout of T_G. Given any two nodes G_p and G_q of L_G, the common prefix of p and q is the smallest node in T_G containing all vertices in both G_p and G_q. All the vertices in any node G_p of T_G are stored in a single contiguous chunk of the array. Thus, we can identify for G_p, which edges are inner, crossing, outer, and outgoing by performing a single linear scan of size $O(|G_p|)$.

We construct the decomposition tree T_G by recursively partitioning of G. While T_G is in the process of being constructed, its encoding is similar to the above, except that (1) a leaf node G_q may contain more than one vertex, and (2) some edges may not yet be labelled as crossing. Thus, when the process begins, T_G is just a single leaf comprising G. The nodes are stored in a single array of size $O(|G|)$ and are stored in an arbitrary order. Then we run the geometric separator algorithm. Once we find a good separator, we partition G into G_0 and G_1, and we store G_0 before G_1 in the same array. We label vertices of G_0 with bit string 0 and vertices of G_1 with bit string 1. We then run through and label all crossing edges with the appropriate bit string (for the leaf node, the empty string). Now the nodes in each of G_0 and G_1 are stored in an arbitrary order, but the subarray containing G_0 is stored before the subarray containing G_1. We then apply the geometric separator algorithm for G_0. We partition into G_{00} and G_{01}, label vertices in G_0 with 00 or 01, and label all crossing edges of G_0 with the bit string 0; we then do the same for G_{00} and so on recursively until all leaf nodes are graphs containing a single vertex.

We now give the complexity of building the decomposition tree. Our high-probability bounds are based on the following observation involving a coin with a constant probability of heads. In order to get at least one head with probability at

least $1 - 1/\text{poly}(N)$, $\Theta(\log N)$ flips are necessary and sufficient. In order to get $\Theta(\log N)$ heads with probability at least $1 - 1/\text{poly}(N)$, the asymptotics do not change; $\Theta(\log N)$ flips are still necessary and sufficient. The following lemma can be proved by Chernoff bounds (or otherwise):

Lemma 3 *Consider $S \geq c \log N$ flips of a coin with a constant probability of heads, for sufficiently large constant c. With probability at least $1 - 1/\text{poly}(N)$, $\Theta(S)$ of the flips are heads.*

Theorem 4 *Let $G = (V, E)$ be a well shaped finite-element mesh in d dimensions ($d > 1$). Mesh G has a $(2d + 3)/(2d + 4)$-balanced-$O(|V|^{1-1/d})$ decomposition tree. On a RAM, the decomposition tree can be computed in time $O(|G| \log |G|)$ both in expectation and with high probability. The decomposition tree can be computed in the DAM and cache-oblivious models using $O(1 + (|G|/B) \log (|G|/M))$ memory transfers in expectation and $O(1 + (|G|/B) \log |G|)$ memory transfers with high probability.*

Proof We first establish that the tree construction takes time $O(|G| \log |G|)$ on a RAM in expectation. The height of the decomposition tree is $O(\log |G|)$, and the total size of all subgraphs at each height is $O(|G|)$. Since the decomposition of a subgraph takes expected linear time, the time bounds follow by linearity of expectation.

We next establish that the tree construction uses $O(1 + (|G|/B) \log(|G|/M))$ expected memory transfers in the DAM and cache-oblivious models. Because we build the decomposition tree recursively, we give a recursive analysis. The base case is when a subtree first has size less than M. For the base case, the cost to build the entire subtree is $O(M/B)$ because this is the cost to read all blocks of the subtree into memory. Said differently, once a subgraph is a constant fraction smaller than M, the cost to build the decomposition tree from the subgraph is 0, because all necessary memory blocks already reside in memory. For the recursive step, recall that when a subgraph G_p has size greater than M, the decomposition of a subgraph takes expected $O(|G_p|/B)$ memory transfers, because this is the cost of a linear scan. Thus, there are $O(\log (|G|/M))$ levels of the tree with subgraphs bigger than M, so the algorithms uses expected $O(1 + (|G|/B) \log (|G|/M))$ memory transfers.

We next establish the high-probability bounds. We show that the building process uses $O(|G| \log |G|)$ time on a RAM and $O(1 + |G| \log |G|/B)$ memory transfers in the DAM and the cache-oblivious models with high probability.

First consider all nodes that have size $\Omega(|G|/\log |G|)$. There are $\Theta(\log |G|)$ such nodes. To build these nodes, we require a total of $\Theta(\log |G|)$ good separators. We can view finding these separators as a coin-flipping game, where we need $\Theta(\log |G|)$ heads; by Lemma 3 we require $\Theta(\log |G|)$ coin flips. However, separators near the top of the tree are more expensive to find than separators deeper in the tree. We bound the cost to find all of these separators by the cost to build the root separator. Thus, building these nodes uses time $O(|G| \log |G|)$ and $O(1 + |G| \log |G|/B)$ memory transfers with high probability. This is now the dominant term in the cost to build the decomposition tree.

Further down the tree, where nodes have size $O(|G|/\log|G|)$, the analysis is easier. Divide the nodes to be partitioned into groups whose sizes are within a constant factor of each other. Now each group contains $\Omega(\log|G|)$ elements. Thus, by Lemma 3 the time to build the rest of the tree with high probability equals the time in expectation, which is $\Theta(|G|\log|G|)$.

We now finish the bound on the number of memory transfers. As above, because we build the decomposition tree recursively, subtrees a constant fraction smaller than M are build for free. Also, because each group contains $\Omega(\log|G|)$ elements, the cost to build these lower levels in the tree with high probability equals the expected cost, which is $O(1+(|G|/B)\log(|G|/M))$. This cost is dominated by the cost to build the nodes of size $\Omega(|G|/\log|G|)$. □

3 Fully-Balanced Decomposition Trees for Meshes

In this section we define fully-balanced partitions and fully-balanced decomposition trees. We give algorithms for generating these structures on a well shaped mesh G. As we show in Sect. 4, we use a fully-balanced decomposition tree of a mesh G to generate a cache-oblivious mesh layout of G. Our construction algorithm is an improvement over [7, 24] in two respects. First the algorithm is faster, requiring only $O(|G|\log^2|G|)$ operations in expectation and with high probability, $O(1+(|G|/B)\log^2(|G|/M))$ memory transfers in expectation, and $O(1+(|G|/B)(\log^2(|G|/M)+\log|G|))$ memory transfers with high probability. Second, the result is simplified, no longer relying on a complicated theorem of [18].

This section makes it easier to present the main result of the paper, which appears in Sect. 5.

3.1 Fully-Balanced Partitions

To begin, we define a fully-balanced partition of a subgraph G_p of G. A *fully-balanced f-partition of $G_p \subseteq G$* is a partitioning of $G_p = (V_p, E_p)$ into two subgraphs $G_{p0} = (V_{p0}, E_{p0})$ and $G_{p1} = (V_{p1}, E_{p1})$ such that

- $|\mathsf{crossing}(G_p)| \leq f(|V_p|)$,
- $|V_{p0}| = |V_{p1}| \pm O(1)$, and
- $|\mathsf{outgoing}(G_{p0})| = |\mathsf{outgoing}(G_{p1})| \pm O(1)$.

We give the following result before presenting our algorithm for computing fully-balanced partitions. The existence proof and time complexity comprise the easiest case in [18].

Lemma 5 *Given an array L of N elements, where each element is marked either blue or red, there exists a subarray that contains half of the blue elements to within one and half of red elements to within one. Such a subarray can be found in $O(N)$ time and $O(1+N/B)$ memory transfers cache-obliviously.*

Proof This result is frequently described in terms of "necklaces." Conceptually, attach the two ends of the array together to make a necklace. By a simple continuity

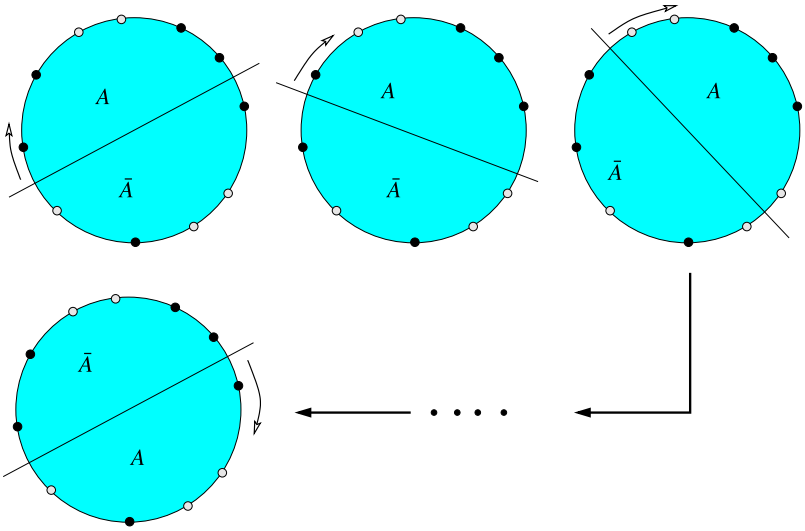


Fig. 1 Unfilled beads represent *blue* elements and filled beads represent *red* elements. *Pick* an arbitrary initial bisection A and $\bar{A}$ of the necklace. Here A contains more than half of all *blue beads*. (We can focus exclusively on *blue beads* because if A contains half of the *blue beads* to within one, it also contains half of *red beads* to within one.) We "turn" the bisection clockwise so that A takes one bead from $\bar{A}$ and relinquishes one bead to $\bar{A}$. Thus, the number of *blue beads* in A can increase/decrease by one or remain the same after each turn. However, after $N/2$ turns, A becomes $\bar{A}$, which contains less than half of all *blue beads*. So by a continuity argument, A contains half of all *blue beads* after some number of turns. The argument is similar for both odd and even N

argument (the easiest case of that in [18]), the necklace can be split into two pieces, A and $\bar{A}$, using two cuts such that both pieces have the same number of blue elements to within one and the same number of red elements to within one. (For details of the continuity argument, see Fig. 1.) Translating back to the array, at least one of A and $\bar{A}$ does not contain the connecting point and is contiguous.

To find a good subarray, first scan L to count the number of blue elements and the number of red elements. Now rescan L, maintaining a window of size $N/2$. The window initially contains the first half of L and at the end contains the second half of L. (For odd N, the middle element of the array appears in all windows.) Stop the scan once the window has the desired number of red and blue elements.

Since only linear scans are used, the algorithm is cache-oblivious and requires $\Theta(1 + N/B)$ memory transfers. □

We now present an algorithm for computing fully-balanced partitions. Given $G_p \subseteq G$, and a $(f(N) = O(N^\alpha), \beta)$-partitioning geometric separator, FullyBalancedPartition(G_p) computes a fully-balanced $(f(N) = O(N^\alpha))$-partition G_{px} and G_{py} of G_p.

FullyBalancedPartition(G_p)

1. *Build a decomposition tree*—Build a decomposition tree T_{G_p} of G_p using the $(f(N) = O(N^\alpha), \beta)$-partitioning geometric separator.
2. *Build a red-blue array*—Build an array of blue and red elements based on the decomposition tree T_{G_p}. Put a blue element for each leaf G_q in T_{G_p}; thus there is a blue element for each vertex v in G_p. Now insert some red elements after each blue element. Specifically, after the blue element representing vertex v, insert $E(v, G - G_p)$ red elements. Thus, the blue elements represent vertices in $G_p = (V_p, E_p)$ for a total of $|V_p|$ blue elements, while the red elements represent edges to vertices in $G - G_p$, for a total of $E(G_p, G - G_p)$ red elements.
3. *Find a subarray in the red-blue array*—Find a subarray of the red-blue array based on Lemma 5. Now partition the vertices in G_p based on this subarray. Specifically, put the vertices representing blue elements in the subarray in set V_{px} and put the remaining vertices in G_p in set V_{py}.
4. *Partition G_p*—Compute G_{px} and G_{py} from V_{px} and V_{py}. This computation also means scanning edges to determine which edges are internal to G_{px} and G_{py} and which have now become external.

We first establish the running time of FullyBalancedPartition(G_p).

Lemma 6 *Given a graph G_p that is a subgraph of a well shaped mesh G,* FullyBalancedPartition(G_p) *runs in $O(|G_p| \log |G_p|)$ on a RAM, both in expectation and with high probability (i.e., probability at least $1 - 1/\text{poly}(|G_p|)$). In the DAM and cache-oblivious models,* FullyBalancedPartition(G_p) *uses $O(1 + (|G_p|/B) \log (|G_p|/M))$ memory transfers in expectation and $O(1 + |G_p| \log |G_p|/B)$ memory transfers with high probability.*

Proof According to Theorem 4, Step 1 of FullyBalancedPartition(G_p) (computing T_{G_p}) takes time $O(|G_p| \log |G_p|)$ on a RAM, both in expectation and with high probability. In the DAM and cache-oblivious models, this steps requires $O(1 + (|G_p|/B) \log (|G_p|/M))$ memory transfers in expectation and $O(1 + |G_p| \log |G_p|/B)$ memory transfers with high probability. Steps 2–4 of FullyBalancedPartition(G_p) each require linear scans of an array of size $O(|G_p|)$, and therefore are dominated by Step 1. □

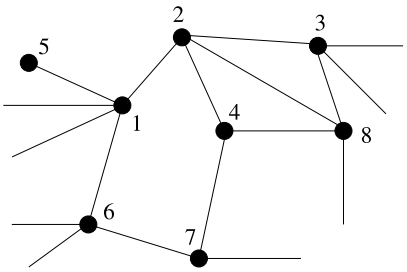
We next establish the correctness of FullyBalancedPartition(G_p). In the following, let constant b represent the maximum degree of mesh G.

Lemma 7 *Given a well shaped mesh G and a subgraph $G_p \subseteq G$,* FullyBalancedPartition *generates a fully-balanced partition of G_p.*

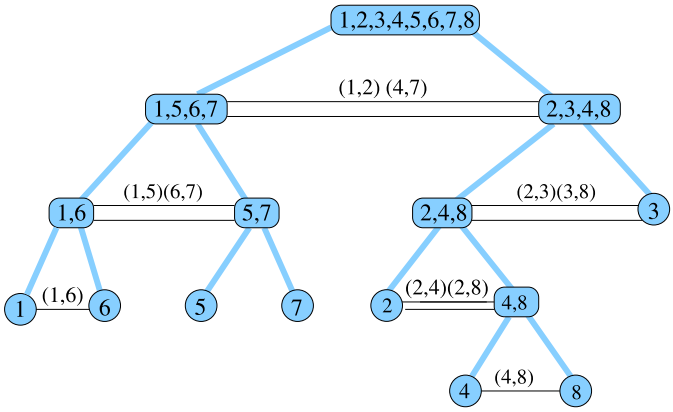
Proof By the way that we generate V_{px} and V_{py}, we have

$$||V_{py}| - |V_{px}|| \leq 1.$$

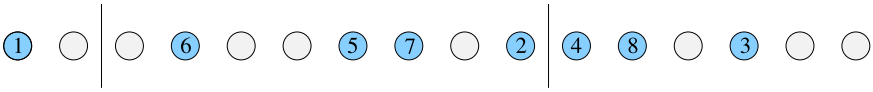
This is because the number of blue elements in the subarray is exactly $|V_{px}|$, and the number of blue elements within and without the subarray differ by at most one.



(**a**) An example subgraph G_p of mesh G. Subgraph G_p has eight vertices, ten edges, and eight outer edges (i.e., $|\mathsf{outer}(G_p)| = 8$)



(**b**) A decomposition tree of the subgraph G_p from (**a**). Building this decomposition tree is the first step for $\mathsf{FullyBalancedPartition}(G_p)$. The crossing edges at each node are indicated by lines between the two children. Thus, $\mathsf{crossing}((G_p)_0) = \{(1,5),(6,7)\}$ and $\mathsf{crossing}((G_p)_{101}) = \{(4,8)\}$. Observe that each edge in G_p is a crossing edge for exactly one node in the decomposition tree



(**c**) The red-blue array for G_p. The *blue* elements have a *dark shade*. The *red* elements have a *light shade*. There is one blue element for each vertex in G_p. There is one red element for each outgoing edge in G_p. Since element 1 is adjacent to two edges in $\mathsf{outer}(G_p)$, there are two red elements after it in the red-blue array. The figure indicates a subarray containing half of the blue elements and half of the red elements to within one. The red-blue array is used to make the fully-balanced partition of G_p. Specifically, G_{px} will contain vertices 2, 5, 6, and 7 and G_{py} will contain vertices 1, 3, 4, and 8. Partition G_{px} inherits three outer edges from G_p, and partition G_{py} inherits five outer edges from G_p. This particular subarray means that two paths in the decomposition tree will be cut. One path, separating element 1 from 6, goes from node $(G_p)_{00}$ to the root. The other path, separating element 2 from 4, goes from node $(G_p)_{10}$ to the root. The edges that are cut by this partition are the crossing edges of these nodes, i.e., $E(G_{px}, G_{py}) = \{(1,6),(1,5),(6,7),(1,2),(4,7),(2,3),(3,8),(2,4),(2,8)\}$. If G_p is a node in the fully-balanced decomposition tree, then its left child will be G_{px} and its right child will be G_{py}

Fig. 2 The steps of the algorithm $\mathsf{FullyBalancedPartition}(G_p)$ run on a sample graph

We next show that

$$||\mathsf{outgoing}(G_{py})| - |\mathsf{outgoing}(G_{px})|| \leq 2b + 1. \tag{1}$$

To determine $|\mathsf{outgoing}(G_{px})|$ and $|\mathsf{outgoing}(G_{py})|$, modify the subarray as follows. Remove from the subarray any red elements at the beginning of the subarray before the first blue element *in* the subarray. Then add to the subarray any red elements before the first blue element *after* the subarray. The number of red elements now in the subarray is $|\mathsf{outgoing}(G_{px})|$ and the number of red elements not in the subarray is $|\mathsf{outgoing}(G_{py})|$. This modification can only increase or decrease $|\mathsf{outgoing}(G_{px})|$ and $|\mathsf{outgoing}(G_{py})|$ each by b, establishing (1).

Now, following [7, 24], we show that

$$E(G_{px}, G_{py}) \leq c|V_p|^{\alpha}(1 + \beta^{\alpha})/(1 - \beta^{\alpha}). \tag{2}$$

By selecting a subarray of the red-blue array, we effectively make two cuts on the leaves of the decomposition tree T_{G_p}. (The only time when there is apparently a single cut is if the subarray is the first half of the array. In this case, the second cut separates the first leaf from the last.) Consider one of these cuts. The array is split between two consecutive leaves of T_{G_p}. Denote by P the root of the smallest subtree of T_{G_p} containing these two leaves; see Fig. 2(c). We consider the upward path $P, P_1, P_2, \ldots, G_p$ in the decomposition tree T_{G_p} from P up to the root G_p of T_{G_p}. Each node in the decomposition tree on this path is a subgraph of G that is being split into two pieces.

We now count the number of edges that get removed as a result of these splits:

$$\begin{aligned} &|\mathsf{crossing}(P) \cup \mathsf{crossing}(P_1) \cup \mathsf{crossing}(P_2) \cup \cdots \cup \mathsf{crossing}(G_p)| \\ &\leq \sum_{i=0}^{\log_{1/\beta}|V|} c\big(|V|\beta^i\big)^{\alpha} \\ &\leq c|V|^{\alpha}/(1 - \beta^{\alpha}). \end{aligned} \tag{3}$$

As reflected in (3), each node along the path has a different depth, which gives a geometric series.

The number of edges that cross from G_{px} to G_{py}, $E(G_{px}, G_{py})$, is the number of edges that get removed when both cuts get made. However, doubling (3) overestimates $E(G_{px}, G_{py})$ by an amount $|\mathsf{crossing}(G_p)|$ since the root G_p can only be cut once. Thus, doubling (3) and subtracting $|\mathsf{crossing}(G_p)|$, we establish (2). □

3.2 Fully-Balanced Decomposition Trees

A *fully-balanced decomposition tree* of a graph G is a decomposition tree of G where the partition of every node (subgraph) in the tree is fully-balanced.

We build a fully-balanced decomposition tree BT_G of G recursively. First we apply the algorithm FullyBalancedPartition on the root G to find the left and right children, G_0 and G_1. We next recursively build the fully balanced decomposition tree rooted at G_0 and the fully-balanced decomposition tree rooted at G_1.

Theorem 8 (Fully-Balanced Decomposition Tree for a Mesh) *A fully-balanced decomposition tree of a mesh G of constant dimension can be computed in time $O(|G|\log^2|G|)$ on a RAM both in expectation and with high probability. The fully-balanced decomposition tree can be computed in the DAM and cache-oblivious models using $O(1+(|G|/B)\log^2(|G|/M))$ memory transfers in expectation and $O(1+(|G|/B)(\log^2(|G|/M)+\log|G|))$ memory transfers with high probability.*

Proof We first establish that the construction algorithm takes expected time $O(|G|\log^2|G|)$ on a RAM. By Lemma 6, for any node G_p in the decomposition tree, we need $O(|G_p|\log|G_p|)$ operations to build the left and right children, G_{p0} and G_{p1}, both in expectation and with probability at least $1-1/\text{poly}(|G_p|)$. Since the left and right children, $|G_{p0}|$ and the $|G_{p1}|$, of every node G_p differ in size by at most 1, BT_G has $\Theta(\log|G|)$ levels. If $|G_p|$ denotes the size of a node at level i, then level i has construction time $O(|G|\log|G_p|)$. Thus, the construction-time bound follows by linearity of expectation.

We next establish that the construction algorithm uses $O(1+|G|\log^2(|G|/M)/B)$ expected memory transfers in the DAM and cache-oblivious models. Because we build the decomposition tree recursively, we give a recursive analysis. The base case is when a node G_p has size less than M while its parent node is greater than M. Then the cost to build the entire subtree T_{G_p} is only $O(M/B)$, because this is the cost to read all blocks of G_p into memory. Said differently, once a node is a constant fraction smaller than M, the cost to build the fully-balanced decomposition tree is 0 because all necessary memory blocks already reside in memory. There are therefore $\Theta(\log|G|-\log M)$ levels of the fully-balanced decomposition tree having nonzero construction cost. Each level uses at most $O((|G|/B)\log(|G|/M))$ memory transfers. Thus, the time bounds follows by linearity of expectation.

We next establish the high-probability bounds. In the following analysis, we examine, for each node G_p in the fully-balanced decomposition tree, the decomposition tree T_{G_p} that is used to build that node. We then group the nodes of all the decomposition trees by size and count the number of nodes in each group.

As an example, suppose that $|G|$ is a power of two and all splits are even. There is one node of size $|G|$—the root node of the decomposition tree T_G. There are four nodes of size $|G|/2$—two nodes in T_G, one node in T_{G_0}, and one node in T_{G_1}. There are 12 node of size $|G|/4$—four nodes in T_G, two nodes in T_{G_0}, two node in T_{G_1}, and one node in each of $T_{G_{00}}$, $T_{G_{01}}$, $T_{G_{10}}$, and $T_{G_{11}}$.

In general, let group i contain all decomposition tree nodes having size in the range $(|G|/2^i, |G|/2^{i-1}]$. Then group i contains $\Theta(i2^i)$ nodes.

Analyzing each group separately, we show that the construction algorithm takes time $O(|G|\log^2|G|)$ on a RAM with high probability. First, consider the $\Theta(\log|G|)$ largest nodes (those most expensive to build), i.e., those in the smallest cardinality groups. As analyzed in Theorem 4, building these nodes takes time $O(|G|\log|G|)$ with high probability.

We analyze the rest of the node constructions group by group. Since each group i contains $\Theta(i2^{i-1})$ nodes, each successive group contains more nodes than the total number of nodes in all smaller groups. As a result, there are $\Omega(\log|G|)$ nodes in

each of the rest of the groups. Thus, by Lemma 3, the time to build the rest of the tree with high probability is the same as the time in expectation, which is $O(|G| \log^2 |G|)$. Thus, we establish high-probability bounds on the running time.

We now show that the construction algorithm takes $O(1 + (|G|/B) \times (\log^2(|G|/M) + \log|G|))$ memory transfers with high probability. First consider the $\Theta(\log|G|)$ largest nodes (those most expensive to build). As analyzed in Theorem 4, building these nodes uses $O(1 + |G| \log|G|/B)$ memory transfers with high probability. Now examine all remaining nodes. We consider each level separately. Each group contains $\Omega(\log|G|)$ nodes. Thus, by Lemma 3, the high-probability cost of building the decomposition trees for all remaining nodes matches the expected cost, which is $O(1 + (|G|/B) \log^2(|G|/M))$ memory transfers. Thus, with high probability, the construction algorithm takes $O(1 + (|G|/B)(\log|G| + \log^2(|G|/M)))$ memory transfers with high probability, as promised. □

3.3 k-Way Partitions

We observe one additional benefit of Theorem 8. In addition to providing a simpler and faster algorithm for constructing fully-balanced decomposition trees, we also provide a new algorithm for k-way partitioning, as described in [23]. For any positive integer $k > 1$, a *k-way partition* of a graph $G = (V, E)$, is a k-tuple $(V_1, V_2, \ldots, V_k)$ (hence $(G_1, G_2, \ldots, G_k)$) such that $\bigcup_{1 \le i \le k} V_i = V$ and $V_i \cap V_j = \emptyset$ for $i \ne j, 1 \le i, j \le k$. For any $\beta \ge 1$, $(V_1, V_2, \ldots, V_k)$ is a (β, k)-way partition if $|G_i| \le \beta \lceil |G|/k \rceil$, for all $i \in \{1, \ldots, k\}$. It has been shown in [23] that every well shaped mesh in d dimensions has a $(1+\epsilon, k)$-way partition, for any $\epsilon > 0$, such that $\max_i\{\mathsf{outer}(G_i)\} = O((|G|/k)^{1-1/d})$.

We now describe our k-way partition algorithm of a well shaped mesh G. The objective is to evenly divide leaves of a fully-balanced decomposition tree of G into k parts such that their number of vertices are the same within one. First build a fully-balanced decomposition tree. Now assign the first $|V|/k$ leaves to V_1, the next $|V|/k$ leaves to V_2, and so on.

In fact, we can modify this approach so that it runs faster by observing that we need not build the complete fully-balanced decomposition tree. First build the top $\Theta(\log k)$ levels of the tree, so that there are $\mathrm{poly}(k)$ leaves. At most k of these leaves need to be refined further, since the remaining leaves will all belong to a single group V_i.

Our k-way partition algorithm using fully-balanced decomposition trees is incomparable to the algorithm of [23]. By building fully-balanced decomposition tree, even a partial one, our algorithm is slower than the algorithm of [23], which uses geometric separators for partitioning instead. On the other hand, it can be used to divide the nodes into k sets whose sizes are equal to within an additive one, instead of only asymptotically the same size as in [23].

4 Cache-Oblivious Layouts

In this section we show how to find a cache-oblivious layout of a mesh G. Given such a layout, we show that a mesh update runs asymptotically optimally in $\Theta(1 + |G|/B)$

memory transfers given the tall cache assumption that $M = \Omega(B^d)$. We also analyze the performance of a mesh update when $M = o(B^d)$, bounding the performance degradation for smaller M.

The layout algorithm is as follows.

CacheObliviousMeshLayout(G)

1. Build a $f(N) = O(N^{1-1/d})$ fully-balanced decomposition tree T_G of G, as described in Theorem 8.
2. Reorder the vertices in G according to the order of the leaves in T_G. (Recall that each leaf in T_G stores a single vertex in G.) This reorder means: (a) assign new indices to all vertices in the mesh, and (b) for each vertex, let all neighbor vertices know the new index.

We now describe the mechanics of relabeling and reordering. Each vertex knows its ordering and location in the input layout; this is the vertex's index. A vertex also knows the index of each of its neighboring vertices. When we change a vertex's index, we apprise all neighbor vertices of the change. These operations entail a small number of scans and cache-oblivious sorts [9, 11, 17, 30], for a total cost of $O((|G|/B)\log_{M/B}(|G|/B)$ memory transfers. This cost is dominated by the cost to build the fully-balanced decomposition tree. (Thus, a standard merge sort, which does not minimize the number of memory transfers, could also be used.)

The cleanest way to explain is through an example. Suppose that we have input graph $G = \{\{a, b, c, d\}, \{(a, c), (a, d), (b, c), (c, d)\}\}$, which is laid out in input order:

$$(a, c), (a, d), (b, c), (c, a), (c, b), (c, d), (d, a), (d, c).$$

Suppose that the leaves of fully-balanced decomposition tree are in the order of a, c, d, b. This means that the renaming of nodes is as follows: $[a : 1], [c : 2]$, $[d : 3], [b : 4]$. (For clarity, we change indices from letters to numbers.) We obtain the reverse mapping $[a : 1], [b : 4], [c : 2], [d : 3]$ by sorting cache-obliviously. We change the labels on the first component of the edges by array scans:

$$(a = 1, c), (a = 1, d), (b = 4, c), (c = 2, a), (c = 2, b), (c = 2, d), (d = 3, a),$$
$$(d = 3, c).$$

We then sort the edges by the second component,

$$(c = 2, a), (d = 3, a), (c = 2, b), (a = 1, c), (b = 4, c), (d = 3, c), (a = 1, d),$$
$$(c = 2, d),$$

and change the labels on the second component of the edge by another scan:

$$(c = 2, a = 1), (d = 3, a = 1), (c = 2, b = 4), (a = 1, c = 2), (b = 4, c = 2),$$
$$(d = 3, c = 2), (a = 1, d = 3), (c = 2, d = 3).$$

We get

$$(2,1),(3,1),(2,4),(1,2),(4,2),(3,2),(1,3),(2,3).$$

We sort these edges by the first component to get the final layout. The final layout is

$$(1,2),(1,3),(2,1),(2,3),(2,4),(3,1),(3,2),(4,2).$$

Thus, we obtain the following layout performance:

Theorem 9 *A cache-oblivious layout of a well shaped mesh G can be computed in $O(|G|\log^2|G|)$ time both in expectation and with high probability. The cache-oblivious layout algorithm uses $O(1+|G|\log^2(|G|/M)/B)$ memory transfers in expectation and $O(1+(|G|/B)(\log^2(|G|/M)+\log|G|))$ memory transfers with high probability.*

With such a layout, we can perform a mesh update cache-obliviously.

Theorem 10 *Every well shaped mesh G in d dimensions has a layout that allows the mesh to be updated cache-obliviously with $O(1+|G|/B)$ memory transfers on a system with block size B and cache size $M=\Omega(B^d)$.*

Proof We apply the algorithm described above on G to build the layout. Since each vertex of G has constant degree bound b, its size is bounded by a constant. Consider a row of nodes $G_{p_1}, G_{p_2}, G_{p_3}, \ldots$ in T_G at a level such that each node G_{p_i} uses $\Theta(M) < M$ space and therefore fits in a constant fraction of memory.

In the mesh update, the nodes of G are updated in the order of the layout, which means that first the vertices in G_{p_1} are updated, then vertices of G_{p_2}, then vertices of G_{p_3}, etc. To update vertices of G_{p_i}, the vertices must be brought into memory, which uses at most $O(1+M/B)$ memory transfers. In the mesh update, when we update a vertex u, we access u's neighbors. If the neighbor v of u is also in G_{p_i}, i.e., edge (u,v) is internal to G_{p_i}, then accessing this neighbor uses no extra memory transfers. On the other hand, if the neighbor v is not in G_{p_i}, then following this edge requires another transfer hence an extra block to be read into memory.

We now show that $\mathsf{outer}(G_{p_i}) = O(|G_{p_i}|^{1-1/d})$. Since all subgraphs at the same level of the fully-balanced decomposition tree are of the same size within one, and outgoing edges of any subgraph are evenly split, each G_{p_i} has roughly the same number of outer edges. Suppose G_{p_i} is in level j. The total number of their outer edges are at most

$$\begin{aligned}&|G|^{1-1/d}+2\left(\frac{|G|}{2}\right)^{1-1/d}+4\left(\frac{|G|}{4}\right)^{1-1/d}+\cdots+2^j\left(\frac{|G|}{2^j}\right)^{1-1/d}\\&\quad\le\left(\frac{2^j}{2^{1/d}-1}\right)\left(\frac{|G|}{2^j}\right)^{1-1/d}.\end{aligned}$$

Hence, $\mathsf{outer}(G_{p_i}) = O((|G|/2^j)^{1-1/d}) = O(|G_{p_i}|^{1-1/d}) = O(M^{1-1/d})$. Therefore the total size of memory that we need to perform a mesh update of the vertices in G_{p_i} is $\Theta(M+BM^{1-1/d})$.

By the tall-cache assumption that $B^d \leq M$, i.e., $B \leq M^{1/d}$, and for a proper choice of constants, the mesh update for G_{p_i} only uses $\Theta(M) < M$ memory. Since updating each node G_{p_i} of size $\Theta(M)$ uses $O(1 + M/B)$ memory transfers, and there are a total of $O(|G|/M)$ such nodes, the update cost is $O(1 + |G|/B)$, which matches the scan bound of G, and is optimal. □

Thus, for dimension $d = 2$, we have the "standard" tall-cache assumption [17], and for higher dimensions we have a more restrictive tall-cache assumption. We now analyze the tradeoff between cache height and complexity. Suppose instead of a cache with $M = \Omega(B^d)$, the cache is of $M = \Omega(B^{d-\epsilon})$. We assume $\epsilon < d - 1$. We show that the cache performance of mesh update is $B^{\epsilon/d}$ away from optimal.

Corollary 11 *Every well shaped mesh G in d dimensions has a vertex ordering that allows the mesh to be updated cache-obliviously with $O(1 + |G|/B^{1-\epsilon/d})$ memory transfers on a system with block size B and cache size $M = \Omega(B^{d-\epsilon})$.*

Proof We apply similar analysis to that in Theorem 10 on G. From Theorem 10, the total size of memory that we need to update mesh G_{p_i} is $\Theta(M + BM^{1-1/d})$. Since $M = \Omega(B^{d-\epsilon})$, we have

$$\begin{aligned} O(M + BM^{1-1/d}) &= O\left(M + M\frac{B}{M^{1/d}}\right) \\ &\leq O\left(M + M\frac{B}{B^{1-\epsilon/d}}\right) \\ &= O(M + MB^{\epsilon/d}). \end{aligned}$$

Thus, updating G_{p_i} uses $O(1 + (M + MB^{\epsilon/d})/B)$ memory transfers, which simplifies to $O(1 + |G|/B^{1-\epsilon/d})$ memory transfers. □

5 Relax-Balanced Decomposition Trees and Faster Cache-Oblivious Layouts

In this section we give the main result of this paper, a faster algorithm for finding a cache-oblivious mesh layout of a well-shaped mesh. The main idea of the algorithm is to construct a new type of decomposition tree, which we call a *relax-balanced decomposition tree*. The relax-balanced decomposition tree is based on what we call a *relax-balanced partition*. We give an algorithm for building an relax-balanced decomposition tree whose performance is nearly a logarithmic factor faster than the algorithm for building a fully-balanced decomposition tree. We prove that an asymptotically optimal cache-oblivious mesh layout can be found by traversing the leaves of the relax-balanced decomposition tree.

5.1 Relax-Balanced Partitions

We first define the relax-balanced partition of a subgraph G_p of G. A *relax-balanced f-partition* of $G_p \subseteq G$ is a partitioning of G_p into two subgraphs G_{p0} and G_{p1} such that

- $|\mathsf{crossing}(G_p)| \leq f(|G_p|)$,
- $|G_{p0}| = |G_{p1}| \pm O(1 + |G_p|/\log^3 |G|)$, and
- $|\mathsf{outgoing}(G_{p0})| = |\mathsf{outgoing}(G_{p1})| \pm O(1 + |\mathsf{outgoing}(G_{p1})|/\log^2 |G|)$.

We next present an algorithm, RelaxBalancedPartition, for computing balanced partitions. Given $G_p \subseteq G$, and a $(f(N) = O(N^\alpha), \beta)$-partitioning geometric separator, RelaxBalancedPartition(G_p) computes a relax-balanced $(f(N) = O(N^\alpha))$-partition G_{px} and G_{py} of G_p.

We find the relax-balanced partition by building what we call a *relax partition tree* T_{G_p}. We call the top $3\log_{1/\beta}\log|G|$ levels of T_{G_p} the *upper tree* of T_{G_p} and the remaining levels the *lower tree* of T_{G_p}.

We build the upper tree by building the top $3\log_{1/\beta}\log|G|$ levels of a decomposition tree of G_p. By construction, all leaves of the upper tree (subgraphs of G_p) contain at most $|G_p|/\log^3|G|$ vertices. Outer edges of G_p are distributed among these leaves. By a counting argument, there are at most $\log^2|G|$ leaves that can contain more than $|\mathsf{outer}(G_p)|/\log^2|G|$ outer edges of G_p.

For each upper-tree leaf having more than $|\mathsf{outer}(G_p)|/\log^2|G|$ outer edges, we refine the leaf by building a decomposition tree on it. We do not refine the other leaves of the upper tree. The union of these decomposition trees comprise the lower tree.

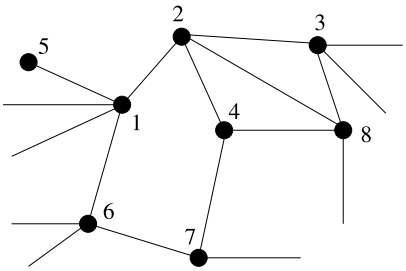
Relax partition tree T_{G_p} has leaves at different depths. Some leaves are subgraphs having a single vertex while others may have up to $|G|/\log^3|G|$ vertices. The tree is stored in the same format as a standard decomposition tree. Thus, leaves of the relax partition tree that are not refined contain vertices stored in an arbitrary order. The relax partition tree T_{G_p} of G_p is just a decomposition tree if there are fewer than $\log^3|G|$ vertices.

RelaxBalancedPartition(G_p)

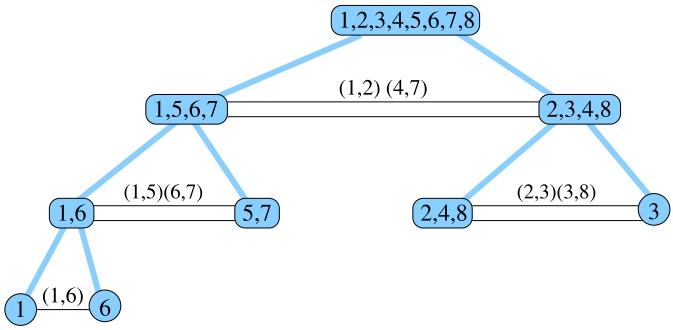
1. *Build* T_{G_p}—Build the relax partition tree T_{G_p} from G_p recursively.
2. *Build red-blue array*—Build an array of vertices by an in-order traversal of leaves of T_{G_p}. Vertices in leaves that are not refined are laid out arbitrarily. Build a red-blue array and find a subarray in the red-blue array as described in FullyBalancedPartition.
3. *Modify red-blue array* – Modify the subarray to satisfy the following constraint. All vertices in an (unrefined) leaf must stay together, either within or without the subarray. If any cut separates the vertices, then move the cut leftward or rightward to be in between the leaf node and a neighbor. Now partition the vertices in G_p based on this modified subarray. Put the vertices representing blue elements that are in the subarray into set V_{px} and put the vertices representing blue elements that are outside of the subarray into set V_{py}.
4. *Partition* G_p—Compute G_{px} and G_{py} from V_{px} and V_{py}. This computation also means scanning edges to determine which edges are internal to G_{px} and G_{py} and which have now become external.

We first establish the running time of RelaxBalancedPartition(G_p).

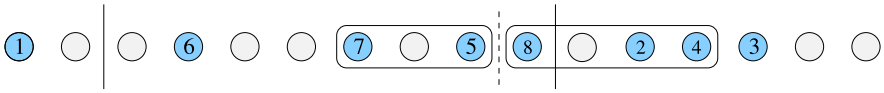
Lemma 12 *Given a subgraph G_p of a well shaped mesh G, $|G_p| \geq \log^3|G|$, RelaxBalancedPartition(G_p) runs in time $O(|G_p|\log\log|G|)$ on a RAM and*



(**a**) An example subgraph G_p of mesh G. Subgraph G_p has eight vertices, ten edges, and eight outer edges (i.e., $|\mathsf{outer}(G_p)| = 8$)



(**b**) A relax partition tree of the subgraph G_p from (**a**). Building this decomposition tree is the first step for RelaxBalancedPartition(G_p). Observe that each edge in G_p is a crossing edge for at most one node in the decomposition tree. Some edges, such as $(2, 4)$, are not crossing edges for any node. The top three levels of the decomposition tree are the upper tree. We refine a leaf of the upper tree if only it has many (at least three) edges from outer(G_p). Upper tree leaf $(G_p)_{00}$ has 4 edges from outer(G_p). Upper tree leaf $(G_p)_{01}$ has 1 edge from outer(G_p). Upper tree leaf $(G_p)_{10}$ has 1 edge from outer(G_p). Upper tree leaf $(G_p)_{11}$ has 2 edges from outer(G_p). Thus, only $(G_p)_{00}$ is further refined



(**c**) The red-blue array for G_p. The *blue* elements have a *dark shade*. The *red* elements have a *light shade*. There is one blue element for each vertex in G_p. There is one red element for each outgoing edge in G_p. The figure indicates a subarray containing half of the blue elements and half of the red elements to within one. However, this subarray separates element 8 from element 2. This cut is not allowed because 8 and 2 are in the same leaf of the relax partition tree. Instead the cut is moved left to the first valid position. The new cut separates element 5 from element 8, which is allowed because 5 and 8 are in different leaves of the relax partition tree. Thus, G_{px} will contain vertices 5, 6, and 7, and G_{py} will contain vertices 1, 2, 3, 4, and 8

Fig. 3 The steps of the algorithm RelaxBalancedPartition(G_p) run on a sample graph

$O(1 + (|G_p|/B)\min\{\log\log|G|, \log(|G_p|/M)\})$ *memory transfers in the DAM and cache-oblivious models in expectation. With high probability, it runs in* $O(|G_p|\log|G_p|)$ *on a RAM and* $O(1 + |G_p|\log|G_p|/B)$ *memory transfers in the DAM and cache-oblivious models.*

Proof We establish that the construction algorithm runs in expected time $O(|G_p|\log\log|G|)$ on a RAM. The upper tree of T_{G_p} takes expected time $O(|G_p|\log\log|G|)$. There are at most $\log^2|G|$ leaves of the upper tree to be refined. For each of these leaves, we build a decomposition tree, and this takes expected time

$$O((|G_p|/\log^3|G|)\log(|G_p|/\log^3|G|)) \leq O(|G_p|/\log^2|G|).$$

Thus, the total expected time to refine all leaves is $O(|G|)$. Steps 2–4 takes linear time. Thus, RelaxBalancedPartition(G_p) finds a relax-balanced partition in expected time $O(|G_p|\log\log|G|)$.

We next establish that the construction algorithm uses $O(1 + (|G_p|/B)\min\{\log\log|G|, \log(|G_p|/M)\})$ expected memory transfers in the DAM and cache-oblivious models. There are two cases. The first case is when $M \geq |G_p|/\log^3|G|$. Then some of nodes in the top $3\log_{1/\beta}\log|G|$ levels of the T_{G_p} may be a constant fraction smaller than M. Such small nodes require no memory transfers to build, because they are already stored in memory. Only the top $O(\log(|G_p|/M))$ levels use memory transfers. The rest of the decompositions are free of memory transfers because all necessary memory blocks already reside in memory. When a subgraph G_p has size $\Omega(M)$, then the partition of a subgraph takes expected $\Theta(|G_p|/B)$ memory transfers, because this is the cost of a linear scan. Hence, the total cost is $O(1 + (|G_p|/B)\log(|G_p|/M))$.

The second case is when $M < |G_p|/\log^3|G|$. Then, the upper tree of T_{G_p} takes $O(1 + |G_p|\log\log|G|/B)$ memory transfers in expectation. There are at most $\log^2|G|$ leaves of the upper tree of T_{G_p} that need further refinement, and the leaf sizes are at most $|G_p|/\log^3|G|$. Building a decomposition tree on one of these leaves takes

$$O(1 + (|G_p|/\log^3|G|)\log(|G_p|/\log^3|G|)/B) \leq O(1 + |G_p|/B\log^2|G|)$$

memory transfers in expectation. Since there are at most $\log^2|G|$ leaves, the total expected number of memory transfers to construct the lower tree of T_{G_p} is $O(|G_p|/B)$, which is dominated by the cost to build the upper tree.

Combining the two cases, we obtain that the expected number of memory transfers to build T_{G_p} is $O(1 + (|G_p|/B)\min\{\log\log G, \log(|G_p|/M)\})$.

We next establish the high-probability bounds. We first consider all nodes that have size $\Omega(|G_p|/\log|G_p|)$. There are $O(\log|G_p|)$ such nodes. Building these nodes uses time $O(|G_p|\log|G_p|)$ and $O(1 + |G_p|\log|G_p|/B)$ memory transfers with high probability by Theorem 4.

For the rest of the *upper tree* of T_{G_p}, each level contains $\Omega(\log|G_p|)$ nodes. Thus, the number of memory transfers with high probability matches the number of memory transfers in expectation, which is $O(1 + (|G_p|/B)\min\{\log\log G, \log(|G_p|/M)\})$. The cost to build the rest of the upper tree is dominated by the cost to build the largest $O(\log|G_p|)$ nodes in the upper tree.

As described above, the expected cost to build the lower tree is $O(|G_p|)$ time and $O(|G_p|/B)$ memory transfers. The high-probability bounds are at most a $O(\log |G_p|)$ factor greater and hence are dominated by the cost to build the upper tree. Thus, we establish the high probability bounds on time and memory transfers. □

We next establish the correctness of RelaxBalancedPartition(G_p). In the following, let b represent the maximum degree of mesh G.

Lemma 13 *Given a well shaped mesh G and a subgraph $G_p \subseteq G$,* RelaxBalancedPartition(G_p) *generates a relax-balanced partition of G_p.*

Proof By the way we construct the relax partition tree T_{G_p}, nodes that are not refined contain $O(|\mathsf{outer}(G_p)|/\log^2|G|)$ outer edges of G_p, and their sizes differ by $O(|G_p|/\log^3|G|)$. Thus, by the way we generate G_{px} and G_{py}, the number of outgoing edges of G_{px} and G_{py} differ by $O(|\mathsf{outer}(G_p)|/\log^2|G|)$ and $|G_{px}|$ and $|G_{py}|$ differ by $O(|G_p|/\log^3|G|)$. Recall that $\mathsf{outgoing}(G_{px}) \cup \mathsf{outgoing}(G_{py}) = \mathsf{outer}(G_p)$. Thus, we have

$$|\mathsf{outgoing}(G_{px})| = |\mathsf{outgoing}(G_{py})| \pm O(|\mathsf{outgoing}(G_{py})|/\log^2|G|).$$

As shown in (2) from Lemma 7, the number of crossing edges satisfies $|\mathsf{crossing}(G_p)| \leq f(|G_p|)$. □

5.2 Relax-Balanced Decomposition Trees

A *relax-balanced decomposition tree* of a well shaped mesh G is a decomposition tree of G where every partition of every node G_p in the tree is relax-balanced.

We construct a relax-balanced decomposition tree of G recursively. First we apply the algorithm RelaxBalancedPartition on the root G to get the left and right children, G_0 and G_1. We next recursively build the (left) subtree rooted at G_0 and then the (right) subtree rooted at G_1.

Theorem 14 (Relax-Balanced Decomposition Tree for a Mesh) *A relax-balanced decomposition tree of a well shaped mesh G of constant dimension can be computed in time $O(|G|\log|G|\log\log|G|)$ on a RAM both in expectation and with high probability. The relax-balanced decomposition tree can be computed in the DAM and cache-oblivious models using $O(1+(|G|/B)\log(|G|/M)\min\{\log\log|G|, \log(|G|/M)\})$ memory transfers in expectation and $O(1+(|G|/B)(\log(|G|/M)\min\{\log\log|G|, \log(|G|/M)\}+\log|G|))$ memory transfers with high probability.*

Proof When $|G| \leq M$, the construction algorithm takes $O(|G|)$ time and $O(|G|/B)$ memory transfers, both in expectation and with high probability. We consider $O(|G|) = \Omega(M)$ in the following analysis.

We first analyze the expected running time of the algorithm on a RAM. The construction time of each node G_p is $O(|G_p| \log\log |G|)$, and there are $O(\log |G|)$ levels in the relax-balanced decomposition tree. Thus, by linearity of expectation, the expected running time is $O(|G| \log |G| \log\log |G|)$.

We show that the construction algorithm uses $O(1 + (|G|/B)\log(|G|/M)\min\{\log\log |G|, \log(|G|/M)\})$ memory transfers in the DAM and cache-oblivious models. We analyze large and small nodes in the relax-balanced decomposition tree differently. There are two cases. The first case is when a tree node G_p is large, i.e., $|G_p| \geq \log^3 |G|$. In this case, RelaxBalancedPartition(G_p) uses expected $O(1 + (|G_p|/B)\min\{\log\log |G|, \log(|G_p|/M)\})$ memory transfers by Lemma 12. Since all nodes a constant factor smaller than M can be constructed with no memory transfers, we only need consider nodes of size $\Omega(M)$. There are $O(\log(|G|/M))$ levels of nodes of size $\Omega(M)$. So the construction of all nodes larger than $\log^3 |G|$ takes $O(1 + (|G|/B)\log(|G|/M)\min\{\log\log |G|, \log(|G|/M)\})$ expected memory transfers.

The second case is when $|G_p| < \log^3 |G|$. In this case, we build a complete decomposition tree at each node. Therefore by Lemma 6, the cost to build one of these nodes is $O(1 + (|G_p|/B)\log(|G_p|/M))$ in expectation. As before, nodes a constant factor smaller than M can be constructed with no memory transfers. Therefore, the number of levels containing nodes of size between $\Omega(M)$ and less than $\log^3 |G|$ is at most $O(\log(\log^3 |G|/M))$. Thus, the construction of all nodes of size $O(\log^3 |G|)$ uses $O(1 + (|G|/B)\log^2(\log^3 |G|/M))$ memory transfers in expectation, which is dominated by the first case.

Now we establish the high probability bounds. We analyze the largest $\Theta(\log |G|)$ nodes and the remaining nodes of the relax-balanced decomposition tree separately. Any level of the relax-balanced decomposition tree below the largest $\Theta(\log |G|)$ nodes has $\Omega(\log |G|)$ nodes. Hence, for each level, the construction cost with high probability matches the construction cost in expectation, which is $O(|G| \log\log |G|)$ expected time and $O(1 + (|G|/B)\min\{\log\log |G|, \log(|G|/M)\})$ expected memory transfers. Since the construction algorithm is recursive, a relax-balanced partition of nodes a constant fraction smaller than M uses no memory transfers. Hence, all levels of the relax-balanced decomposition tree other than the largest $\Theta(\log |G|)$ nodes can be constructed in $O(|G| \log |G| \log\log |G|)$ time in a RAM and $O(1 + (|G|/B)\log(|G|/M)\min\{\log\log |G|, \log(|G|/M)\})$ memory transfers with high probability.

For the largest $\Theta(\log |G|)$ nodes of the relax-balanced decomposition tree, we establish the high probability bounds using a different approach. Similar to the proof of Theorem 8, we examine each relax partition tree that is used to build each node of the relax-balanced decomposition tree, and we examine all nodes within all of these relax partition trees. However, now there are upper trees and lower trees; we examine the nodes within upper and lower trees separately.

We look at the upper trees of the relax partition trees of the largest $\Theta(\log |G|)$ nodes of the relax-balanced decomposition tree. There are $\Theta(\log |G|)$ upper trees, which are complete binary trees. Following a similar analysis to that in the proof of Theorem 8, the construction of the largest $\Theta(\log |G|)$ nodes from among the $\Theta(\log |G|)$ upper trees takes $O(|G| \log |G|)$ time and uses $O(1 + |G| \log |G|/B)$ memory transfers with high probability.

For the rest of the nodes in the upper trees, the high probability bounds match the expectation bounds, both in time and memory transfers by Theorem 8. Therefore building the nodes in the rest of the upper trees takes $O(|G|\log^2\log|G|)$ time and uses $O(|G|\log^2\log|G|/B)$ memory transfers with high probability. This cost is dominated by the construction cost of the largest $\Theta(\log|G|)$ nodes of the upper trees.

We now focus on the lower trees of the relax partition trees of the largest $\Theta(\log|G|)$ nodes of the relax-balanced decomposition tree. We show that the cost to build all of the lower trees takes time $O(|G|\log|G|)$ and uses $O(1+|G|\log|G|/B)$ memory transfers with high probability (i.e., probability $1-1/\text{poly}(|G|)$). With high probability, the lower tree of a partition tree T_{G_p} of a subgraph G_p can be computed in $O(|G_p|)$ on a RAM and with $O(|G_p|/B)$ memory transfers in the DAM and the cache-oblivious models. Given a node G_p and its relax partition tree T_{G_p}, there are two cases. The first case is when there are $\Omega(\log|G|)$ leaves of the upper tree of T_{G_p} that need to be refined. Thus, with high probability, the construction cost of the lower tree of T_{G_p} matches the expected construction cost, which is in $O(|G_p|)$ time and $O(|G_p|/B)$ memory transfers, as analyzed in Lemma 12.

The second case is when there are $O(\log|G|)$ leaves of the upper tree of T_{G_p} that need to be refined. The construction cost of a single leaf is $O(|G_p|/\log^2|G|)$ time and $O(|G_p|/B\log^2|G|)$ memory transfers in expectation. Thus, the construction cost to refine a single leaf with high probability is $O(|G_p|/\log|G|)$ time and $O(|G_p|/B\log|G|)$ memory transfers and the construction cost to refine all leaves with high probability is $O(|G_p|)$ time and $O(|G_p|/B)$ memory transfers. Thus, all lower trees of the relax partition trees of the largest $\Theta(\log|G|)$ nodes of the relax-balanced decomposition tree can be constructed in $O(|G|)$ time and $O(|G|/B)$ memory transfers with high probability, which is dominated by the construction of all upper trees.

Hence, with high probability, the construction algorithm runs in $O(|G|\log|G|\log\log|G|)$ time on a RAM and uses $O(1+(|G|/B)(\log(|G|/M)\times\min\{\log\log|G|,\log(|G|/M)\}+\log|G|))$ memory transfers in the DAM and the cache-oblivious models. □

We now show that a relax-balanced decomposition tree can serve the same purpose as a fully-balanced decomposition tree in giving cache-oblivious layout. The crucial property is the following.

Lemma 15 *Given a relax-balanced decomposition tree of graph G, all nodes on any level of the relax-balanced decomposition tree contain the same number of vertices to within an $o(1)$ factor and all outgoing degrees are the same size to within an $o(1)$ factor.*

Proof From the definition of relax-balanced, for any subgraph G_p and its two children G_{p_0} and G_{p_1} $|\mathsf{outgoing}(G_{p0})| = |\mathsf{outgoing}(G_{p1})| \pm O(|\mathsf{outgoing}(G_{p1})|/\log^2|G|)$, and $|G_{p0}| = |G_{p1}| \pm O(|G_p|/\log^3|G|)$. Thus, for constant c, the ratio of the outgoing degree or the size between any two subgraphs at depth i is at most $(1+c/\log^2|G|)^i$ and $(1+c/\log^3|G|)^i$. Since there are $O(\log|G|)$ levels, these quantities differ by at most an $o(1)$ factor, as promised. □

Similar to Sect. 4, to find a cache-oblivious layout of a well shaped mesh G, we build a relax-balanced decomposition tree of G. The in-order traversal of the leaves gives the cache-oblivious layout. Lemma 15 guarantees that we can apply the same analysis from Sect. 4 to show that we have a cache-oblivious layout.

We thus obtain the following result:

Theorem 16 *A cache-oblivious layout of a well shaped mesh G can be computed in time $O(|G|\log|G|\log\log|G|)$ on a RAM both in expectation and with high probability. The cache-oblivious layout can be computed in the DAM and cache-oblivious models using $O(1+(|G|/B)\log(|G|/M)\min\{\log\log|G|,\log(|G|/M)\})$ memory transfers in expectation and $O(1+(|G|/B)(\log(|G|/M)\min\{\log\log|G|,\log(|G|/M)\}+\log G))$ memory transfers with high probability.*

6 Applications and Related Work

6.1 Applications of Mesh Update

The mesh update problem appears in many scientific computations and ranks among most basic primitives for numerical computations. In finite-element and finite-difference methods, one must solve very large-scale sparse linear systems whose underlying matrix structures are meshes [27]. In practice, these linear systems are solved by conjugate gradient or preconditioned conjugate gradient methods [15, 31]. The most computational intensive operation of conjugate gradient is a matrix-vector multiplication operation [6, 15, 36, 37] which amounts to a mesh update in finite-element applications. The iterative conjugate gradient method repeatedly performs mesh updates. Mesh update is also the key operation in fast multipole methods (FMM) for N-body simulation [19, 20], especially when particles are not uniformly distributed [32]. The partitioning and layout techniques presented here also apply to the adaptive quadtrees or octtrees used in non-uniform N-body simulation.

6.2 Related Work

The cache-oblivious memory model was introduced in [17, 30], and cache-oblivious algorithms have been developed for many scientific problems such as matrix multiplication, FFT, and LU decomposition [8, 17, 30, 35]. Now the area of cache-oblivious data structures and algorithms is a lively field.

There are other approaches to achieving good locality in scientific computations. One alternative to the cache-oblivious approach is to write self-tuning programs, which measure the memory system and adjust their behavior accordingly. Examples include scientific applications such as FFTW [16], ATLAS [39], and self-tuning databases (e.g., [38]). The self-tuning approach can be complementary to the cache-oblivious approach. For example, some versions of FFTW [16] begin optimization starting from a cache-oblivious algorithm.

Methods exploiting locality for both sequential (out-of-core) and parallel implementation of iterative methods for sparse linear systems have long history in scientific

computing. Various partitioning algorithms have been developed for load balancing and locality on parallel machines [21, 22, 27, 31], and algorithms that have good temporal locality have been proposed and implemented for the out-of-core sparse linear solvers [34]. A mesh update can be viewed as a sparse matrix-dense vector multiplication, and there exist upper and lower bounds on the I/O complexity of this primitive [6]. However, these bounds apply to any type of matrix, whereas special structure of well-shaped meshes enables more efficient mesh updates.

Since the mesh-update problem is reminiscent of graph traversal, we briefly summarize a few results in external-memory graph traversal. The earliest papers in this area apply to general directed graphs [1, 12, 13, 29] and others focus on more specialized graphs, such as planar directed graphs [3] or undirected graphs perhaps of bounded degree [4, 14, 25, 26, 28]. The problem of cache-oblivious graph traversal and related problems is addressed by [5, 10]. There are also external-memory and cache-oblivious algorithms for other common graph problems, but such citations are beyond the scope of this paper.

The problem of cache-oblivious mesh layouts is first described in [40]. This paper gives no theoretical guarantees either on the traversal cost or the cost to generate the mesh layout, however. It does propose heuristics for mesh layout that give good running times, in practice, for a range of types of mesh traversals.

References

1. Abello, J., Buchsbaum, A.L., Westbrook, J.: A functional approach to external graph algorithms. In: Proc. of the 6th Annual European Symposium on Algorithms (ESA), pp. 332–343 (1998)
2. Aggarwal, A., Vitter, J.S.: The input/output complexity of sorting and related problems. Commun. ACM **31**(9), 1116–1127 (1988)
3. Arge, L., Brodal, G.S., Toma, L.: On external-memory MST, SSSP, and multi-way planar graph separation. In: Proc. of the 7th Scandinavian Workshop on Algorithm Theory (SWAT), pp. 433–447. Springer, Berlin (2000)
4. Arge, L., Meyer, U., Toma, L.: External memory algorithms for diameter and all-pair shortest-paths on sparse graphs. In: Proc. of 31st International Colloquium on Automata Languages and Programming (ICALP), pp. 146–157 (2004)
5. Arge, L., Bender, M.A., Demaine, E.D., Holland-Minkley, B., Munro, J.I.: An optimal cache-oblivious priority queue and its application to graph algorithms. SIAM J. Comput. **36**(6), 1672–1695 (2007)
6. Bender, M.A., Brodal, G.S., Fagerberg, R., Jacob, R., Vicari, E.: Optimal sparse matrix dense vector multiplication in the I/O-model. In: Proceedings of the 19th Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA), pp. 61–70 (2007)
7. Bhatt, S.N., Leighton, F.T.: A framework for solving VLSI graph layout problems. J. Comput. Syst. Sci. **28**(2), 300–343 (1984)
8. Blumofe, R.D., Joerg, C.F., Kuszmaul, B.C., Leiserson, C.E., Randall, K.H., Zhou, Y.: Cilk: an efficient multithreaded runtime system. J. Parallel Distrib. Comput. **37**(1), 55–69 (1996)
9. Brodal, G.S., Fagerberg, R.: Cache oblivious distribution sweeping. In: Proc. of 29th International Colloquium on Automata Languages and Programming (ICALP). LNCS, vol. 2380, pp. 426–438. Springer, Berlin (2002)
10. Brodal, G.S., Fagerberg, R., Meyer, U., Zeh, N.: Cache-oblivious data structures and algorithms for undirected breadth-first search and shortest paths. In: Proc. of the 9th Scandinavian Workshop on Algorithm Theory (SWAT), vol. 3111, pp. 480–492 (2004)
11. Brodal, G.S., Fagerberg, R., Vinther, K.: Engineering a cache-oblivious sorting algorithm. ACM J. Exp. Algorithmics **12**, 1–23 (2008)

12. Buchsbaum, A.L., Goldwasser, M., Venkatasubramanian, S., Westbrook, J.R.: On external memory graph traversal. In: Proc. of the 11th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA), pp. 859–860 (2000)
13. Chiang, Y.-J., Goodrich, M.T., Grove, E.F., Tamassia, R., Vengroff, D.E., Vitter, J.S.: External-memory graph algorithms. In: Proc. of the 6th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA), pp. 139–149 (1995)
14. Chowdhury, R.A., Ramachandran, V.: External-memory exact and approximate all-pairs shortest-paths in undirected graphs. In: Proc. of the 16th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA), pp. 735–744 (2005)
15. Demmel, J., Dongarra, J., Eijkhout, V., Fuentes, E., Antoine Petitet, R.V., Whaley, R.C., Yelick, K.: Self-adapting linear algebra algorithms and software. Proc. IEEE **93**(2), 293–312 (2005). Special Issue on Program Generation, Optimization, and Adaptation
16. Frigo, M., Johnson, S.G.: F.F.T.W.: an adaptive software architecture for the FFT. In: Proceedings of the Acoustics, Speech, and Signal Processing, vol. 3, pp. 1381–1384. IEEE Press, New York (1998)
17. Frigo, M., Leiserson, C.E., Prokop, H., Ramachandran, S.: Cache-oblivious algorithms. In: Proc. of the 40th IEEE Annual Symp. on Foundation of Computer Science (FOCS), pp. 285–297 (1999)
18. Goldberg, C., West, D.: Bisection of circle colorings. SIAM J. Algebr. Discrete Methods **6**(1), 93–106 (1985)
19. Greengard, L., Rokhlin, V.: A fast algorithm for particle simulations. J. Comput. Phys. **73**(2), 325–348 (1997)
20. Hackney, R.W., Eastwood, J.W.: Computer Simulation Using Particles. McGraw Hill, New York (1981)
21. Hendrickson, B., Leland, R.: The Chaco user's guide—version 2.0. Technical Report SAND94-2692, Sandia National Laboratories (1994)
22. Karypis, G., Kumar, V.: A fast and high quality multilevel scheme for partitioning irregular graphs. SIAM J. Sci. Comput. **20**(1), 359–392 (1998)
23. Kiwi, M., Spielman, D.A., Teng, S.-H.: Min-max-boundary domain decomposition. In: Theoretical Computer Science, vol. 261, pp. 253–266 (2001)
24. Leighton, F.T.: A layout strategy for VLSI which is provably good. In: Proc. of the 14th Ann. ACM Symp. on Theory of Computing (STOC), pp. 85–98 (1982)
25. Mehlhorn, K., Meyer, U.: External-memory breadth-first search with sublinear I/O. In: Proc. of the 10th Annual European Symposium on Algorithms (ESA), pp. 723–735. Springer, Berlin (2002)
26. Meyer, U.: External memory BFS on undirected graphs with bounded degree. In: Proc. of the 12th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA), pp. 87–88 (2001)
27. Miller, G.L., Teng, S.-H., Thurston, W., Vavasis, S.: Geometric separators for finite element meshes. SIAM J. Sci. Comput. **19**, 364–386 (1995)
28. Munagala, K., Ranade, A.: I/O-complexity of graph algorithms. In: Proc. of the 10th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA), pp. 687–694 (1999)
29. Nodine, M.H., Goodrich, M.T., Vitter, J.S.: Blocking for external graph searching. Algorithmica **16**(2), 181–214 (1996)
30. Prokop, H.: Cache-oblivious algorithms. Master's Thesis, MIT EECS, June 1999
31. Simon, H.D.: Partitioning of unstructured mesh problems for parallel processing. Comput. Syst. Eng. **2**, 125–148 (1991)
32. Teng, S.-H.: Provably good partitioning and load balancing algorithms for parallel adaptive n-body simulation. SIAM J. Sci. Comput. **19**(2), 635–656 (1998)
33. Teng, S.-H., Wong, C.W.: Unstructured mesh generation: theory, practice, and perspectives. Int. J. Comput. Geom. Appl. **10**(3), 227–266 (2000)
34. Toledo, S.A.: Quantitative performance modeling of scientific computations and creating locality in numerical algorithms. Ph.D. Thesis (1995). Supervisor: Charles E. Leiserson
35. Toledo, S.A.: Locality of reference in LU decomposition with partial pivoting. SIAM J. Matrix Anal. Appl. **18**(4), 1065–1081 (1997)
36. Vudac, R., Demmel, J.W., Yelick, K.A.: The optimized sparse kernel interface (OSKI) library: user's guide for version 1.0.1b. Berkeley Benchmarking and OPtimizationBeBOP) Group, 15 March 2006
37. Vuduc, R.W.: Automatic performance tuning of sparse matrix kernels. Ph.D. Thesis, University of California, Berkeley (2003)

38. Weikum, G., Moenkeberg, A., Hasse, C., Zabback, P.: Self-tuning database technology and information services: from wishful thinking to viable engineering. In: Proceedings of International Conference on Very Large Data Bases (VLDB), pp. 20–31 (2002)
39. Whaley, R.C., Dongarra, J.: Automatically tuned linear algebra software. In: SuperComputing, pp. 1–27 (1998)
40. Yoon, S.-E., Lindstrom, P., Pascucci, V., Manocha, D.: Cache-oblivious mesh layouts. In: ACM SIGGRAPH and Transactions on Graphics, pp. 886–893 (2005)