

# Contiguous Graph Partitioning For Optimal Total Or Bottleneck Communication

Willow Ahrens

**Abstract**—Graph partitioning schedules parallel calculations like sparse matrix-vector multiply (SpMV). We consider contiguous partitions, where the  $m$  rows (or columns) of a sparse matrix with  $N$  nonzeros are split into  $K$  parts without reordering. We propose the first near-linear time algorithms for several graph partitioning problems in the contiguous regime.

Traditional objectives such as the simple edge cut, hyperedge cut, or hypergraph connectivity minimize the total cost of all parts under a balance constraint. Our total partitioners use  $O(Km + N)$  space. They run in  $O((Km \log(m) + N) \log(N))$  time, a significant improvement over prior  $O(K(m^2 + N))$  time algorithms due to Kernighan and Grandjean et. al.

Bottleneck partitioning minimizes the maximum cost of any part. We propose a new bottleneck cost which reflects the sum of communication and computation on each part. Our bottleneck partitioners use linear space. The exact algorithm runs in linear time when  $K^2$  is  $O(N^C)$  for  $C < 1$ . Our  $(1 + \epsilon)$ -approximate algorithm runs in linear time when  $K \log(c_{\text{high}}/(c_{\text{low}}\epsilon))$  is  $O(N^C)$  for  $C < 1$ , where  $c_{\text{high}}$  and  $c_{\text{low}}$  are upper and lower bounds on the optimal cost. We also propose a simpler  $(1 + \epsilon)$ -approximate algorithm which runs in a factor of  $\log(c_{\text{high}}/(c_{\text{low}}\epsilon))$  from linear time.

We empirically demonstrate that our algorithms efficiently produce high-quality contiguous partitions on a test suite of 42 test matrices. When  $K = 8$ , our hypergraph connectivity partitioner achieved a speedup of  $53\times$  (mean  $15.1\times$ ) over prior algorithms. The mean runtime of our bottleneck partitioner was 5.15 SpMVs.

**Index Terms**—Contiguous, Partitioning, Load Balancing, Communication-Avoiding, Chains-On-Chains, Dominance Counting, Least Weight Subsequence, Quadrangle Inequality, Convex, Concave, Monotonic



## 1 INTRODUCTION

**S**PARSE matrix multiplication is often the most expensive subroutine in scientific computing applications. When multiplying a dense vector or matrix, we refer to this kernel as **SpMV** or **SpMM** respectively. Many applications, such as iterative solvers, multiply by the same sparse matrix repeatedly. Parallelization can increase efficiency, and datasets can be large enough that distributed memory approaches are necessary. A common parallelization strategy is to partition the rows (or similarly, the columns) of the sparse matrix and corresponding elements of the dense vector(s) into disjoint parts, each assigned to a separate processor. While there are a myriad of methods for partitioning the rows of sparse matrices, we focus on the case where the practitioner does not wish to change the ordering of the rows and the parts are therefore contiguous.

There are several reasons to prefer contiguous partitioning. The row ordering may already be carefully optimized for numerical considerations (such as fill-in), the natural row ordering may already be amenable to partitioning, or reordering may simply be too costly to implement on the target architecture. Furthermore, several noncontiguous partitioners (spectral methods or other heuristics [1], [2]) work by producing a one-dimensional embedding of the rows, then using a contiguous partitioner to subsequently “round” the ordering into a partition. Similarly, geometric partitioners partition points by splitting a multidimensional embedding, such as the multidimensional spectral embed-

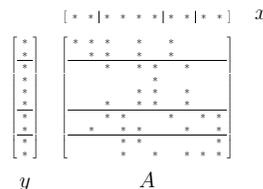


Fig. 1. Our running example matrix, together with an example symmetric partition of  $x$  and  $y$ . Nonzeros are denoted with  $*$ .

ding or simply the natural locations of the corresponding mesh cells. The popular multi-jagged geometric approach recursively splits one spatial dimension at a time using a load-balancing contiguous partitioner [3], [4]. As a final example, acyclic partitioning is used to schedule execution of directed acyclic computation graphs. A prominent approach to acyclic partitioning applies contiguous partitioning to a topological ordering of the graph that respects data dependence relationships [5], [6].

Since noncontiguous partitioning to minimize communication costs is NP-Hard [7], [8], one can view contiguous partitioning as a compromise, where the user is asked to use domain knowledge or heuristics to produce a good ordering, but the partitioning can be performed efficiently and optimally. Our algorithms demonstrate that contiguous partitioning can be efficient in theory and practice, and support highly expressive cost models.

We consider two main partitioning objectives, **total** partitioning and **bottleneck** partitioning. Traditional partitioning objectives minimize the total communication under the constraint that the work or storage should be approximately balanced. However, in many applications of SpMV or SpMM, such as iterative solvers, all processors wait for the last processor to finish. Therefore, it may be more accurate

• Computer Science and Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, MA (willow@csail.mit.edu). This work was supported by a Department of Energy Computational Science Graduate Fellowship, DE-FG02-97ER25308.

to minimize the runtime of the longest-running (bottleneck) processor. Here, we model runtime as the combined cost of communication and computation on a part. This allows us to shift work burdens from extroverted processors which communicate frequently to introverted processors which communicate rarely, more efficiently utilizing resources [9].

## 1.1 Contributions

We propose efficient contiguous partitioners for total and bottleneck objectives. Our algorithms run in linear or near-linear time, representing asymptotic improvements over the state of the art. We evaluate our algorithms experimentally<sup>1</sup> on a suite of 42 test matrices. We show our algorithms to be efficient in practice. Our partitions are often 3× the quality of computation-balanced partitions or evenly sized partitions on matrices in their natural order, spectral order, or Cuthill-McKee order [10], [11]. Our partitioners are competitive with existing graph partitioning approaches when we account for the runtime of the partitioners themselves.

We address symmetric and nonsymmetric partitions. When the partition is nonsymmetric, we address the cases where either the rows or column partition is considered fixed. Without loss of generality, we consider partitions of the rows of an  $m \times n$  matrix with  $N$  nonzeros to run on  $K$  processors. We normalize our runtimes to the time it takes to perform a sparse matrix-vector multiply (SpMV) on the same matrix.

### 1.1.1 Contiguous Total Partitioners

The total contiguous partitioning problem includes the traditional simple edge cut, hyperedge cut, and hypergraph connectivity objectives. We propose an exact algorithm for these costs which runs in time

$$O(Km \log(m) \log(N) + n + N \log(N)),$$

and uses  $O(Km + n + N)$  space. This represents a significant improvement (quadratic to near-linear) over the previous  $O(K(m^2 + n + N))$  time algorithms due to Grandjean et. al. and Kernighan [12], [13]. When  $K = 8$  and the work was 10% balanced, our total partitioner for hypergraph connectivity runs in an average of 667 SpMVs, a mean speedup of 15.1× over the existing algorithm.

### 1.1.2 Contiguous Bottleneck Partitioners

The bottleneck contiguous partitioning problem includes the traditional “Chains-On-Chains” computational load balancing objective. We propose new cost functions which add hypergraph communication terms to the computational load. We propose exact and approximate algorithms for our new costs. Our algorithms use linear space. Our exact algorithm is parameterized by a constant  $H$  and runs in time

$$O(m + n + HN + K^2 \log(m)^2 H^2 N^{1/H}).$$

This can be made strictly linear when when  $K^2$  is  $O(N^C)$  for some constant  $C < 1$ . Our  $(1 + \epsilon)$ -approximate algorithm is parameterized by a constant  $H$  and runs in time

$$O\left(HN + K \log(N) \log\left(\frac{c_{\text{high}}}{c_{\text{low}}\epsilon}\right) H^2 N^{1/H}\right),$$

where  $c_{\text{high}}$  and  $c_{\text{low}}$  are upper and lower bounds on the optimal cost. This can be made strictly linear when  $K \log(c_{\text{high}}/(c_{\text{low}}\epsilon))$  is  $O(N^C)$  for some constant  $C < 1$ . Our experiments set  $H$  to 3. We also propose a simpler  $(1 + \epsilon)$ -approximate algorithm that runs in time

$$O(\log(c_{\text{high}}/(c_{\text{low}}\epsilon))(m + n + N)),$$

which is only near linear but is faster in practice. If the cost functions are monotonic increasing and subadditive, we can derive bounds so that  $\log(c_{\text{high}}/(c_{\text{low}}\epsilon))$  is  $O(\log(K/\epsilon))$ . When  $K = 8$  and  $\epsilon = 10\%$ , our exact, approximate, and lazy bottleneck hypergraph partitioners run in an average of 18, 17.7, and 5.15 SpMVs over our symmetric test matrices, making our algorithms practical even in situations where the matrix is not heavily reused.

### 1.1.3 Core Algorithmic Advances

Our algorithms perform structured queries of the cost of various potential parts to determine a good partition. Our results are enabled by three main algorithmic advances.

- We decompose our novel and traditional partitioning objectives into partwise terms, and taxonomize the resulting partwise costs based on the key properties of monotonicity, sub- or super-additivity, and convexity or concavity. We then reduce the task of computing our costs to two dimensional dominance counting (sparse prefix sum) queries. We generalize an offline two-dimensional dominance counting datastructure due to Chazelle to trade construction time for query time, allowing our bottleneck partitioners to run in linear time [14]. Our dominance counting algorithm may also be of interest for two-dimensional rectilinear load balancing problems with linear cost functions [15], [16], [17].
- Our total partitioner reduces the problem to a sequence of constrained convex or concave least-weight-subsequence (LWS) subproblems. We generalize Eppstein’s constrained convex LWS algorithm from part size constraints to arbitrary monotonic constraints [18]. The constrained LWS algorithm (which runs in time  $O(m \log(m) \log(N) + n + N \log(N))$ ) may be of independent interest for the purposes of partitioning into an arbitrary number of row blocks to minimize total cache misses [19], [20].
- To minimize the bottleneck cost, we adapt state of the art chains-on-chains partitioning algorithms (originally due to Iqbal et. al. and Nicol et. al. [15], [21] and further improved by Pinar et. al. [22]) to support arbitrary monotonic increasing and decreasing cost functions. We simplify the approximate algorithm by relaxing a binary search to a linear search and using an online dominance counter.

## 2 BACKGROUND

We index starting from 1. Consider the  $m \times n$  matrix  $A$ . We refer to the entry in the  $i^{\text{th}}$  row and  $j^{\text{th}}$  column of  $A$  as  $a_{ij}$ . A matrix is called **sparse** if most of its entries are zero. It is more efficient to store sparse matrices in **compressed** formats that only store the nonzeros. Let  $N$  be the number

<sup>1</sup> [github.com/willow-ahrens/ChainPartitioners.jl/tree/2007.16192v4](https://github.com/willow-ahrens/ChainPartitioners.jl/tree/2007.16192v4)

of nonzeros in  $A$ . Without loss of generality, as transposition and format conversion usually run in linear time, we consider the **Compressed Sparse Row (CSR)** format. CSR stores a sorted vector of nonzero column coordinates in each row, and a corresponding vector of values. This is accomplished with three arrays  $pos$ ,  $idx$ , and  $val$  of length  $m+1$ ,  $N$ , and  $N$  respectively. Locations  $pos_j$  to  $pos_{j+1}-1$  of  $idx$  hold the sorted column indices  $i$  such that  $a_{ij} \neq 0$ , and the corresponding entries of  $val$  hold the nonzero values.

A  $K$ -**partition**  $\Pi$  of the rows of  $A$ , assigns each row  $i$  to a single part  $\pi_k$ . Any partition  $\Pi$  on  $n$  elements must satisfy coverage and disjointness constraints.

$$\bigcup_k \pi_k = \{1, \dots, n\}, \quad \forall k \neq k', \pi_k \cap \pi_{k'} = \emptyset. \quad (1)$$

A partition is **contiguous** when adjacent elements are assigned to the same part. Formally,  $\Pi$  is contiguous when,

$$\forall k < k', \forall (i, i') \in \pi_k \times \pi_{k'}, i < i'. \quad (2)$$

We can describe a contiguous partition  $\Pi$  using its **split points**  $S$ , where  $i \in \pi_k$  for  $S_k \leq i < S_{k+1}$ . Thus,  $\pi_k = S_k : (S_{k+1} - 1)$ .

We consider **graphs** and **hypergraphs**  $G = (V, E)$  on  $m$  vertices and  $n$  edges. **Edges** can be thought of as connecting sets of **vertices**. We say that a vertex  $i$  is **incident** to edge  $j$  if  $i \in e_j$ . Note that  $i \in e_j$  if and only if  $j \in v_i$ . Graphs can be thought of as a specialization of hypergraphs where each edge is incident to exactly two vertices. The **degree** of a vertex is the number of incident edges, and the **degree** of an edge is the number of incident vertices.

We distinguish between symmetric and nonsymmetric partitioning regimes. A **symmetric partitioning** regime assumes  $A$  to be square and that the input and output vectors will use the same partition  $\Pi$ . Note that we can use symmetric partitioning on square, asymmetric matrices. The **nonsymmetric partitioning** regime makes no assumption on the size of  $A$ , and allows the input vector (columns) to be partitioned according to some partition  $\Phi$  which may differ from the output vector (row) partition  $\Pi$ . Since our load balancing algorithms are designed to optimize only one partition at a time, we alternate between optimizing  $\Pi$  or  $\Phi$ , considering the other partition to be fixed. When  $\Phi$  is considered fixed and our goal is only to find  $\Pi$ , we refer to this more restrictive problem as **primary alternate partitioning**. When  $\Pi$  is considered fixed and our goal is only to find  $\Phi$ , we refer to this problem as **secondary alternate partitioning**. Alternating partitioning has been examined as a subroutine in heuristic solutions to nonsymmetric partitioning regimes, where the heuristic alternates between improving the row partition and the column partition, iteratively converging to a local optimum [23], [24]. Similar alternating approaches have been used for the related two-dimensional rectilinear partitioning regimes [15], [17].

### 3 PROBLEM STATEMENT

We seek a contiguous partition  $\Pi$  minimizing either

$$\sum_k f_k(\pi) \text{ or } \max_k f_k(\pi), \quad (3)$$

where  $f_k$  is a cost function mapping parts (sets of vertices) to the extended reals (the real numbers and  $\pm\infty$ ). We refer

to these problems as **total** or **bottleneck** partitioning, respectively. The number of parts,  $K$ , may be fixed or allowed to vary. Because we consider only contiguous parts, we may use  $f_k(i, j)$  as a shorthand for  $f_k(i : j - 1)$ .

The cost function,  $f$ , might enjoy certain properties which allow us to make algorithmic optimizations. We say that  $f$  is **monotonic increasing** if for any  $i' \leq i \leq j \leq j'$ ,

$$f(i, j) \leq f(i', j'), \quad (4)$$

and  $f$  is **monotonic decreasing** when the inequality is reversed. We say that  $f$  is **superadditive** if for any  $i \leq l \leq j$ ,

$$f(i, l) + f(l, j) \leq f(i, j), \quad (5)$$

and  $f$  is **subadditive** when the inequality is reversed. Finally, we say that  $f$  is **concave** when  $f$  satisfies the quadrangle inequality for any  $i \leq i' \leq j \leq j'$ ,

$$f(i, j) + f(i', j') \leq f(i, j') + f(i', j), \quad (6)$$

and  $f$  is **convex** when the inequality is reversed. Concavity and convexity imply superadditivity and subadditivity, respectively. When the inequality in (5) becomes an equality, we say  $f$  is **additive**. Additive functions are subadditive, superadditive, convex, and concave.

Note that negating an increasing, subadditive, or concave cost function will produce a decreasing, superadditive, or convex cost function, respectively. All of our properties are preserved under addition and scaling by positive constants. Subadditivity is preserved under max, superadditivity is preserved under min. Monotonicity is preserved under both max and min, but neither concavity nor convexity are preserved under either.

Partitioners often use constraints to reflect storage limits or to balance work. We can use thresholds in our cost functions to reflect such constraints. If our weight limit is  $w_{\max}$ , then we define

$$\tau_{w_{\max}}(w) = \begin{cases} 0 & \text{if } w \leq w_{\max} \\ 1 & \text{otherwise} \end{cases}$$

If  $w$  is increasing, then  $\tau$  is increasing, superadditive, and concave. If  $w$  is decreasing, then  $\tau$  is decreasing, subadditive, and concave. A simple way to add a weight limit on  $w$  to a cost function  $f$  would be to form  $f + \tau_{w_{\max}}(w) \cdot \infty$  (we assume that 0 times  $\infty$  is 0). We can also use thresholds to capture discontinuous phenomena like cache effects. For example, one might use thresholds to switch to more expensive cost models when the input or output vectors no longer fit in cache.

#### 3.1 Traditional Partitioning Objectives

Traditional graph and hypergraph partitioning problems map rows of  $A$  to vertices of a graph or hypergraph, and use edges to represent communication costs. The parts are weighted to represent storage or computation costs, and the objective is to minimize the total communication between a fixed number of parts  $K$  under an  $\epsilon$ -approximate weight balance constraint like

$$\forall k, \sum_{i \in \pi_k} |v_i| < \frac{1 + \epsilon}{K} \sum_i |v_i|, \quad (7)$$

which we might represent as  $\tau_{(1+\epsilon) \cdot N/K}(\sum_{i \in \pi_k} |v_i|)$ .

Graph models for symmetric partitioning of symmetric matrices typically use the **adjacency representation**  $\text{adj}(A)$  of a sparse matrix  $A$ . If  $G = \text{adj}(A)$ , an edge exists between vertices  $i$  and  $i'$  if and only if  $a_{ii'} \neq 0$ . Thus, cut edges (edges whose vertices lie in different parts) require communication of their corresponding columns. However, this model overcounts communication costs in the event that multiple cut edges correspond to the same column, since each column only represents one entry of  $y$  which needs to be sent. Reductions to bipartite graphs are used to extend this model to the possibly rectangular, nonsymmetric case [24]. Graph partitioning under the “edge cut” seeks to optimize

$$\arg \min_{\Pi} |\{E \cap (\pi_k \times \pi_{k'}) : k \neq k'\}|, \quad (8)$$

under balance constraints (7), where  $G = (V, E) = \text{adj}(A)$ . We can rearrange the edge cut (8) objective in the form of Problem (3) by subtracting the constant  $|E|$ ,

$$|\{E \cap (\pi_k \times \pi_{k'}) : k \neq k'\}| - |E| = \sum_k \sum_{i \in \pi_k} -|v_i \cap \pi_k|,$$

where  $\sum_{i \in \pi_k} -|v_i \cap \pi_k|$  is monotonic decreasing and convex.

Inaccuracies in the graph model led to the development of the hypergraph model of communication. Here we use the **incidence representation** of a hypergraph,  $\text{inc}(A)$ . If  $G = \text{inc}(A)$ , edges correspond to columns in the matrix, vertices correspond to rows, and we connect the edge  $e_j$  to the vertex  $v_i$  when  $a_{ij} \neq 0$ . Thus, if there is some edge  $e_j$  which is not cut in a row partition  $\Pi$ , all incident vertices to  $e_j$  must belong to the same part  $\pi_k$ , and we can avoid communicating the input  $j$  by assigning it to processor  $k$  in our column partition  $\Phi$ . In this way, we can construct a secondary column partition  $\Phi$  such that the number of cut edges in a row partition  $\Pi$  corresponds exactly to the number of entries of  $y$  that must be communicated, and the number of times an edge is cut is one more than the number of processors which need to receive that entry of  $y$ , since one of these processors has that entry of  $y$  stored locally. By filling in the diagonal of the matrix, this correspondence still holds when the partition is symmetric [8]. To formalize these cost functions on a partition  $\Pi$ , we define  $\lambda_j(A, \Pi)$  as the set of row parts which contain nonzeros in the  $j^{\text{th}}$  column. Tersely,  $\lambda_j = \{k : i \in \pi_k, a_{ij} \neq 0\}$ . Hypergraph partitioning with the “hyperedge cut” metric seeks to optimize

$$\arg \min_{\Pi} |\{j : |\lambda_j| > 1\}|, \quad (9)$$

and with the “ $(\lambda - 1)$  cut” metric seeks to optimize

$$\arg \min_{\Pi} \sum_{e_j \in E} |\lambda_j| - 1, \quad (10)$$

under balance constraints (7), where  $G = (V, E) = \text{inc}(A)$ . Rearranging again by subtracting  $|E|$ , we find that minimizing the “hyperedge cut” (9) is equivalent to a partwise objective in the form of Problem (3),

$$|\{e_j \in E : |\lambda_j| > 1\}| - |E| = \sum_k -|\{e_j \subseteq \pi_k\}|,$$

because edges entirely contained within a part are unique to that part. The cost  $-|\{e_j \subseteq \pi_k\}|$  is monotonic decreasing

and convex. If we instead add  $|E|$  to the “ $(\lambda - 1)$  cut” (10), we obtain

$$|E| + \sum_{e_j \in E} |\lambda_j| - 1 = \sum_{e_j \in E} |\lambda_j| = \sum_k \left| \bigcup_{i \in \pi_k} v_i \right|,$$

where the last equality comes from counting incidences over parts rather than over edges. The cost  $|\bigcup_{i \in \pi_k} v_i|$  is monotonic increasing and convex.

If we block the rows of a matrix on a serial processor for caching purposes,  $|\bigcup_{i \in \pi_k} v_i|$  also corresponds to the number of cache misses on  $x$  when the row block’s portion of  $y$  fits in cache. While hypergraph partitioning is usually applied to distributed settings with a fixed number of processors  $K$  and a balance constraint on nonzeros, this setting would allow variable  $K$  and limit the number of rows in each part (since each row corresponds to a cached entry of  $y$ .) [19], [20]

The hypergraph model better captures communication in our kernel, but heuristics for noncontiguous hypergraph partitioning problems can be expensive. For example, state-of-the-art multilevel hypergraph partitioners recursively merge pairs of similar vertices at each level. Finding similar pairs usually takes quadratic time [8], [25].

Both the graph and hypergraph formulations minimize total communication subject to a work or storage balance constraint, but it has been observed that the runtime depends more on the processor with the most work and communication, rather than the sum of all communication [24], [26]. Several approaches seek to use a two-phase approach to nonsymmetric partitioning where the matrix is first partitioned to minimize total communication volume, then the partition is refined to balance the communication volume (and other metrics) across processors [27], [28], [29]. Other approaches modify traditional hypergraph partitioning techniques to incorporate communication balance (and other metrics) in a single phase of partitioning [30], [31], [32]. Given a row partition, Bisseling et. al. consider column partitioning to balance communication (the secondary alternate partitioning regime) [33].

While noncontiguous primary and secondary graph and hypergraph partitioning are usually NP-Hard [33], [34], [35], the contiguous case is much more forgiving. Kernighan proposed a dynamic programming algorithm which solves the contiguous graph partitioning problem to optimality in quadratic time, and this result was extended to hypergraph partitioning by Grandjean and Uçar [13], [36].

Simplifications to the cost function lead to faster algorithms. If we ignore communication and minimize only the maximum amount of work in a noncontiguous partition (where the cost of a part is modeled as the sum of some per-row cost model), our partitioning problem becomes equivalent to bin-packing, which is approximable using straightforward heuristics that can be made to run in log-linear time [37]. The “Chains-On-Chains” partitioning problem also optimizes a linear model of work, but further constrains the partition to be contiguous. Chains-On-Chains has a rich history of study, we refer to [22] for a summary. These problems are often described as “load balancing” rather than “partitioning.” In Chains-On-Chains partitioning, the work of a part is typically modeled as directly proportional

to the number of nonzeros in that part. Formally, Chains-On-Chains seeks the best contiguous partition  $\Pi$  under

$$\arg \min_{\Pi} \max_k \sum_{i \in \pi_k} |v_i| \quad (11)$$

where  $G = (V, E) = \text{adj}(A)$ . This objective is already in the form of Problem 3, and the cost model is monotonic increasing and additive.

This cost model is easily computable, and algorithms for Chains-On-Chains partitioning can run in sublinear time. Nicol observed that the work terms for the rows in Chains-On-Chains partitioners can each be augmented by a constant to reflect the cost of vector operations in iterative linear solvers [38]. Local refinements to contiguous partitions have been proposed to take communication factors into account, but our work proposes the first globally optimal linear-time contiguous partitioner with a communication-aware cost model [2], [39].

### 3.2 Novel Objectives

As we will soon see, the techniques used to solve Chains-On-Chains problems can be applied to any monotonic bottleneck objectives in contiguous settings. [22], [40], [41]. In fact, since the runtime of parallel programs depends on the longest-running processor, bottleneck partitioning may be more accurate than total partitioning in parallel settings. Therefore, we propose several monotonic cost models for use in bottleneck partitioning.

Most applications of SpMV or SpMM don't have a synchronization point between communication and computation, so we model the runtime of each processor as the sum of work and communication. The inner loop of the conjugate gradient method, for example, has two synchronization points where all processors wait for the results of a global reduction (a dot product) [42, Chapter 6]. This separates the inner loop into two phases; the dominant phase contains our SpMV or SpMM. The processors first send and receive the required elements of their local portions of the input vector  $x$ , then multiply by  $A$  to produce  $y$ . This phase also contains some elementwise vector operations. We model the per-row computation cost (due to dot products, vector scaling, and vector addition) with the scalar  $c_{\text{row}}$ , and the per-entry computation costs (due to matrix multiplication) with the scalar  $c_{\text{entry}}$ . Thus, the computational work on a processor can be described as

$$c_{\text{row}}|\pi_k| + c_{\text{entry}} \sum_{i \in \pi_k} |v_i| \quad (12)$$

which is monotonically increasing in  $\pi_k$ .

In distributed memory settings, both the sending and the receiving node must participate in transmission of the message. We assume that the runtime of a part is proportional to the sum of local work and the number of received entries. This differs from the model of communication used by Bisseling Et. Al., where communication is modeled as proportional to the maximum number of sent entries or the maximum number of received entries, whichever is larger [33]. While ignoring sent entries may seem to represent a loss in accuracy, there are several reasons to prefer such a model. Processors have multiple threads which can perform local work or process sends or receives independently. If

we assume that the network is not congested (that sending processors can handle requests when receiving processors make them), then the critical path for a single processor to finish its work consists only of receiving the necessary input entries and computing its portion of the matrix product. We model the cost of receiving an entry with the scalar  $c_{\text{message}}$ .

#### 3.2.1 Nonsymmetric Bottleneck Cost Modeling

Since it admits a more accurate cost model, we consider the nonsymmetric partitioning regime first. This regime considers only one of the row (output) space partition  $\Pi$  or the column (input) space partition  $\Phi$ , considering the other to be fixed.

We model our matrix as an incidence hypergraph. The nonlocal entries of the input vector which processor  $k$  must receive are the edges  $j$  incident to vertices  $i \in \pi_k$  such that  $j \notin \phi_k$ . We can express this tersely as  $(\bigcup_{i \in \pi_k} v_i) \setminus \phi_k$ . Thus, our cost model  $f_k(\pi_k, \phi_k)$  for the alternating regime is

$$c_{\text{row}}|\pi_k| + c_{\text{entry}} \sum_{i \in \pi_k} |v_i| + c_{\text{message}} \left| \bigcup_{i \in \pi_k} v_i \setminus \phi_k \right|, \quad (13)$$

which is monotonically increasing in  $\pi_k$  and decreasing in  $\phi_k$ . Intuitively, adding rows to the processor increases work and communication, while adding columns decreases communication.

While we require the opposite partition to be fixed, we will only require the partition we are currently constructing to be contiguous. Requiring both  $\Pi$  and  $\Phi$  to be contiguous is sometimes but not always desirable; such a constraint would limit us to matrices whose nonzeros are clustered near the diagonal. Allowing arbitrary fixed partitions gives us the flexibility to use other approaches for the secondary alternate partitioning problem. For example, one might assign each column to an arbitrary incident part to optimize total communication volume as suggested by Çatalyurek [8]. We also consider the similar greedy strategy to assign each column to a currently most expensive incident part, attempting to reduce the cost of the most expensive part.

Of course, since our alternating partitioning regime assumes we have a fixed  $\Pi$  or  $\Phi$ , we need an initial partition. We propose starting by constructing  $\Pi$  because this partition involves more expensive tradeoffs between work and communication. Since we would have no  $\Phi$  to start with, we assume no locality, upper-bounding the cost of communication and removing  $\Phi$  from our cost model. In the hypergraph model, processor  $k$  receives at most  $|\bigcup_{i \in \pi_k} v_i|$  entries of the input vector. Thus, our cost model would be

$$f(\pi_k) = c_{\text{row}}|\pi_k| + c_{\text{entry}} \sum_{i \in \pi_k} |v_i| + c_{\text{message}} \left| \bigcup_{i \in \pi_k} v_i \right|. \quad (14)$$

which is monotonically increasing in  $\pi_k$ .

Note that any column partition  $\Phi$  will achieve or improve on the modeled cost (14).

#### 3.2.2 Symmetric Bottleneck Cost Modeling

The symmetric case asks us to produce a single partition  $\Pi$  which will be used to partition both the row and column space simultaneously. We do not need to alternate between rows and columns; by adjusting scalar coefficients, we can

achieve an approximation of the accuracy of the nonsymmetric model by optimizing a single contiguous partition under objective (3). Replacing  $\Phi$  with  $\Pi$  in cost (13) yields,

$$f(\pi_k) = c_{\text{row}}|\pi_k| + c_{\text{entry}} \sum_{i \in \pi_k} |v_i| + c_{\text{message}} \left| \left( \bigcup_{i \in \pi_k} v_i \right) \setminus \pi_k \right| \quad (15)$$

Unfortunately, the factor  $|\left(\bigcup_{i \in \pi_k} v_i\right) \setminus \pi_k|$  is not monotonic. However, notice that  $|\left(\bigcup_{i \in \pi_k} v_i\right) \setminus \pi_k| + |\pi_k| = |\left(\bigcup_{i \in \pi_k} v_i\right) \cup \pi_k|$  is monotonic. Assume that each row of the matrix has at least  $w_{\min}$  nonzeros (in linear solvers,  $w_{\min}$  should be at least two, or less occupied rows could be trivially computed from other rows.) We rewrite cost (15) in the following form,

$$f(\pi_k) = (c_{\text{row}} + w_{\min} \cdot c_{\text{entry}} - c_{\text{message}})|\pi_k| + c_{\text{entry}} \sum_{i \in \pi_k} (|v_i| - w_{\min}) + c_{\text{message}} \left| \left( \bigcup_{i \in \pi_k} v_i \right) \cup \pi_k \right|. \quad (16)$$

This function is monotonic when the coefficients on all terms are positive. We therefore require

$$c_{\text{row}} + w_{\min} c_{\text{entry}} \geq c_{\text{message}} \quad (17)$$

Informally, constraint (17) asks that the rows and dot products hold “enough” local work to rival communication costs. These conditions roughly correspond to situations where it is cheaper to communicate an entry of  $y = A \cdot x$  than it is to compute it if the relevant entries of  $x$  were stored locally. These constraints are most suitable to matrices with heavy rows, because increasing the number of nonzeros in a row increases the amount of local work and the communication footprint, but not the cost to communicate the single entry of output corresponding to that row.

Depending on the sparsity of our matrix, we may approximate the modeled cost of the matrix by assuming  $w_{\min}$  to be larger than it really is. If there are at most  $m'$  “underfull” rows with less than  $w_{\min}$  nonzeros, then cost (15) will be additively inaccurate by at most  $m' \cdot w_{\min} c_{\text{entry}}$ .

## 4 COMPUTING COSTS

Our contiguous partitioners will probe the cost of many potential parts, and rely on datastructures that efficiently answer such queries. Our goal is to create datastructures to compute costs (8), (9), (10), (11), (13), (14), and (16).

In (13), we compute  $|\bigcup_{i \in \pi_k} v_i \setminus \phi_k|$  as  $|\bigcup_{i \in \pi_k} v_i| - |\bigcup_{i \in \pi_k} v_i \cap \phi_k|$ . In (16), since  $|\bigcup_{i \in \pi_k} v_i \cup \pi_k|$  is just  $|\bigcup_{i \in \pi_k} v_i|$  with the diagonal of the matrix filled in, we only consider the latter term. One can fill the diagonal explicitly, or compute the result lazily to avoid a copy of the matrix.

Therefore, apart from constants, we need only compute the unique terms  $|\pi_k|$ ,  $\sum_{i \in \pi_k} |v_i|$ ,  $\sum_{i \in \pi_k} |v_i \cap \pi_k|$ ,  $|\{e_j \subseteq \pi_k\}|$ ,  $|\bigcup_{i \in \pi_k} v_i|$ , and  $|\bigcup_{i \in \pi_k} v_i \cap \phi_k|$ .

Our algorithms only make contiguous queries of the form  $\pi_k = i : (i' - 1)$  or  $\phi_k = j : (j' - 1)$ , although the fixed partition  $\Phi$  or  $\Pi$  may not be contiguous, respectively.

Our first term,  $|\pi_k| = i' - i$ , is easy to compute in constant time. Similarly, since the  $pos$  vector in CSR format is a prefix sum (cumulative sum) of the number of nonzeros in each row  $|v_i|$ , we can compute  $\sum_{i \in \pi_k} |v_i| = pos_{i'} - pos_i$

in constant time. If our matrix is not stored in CSR format, we can usually construct  $pos$  in linear time.

The only term with an explicit dependence on  $\phi_k$  is  $|\bigcup_{i \in \pi_k} v_i \cap \phi_k|$ . When  $\Pi$  is fixed, we can construct sorted list representations of each set  $\bigcup_{i \in \pi_k} v_i$  in linear time and space (using, for example, a histogram sort and deleting adjacent duplicates). This allows us to evaluate  $|\bigcup_{i \in \pi_k} v_i \cap \phi_k|$  in  $O(\log(m))$  time by searching our list for the boundaries of the contiguous region of elements which are also in  $\phi_k$ .

What remains to be shown is how to compute the remaining terms for various contiguous  $\pi_k$  when  $\Phi$  is held constant.

### 4.1 Reduction to Dominance Counting

In this section, we reduce computing our remaining terms to two-dimensional dominance counting, a classic computational geometry problem. Consider points of the form  $(i, i')$  in an integer grid  $\mathbb{N}^2$ . We say that a point  $(i_1, i'_1)$  **dominates** a point  $(i_2, i'_2)$  if  $i_1 \geq i_2$  and  $i'_1 \geq i'_2$ . The **dominance counting** problem in two dimensions asks for a data structure to count the number of points dominated by each query point. Note that by negating either the first or second coordinate, we can reverse the first or second inequality to ask for containment constraints.

We start with the term  $\sum_{i \in \pi_k} |v_i \cap \pi_k|$ , which counts the number of nonzero entries  $A_{j,j'}$  where  $j, j' \in \pi_k$ . When  $j < j'$  we represent it with the point  $(-j, j')$ , otherwise we represent it with the point  $(-j', j)$ . Since  $\pi_k = i : (i' - 1)$ , the number of points dominated by  $(-i, i' - 1)$  is the number of nonzeros where both coordinates are contained in  $\pi_k$ .

We handle the term  $|\{e_j \subseteq \pi_k\}|$  similarly. In linear time, we compute  $(i_j, i'_j)$ , the smallest and largest elements of each  $e_j$ . This corresponds to the position of the first and last nonzero in each column of  $A$ . Since  $\pi_k = i : (i' - 1)$ , the number of points  $(-i_j, i'_j)$  dominated by  $(-i, i' - 1)$  is the number of columns  $e_j$  whose nonzeros are completely contained by  $\pi_k$ , or  $|\{e_j \subseteq \pi_k\}|$ .

The final two terms,  $|\bigcup_{i \in \pi_k} v_i|$  and  $|\bigcup_{i \in \pi_k} v_i \cap \phi_k|$ , present more of a challenge. These quantities concern the size of the set of distinct nonzero column locations in some row part. We know that  $\sum_{i \in \pi_k} |v_i|$  is an easy upper bound on  $|\bigcup_{i \in \pi_k} v_i|$ , but it overcounts columns for each row  $v_i$  they are incident to. If we were somehow able to count only the first appearance of a nonzero column in our part, we could compute  $|\bigcup_{i \in \pi_k} v_i|$ . We refer to the pair of a nonzero entry and the next (redundant) nonzero entry in the column as a **link**. If the  $l^{\text{th}}$  nonzero occurs at row  $i_l$  in some column and the closest following nonzero occurs at  $i'_l$  in that column, we call this a  $(i_l, i'_l)$  link, and represent it with the point  $(-i_l, i'_l)$ . Thus,

$$\left| \bigcup_{i \in \pi_k} v_i \right| = \sum_{i \in \pi_k} |v_i| - |\{l : (i_l, i'_l) \in \pi_k\}|. \quad (18)$$

We have already shown how to compute the first term from the  $pos$  array. The second term is the number of points dominated by  $(-i, i' - 1)$ . Figure 2 illustrates this relationship. Our reduction is almost equivalent to the reduction from multicolored one-dimensional dominance counting to two-dimensional standard dominance counting described by Gupta et. al., but our reduction requires only one dominance query, while Gupta’s requires two [43]. We have re-used the

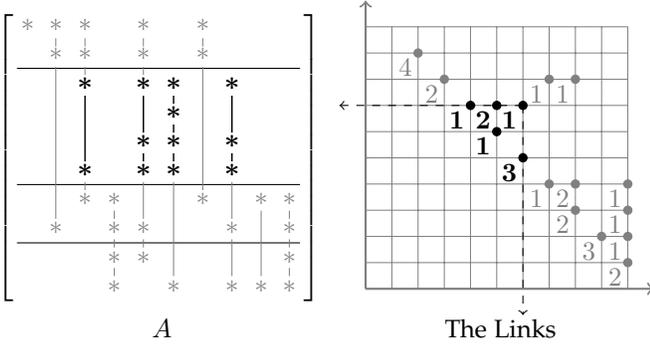


Fig. 2. Links of our example matrix  $A$  are illustrated as line segments connecting elements of  $A$  on left, and as points (with labeled multiplicities) on right. Links residing entirely within part 2 are shown in bold. Part 2 contains two links starting at  $i = 3$  and terminating at  $i = 5$ , and three links starting at  $i = 5$  and terminating at  $i = 6$ . In total, part 2 contains  $1 + 2 + 1 + 1 + 3 = 8$  links, which is equal to the number of points dominated by our dotted region representing the partition split points.

values in the  $pos$  array of the link matrix in CSR format to avoid the second dominance query (dominance queries can be expensive in practice).

Finally, to compute  $|\bigcup_{i \in \pi_k} v_i \cap \phi_k|$  when  $\Phi$  is fixed, we can logically split  $A$  into  $K$  separate matrices where  $A^{(k)}$  is  $A$  where all columns other than  $\phi_k$  are zeroed out. Note that the total number of nonzeros is still  $N$ . Computing  $|\bigcup_{i \in \pi_k} v_i|$  on  $A^{(k)}$  gives us  $|\bigcup_{i \in \pi_k} v_i \cap \phi_k|$ .

To avoid  $K$  separate dominance counting problems for each column part  $\phi_k$ , we concatenate the  $K$  separate problems into one problem and then transform our problem into **rank space** [14], [44]. Recall that our points  $(i_1, j_1), \dots, (i_N, j_N)$  are given to us in  $i$ -major,  $j$ -minor order. We transform to rank space by resorting the points into a  $j$ -major,  $i$ -minor order  $(i_{\sigma(1)}, j_{\sigma(1)}), \dots, (i_{\sigma(N)}, j_{\sigma(N)})$ . Our transformation maps a point  $(i_q, j_q)$  to its position pair  $(q, \sigma(q))$ . Notably,  $i_q < i_{q'}$  if and only if  $q < q'$  and  $j_q < j_{q'}$  if and only if  $\sigma(q) < \sigma(q')$ , so our dominance counts are preserved under our transformation. If we store our two orderings, we can use binary search to find the transformed point at query time.

We can sort the links first by their column part, then into  $i$ -major and  $j$ -major orders in linear time with, for example, two histogram sorts. By storing the boundaries of where each part's subproblem begins and ends, we can then resolve dominance queries within the region corresponding to the target part.

## 4.2 Online Dominance Counting

First, we consider a simple direct method for incremental dominance counting, adapted from [45], the most directly applicable method of its kind [2], [36], [39], [46]. The datastructure requires  $n + 1$  words, can be constructed in  $O(n)$  time, and given we have just answered a query for  $\pi_k = i : (i' - 1)$ , we can answer a query for  $i : i'$  in  $O(|v_{i'}|)$  time, and a query for  $(i - 1) : (i' - 1)$  or  $i' : i'$  in constant time.

Our datastructure starts at  $0 : 0$  with a counter  $c$  of dominated points and a vector  $\Delta$  initialized to 0. To increment  $i'$ , we iterate over each point  $(i_l, i')$  with right coordinate

$i'$ , incrementing  $\Delta_{i_l}$  by one each time, and incrementing  $c$  when  $i_l \leq i'$ . To decrement  $i$ , we simply add  $\Delta_{i-1}$  to  $c$ . To set  $i$  to  $i'$ , we need only set  $c$  to  $\Delta_{i'}$ .

When links are needed, we can use a vector  $h$  of size  $n$  to record the last time we have seen a particular nonzero column as  $i'$  advances from the top of the matrix to the bottom. The next time we see a nonzero in the same column, we can report the corresponding link.

When we compute  $|\bigcup_{i \in \pi_k} v_i \cap \phi_k|$  for various  $\pi_k$ , we maintain  $K$  separate counters for the range  $i' : i'$  for each column part, so that we may increment  $k$  in constant time.

The online approach is efficient in settings when we wish to compute the cost of parts for many starting points corresponding to a fixed end point, or for many end points corresponding to a fixed start point. It is also a simple way to compute the cost of a single partition once.

## 4.3 Offline Dominance Counting

Offline dominance counting is more appropriate when our partitioner makes fewer queries of more arbitrary intervals. The two dimensional problem has been the subject of intensive theoretical study [14], [47], [48]. However, because of the focus on only query time and storage, little attention has been given to construction time, which is always superlinear. Therefore, we modify Chazelle's algorithm to allow us to trade construction time for query time [14].

Dominance counting (and the related semigroup range sum problem [47], [49], [50]) are roughly equivalent to the problem of computing prefix sums on sparse matrices and tensors in the database community. These data structures are called "Summed Area Tables" or "Data Cubes," and they value dynamic update support and low or constant query time at the expense of storage and construction time. We refer curious readers to [51] for an overview of existing approaches, with the caution that most of these works reference the  $o(m^2)$  size of a naive dense representation of the summed area table when they use words like "sublinear" and "superlinear."

Chazelle's dominance counting algorithm uses linear space in the number of points to be stored, requiring log-linear construction time and logarithmic query time. We adapt this structure to a small integer grid, allowing us to trade off construction time and query time. Whereas Chazelle's algorithm can be seen as searching through the wake of a merge sort, our algorithm can be seen as searching through the wake of a radix sort. Our algorithm can also be thought of as a decorated transposition of a CSR matrix. Because the algorithm is so detail-oriented, we give a high-level description, but leave the specifics to the pseudocode presented in Algorithms 1 and 2.

**Algorithm 1.** Construct the dominance counting data structure over  $N$  points  $(i_1, j_1), \dots, (i_N, j_N)$ , ordered on  $i$  initially. We assume we are given the  $j$  coordinates in a vector  $idx$ . We require that  $2^{H_b} > n$ .

```

function CONSTRUCTDOMINANCE( $N, idx$ )
   $gos \leftarrow$  zeroed vector of length  $n + 2$ 
   $gos_1 \leftarrow 1$ 
   $gos_{n+2} \leftarrow N + 1$ 
   $tmp \leftarrow$  uninitialized vector of length  $2^b + 1$ 
   $cnt \leftarrow$  zeroed 3-tensor of size  $2^b + 1 \times \lfloor N/2^b \rfloor + 1 \times H$ 

```

```

byt ← uninitialized vector of length N
for h ← H, H − 1, ..., 1 do
  for J ← 1, 1 + 2hb, ..., n + 1 do
    Fill tmp with zeros.
    for q ← qosJ, qosJ + 1, ..., qosJ+2hb do
      d ← keyh(idxq)
      tmpd+1 ← tmpd+1 + 1
    end for
    tmp1 ← qosJ
    for d ← 1, 2, ..., 2b do
      tmpd+1 ← tmpd + tmpd+1
    end for
    for q ← qosJ, qosJ + 1, ..., qosJ+2hb do
      d ← keyh(idxq)
      q' ← tmpd
      bytq' ← 2hb⌊idxq'/2hb⌋ + idxq mod 2hb
      tmpd ← q' + 1
    end for
    for d ← 1, 2, ..., 2b do
      qosJ+d2(h-1)b ← tmpd
    end for
  end for
  Fill tmp with zeros.
  for Q ← 1, 1 + 2b', ..., N do
    for q ← Q, Q + 1, ..., Q + 2b' do
      d ← keyh(idxq)
      tmpd ← tmpd + 1
    end for
    for d ← 1, 2, ..., 2b do
      cnt(d+1)Qh ← tmpd + cntdQh
    end for
  end for
  (idx, byt) ← (byt, idx)
end for
byt ← idx
return (qos, byt, cnt)
end function

```

**Algorithm 2.** Query the dominance counting data structure for the number of points dominated by  $(i, j)$ .

```

function QUERYDOMINANCE(i, j)
   $\Delta q$  ← posi+1 − 1
  c ← 0
  for h ← H, H − 1, ..., 1 do
    j' ← 2hb⌊j/2hb⌋
    q1 ← qosj' − 1
    q2 ← q1 +  $\Delta q$ 
    d ← keyh(j)
    Q1 ← ⌊q1/2b'⌋ + 1
    Q2 ← ⌊q2/2b'⌋ + 1
    c ← cntdQ2h − cntdQ1h
     $\Delta q$  ← (cnt(d+1)Q2h − cntdQ2h)
     $\Delta q$  ←  $\Delta q$  − (cnt(d+1)Q1h − cntdQ1h)
    for q ← 2b'(Q1 − 1) + 1, 2b'(Q1 − 1) + 2, ..., q1 do
      d' ← keyh(bytq)
      if d' < d then
        c ← c − 1
      else if d' = d then
         $\Delta q$  ←  $\Delta q$  − 1
      end if
    end for
  end for

```

```

for q ← 2b'(Q2 − 1) + 1, 2b'(Q2 − 1) + 2, ..., q2 do
  d' ← keyh(bytq)
  if d' < d then
    c ← c + 1
  else if d' = d then
     $\Delta q$  ←  $\Delta q$  + 1
  end if
end for
end for
return c
end function

```

In this section, assume we have been given  $N$  points  $(i_1, j_1), \dots, (i_N, j_N)$  in the range  $[1, \dots, m] \times [1, \dots, n]$ . Since these points come from CSR matrices or our link construction, we assume that our points are initially sorted on their  $i$  coordinates ( $i_q \leq i_{q+1}$ ) and we have access to an array *pos* to describe where the points corresponding to each value of  $i$  starts. If this is not the case, either the matrix or the following algorithm can be transposed, or the points can be sorted with a histogram sort in  $O(m + N)$  time.

The construction phase of our algorithm successively sorts the points on their  $j$  coordinates in rounds, starting at the most significant digit and moving to the least. We refer to the ordering at round  $h$  as  $\sigma_h$ . We use  $H$  rounds, each with  $b$  bit digits, where  $b$  is the smallest integer such that  $2^{H \cdot b} \geq m$ . Let  $key_h(j)$  refer to the  $h^{\text{th}}$  most significant group of  $b$  bits (the  $h^{\text{th}}$  digit). Formally,

$$key_h(j) = \lfloor j/2^{(h-1)b} \rfloor \bmod 2^b$$

At each round  $h$ , our points will be sorted by the top  $h \cdot b$  bits of their  $j$  coordinates using a histogram sort in each bucket formed by the previous round. We use an array *qos* (similar to *pos*) to store the starting position of each bucket in the current ordering of points. Formally, *qos<sub>j</sub>* will record the starting position for points  $(i_q, j_q)$  where  $j_q \geq j$ . Note that *qos* is of size  $n + 2$  instead of  $n + 1$ , as one might expect, to allow 0 as a possible value for  $j$  during query time.

Although we can interpret the algorithm as resorting the points several times, each construction phase only needs access to its corresponding bit range of  $j$  coordinates (the keys) in the current ordering. The query phase needs access to the ordering of keys before executing each phase. Thus, the algorithm iteratively constructs a vector *byt*, where the  $h^{\text{th}}$  group of  $b$  bits in *byt* corresponds to the  $h^{\text{th}}$  group of  $b$  bits in current ordering of  $j$  coordinates ( $key_h(byt_q) = key_h(j_{\sigma_h(q)})$ ). As the construction algorithm proceeds, we can use the lower bits of *byt* to store the remaining  $j$  coordinate bits to be sorted.

Each phase of our algorithm needs to sort  $\lceil n/2^{hb} \rceil$  buckets. Our histogram sort uses a scratch array of size  $2^b$  to sort a bucket of  $N'$  points in  $O(N' + 2^b)$  time. Thus, we can sort the buckets of level  $h$  in  $O(2^b \lceil n/2^{hb} \rceil + N)$  time, and bucket sorting takes  $O(n + HN)$  time in total over all levels.

A query requests the number of points in our data structure dominated by  $(i, j)$ . In the initial ordering,  $i_q < i$  is equivalent to  $q < pos_i$ . Thus, the dominating points reside within the first  $pos_i - 1$  positions of the initial ordering. Our algorithm starts by counting the number of points such that  $key_1(j_q) < key_1(j)$  and  $q < pos_i$ . All remaining dominating points satisfy  $key_1(j_q) = key_1(j)$ , so let  $q'$  be the number of

points  $key_1(j_q) = key_1(j)$  and  $q < pos_i$ . After our first sorting round, the set of points in the initial ordering where  $key_1(i_q) = key_1(i)$  would have been stored contiguously, and therefore the first  $q'$  of them satisfy  $i_{\sigma_h(q)} < i$ . We can then apply our procedure recursively within this bucket to count the number of dominating points.

We have left out an important aspect of our algorithm. Our query procedure needs to count the number of dominating points that satisfy  $key_h(j_{\sigma_h(q)}) < h$  within ranges of  $q$  that agree on the top  $h \cdot b$  bits of each  $j$ . While  $qos$  stores the requisite ranges of  $q$ , we still need to count the points. In  $O(N + 2^b)$  time, for a particular value of  $h$ , we can walk  $byt$  from left to right, using a scratch vector of size  $2^b$  to count the number of points we see with each value of  $key_h(byt_q)$ . If we cache a prefix sum of our scratch vector once every  $2^{b'}$  points (the prefix sum takes  $O(2^b)$  time), we can use the cache to jump-start the counting process at query time. During a query, after checking our cached count in constant time, we only need to count a maximum of  $2^{b'}$  points at each level to obtain the correct count. Our cache is a 3-tensor  $cnt$ , where  $cnt_{hqd}$  stores the number of points  $q'$  such that  $q' < cq$  and  $key_h(j_{\sigma_h(q')}) < d$ . If we cache every  $2^{b'}$  points, computing  $cnt$  takes  $O(2^b \lceil N/2^{b'} \rceil) = O(N2^{b-b'})$  time per phase. The  $pos$  vector uses  $m + 1$  words, the  $qos$  vector uses  $n + 2$  words, the  $byt$  vector uses  $N$  words, and the  $cnt$  tensor uses at most  $HN2^{b-b'}$  words. The runtime and storage of our offline algorithms are summarized in Table 1.

We consider two ways to set  $H$ ,  $b$ , and  $b'$ . Chazelle proposed setting  $b = 2$ ,  $H = \lceil \log_2(n) \rceil$ , and  $b' = \lceil \log_2(H) \rceil$ . When  $b = 2$ , each key is one bit, and Chazelle suggested we “transpose” the  $byt$  array by storing a bit-packed vector for each level. Because the size of a word bounds the size of the input, we can count the bits at each level of the query step with a constant number of bit-counting instructions, saving a factor of  $\log(n)$  at query time. While storage would be linear and query time would be logarithmic, constructing a dominance counter with these settings would require  $\log_2(n)$  passes. In rank space, this would require an onerous 20 passes for just 1, 048, 576 nonzeros.

We also consider setting  $H$ , the height of the tree and the number of passes, to a small constant like 2 or 3, while keeping storage costs linear, since storage is often a critical resource in scientific computing. For correctness, we minimize  $b$  subject to  $2^{Hb} \geq n$ . We minimize  $b'$  subject to  $2^{b'} \geq H2^b$  to ensure that the footprint of our dominance counter is at most four times the size of  $A$ . These settings do not permit the bit-counting optimization at query time because the digits become larger than a single bit.

When our points come from a rank space transformation, then  $m$  and  $n$  become  $N$ . Note that transforming to rank space simplifies our algorithm because  $pos$  and  $qos$  become the identity and we no longer need to store them. However, we do need to store our orderings of  $i$  and  $j$  values, so the storage requirement is the same and  $m = n = N$ . Although we need to perform binary search for rank-space queries, the  $O(\log(N))$  runtime is dominated by the query time in both parameterizations that we consider.

TABLE 1

Runtime of offline dominance counting parameterizations for points in an  $m \times n$  grid. For simplicity, we assume that  $N \geq m$  and  $N \geq n$ . When we apply a link-based reduction or the matrix is symmetric,  $n = m$ . When points are transformed to rank space,  $m = n = N$ .

Settings	Generic	$b = 2$	$H$ constant
		$H = \lceil \log_2(n) \rceil$ $b' = \lceil \log_2(H) \rceil$	$b = \lceil \log_2(n)/H \rceil$ $b' = b + \lceil \log_2(H) \rceil$
Construct	$O(n + H \cdot N(1 + 2^{b-b'}))$	$O(N \log_2(n))$	$O(n + H \cdot N)$
Query	$O(H2^{b'})$	$O(\log_2(n))$	$O(H^2 n^{1/H})$
Storage	$m + n + N + H \cdot N2^{b-b'}$	$m + n + N$	$m + n + N$

## 5 BOTTLENECK PARTITIONERS

Pinar et. al. present a multitude of algorithms for optimizing linear cost functions [22]. We examine both an approximate and an optimal algorithm, and point out that with fairly minor modifications they can be modified to optimize arbitrary monotonic increasing or decreasing cost functions, given an oracle to compute the cost of a part. We chose the approximate “ $\epsilon$ -BISECT+” algorithm (originally due to Iqbal et. al. [21]) and the exact “NICOL+” algorithm (originally due to Nicol et. al. [15]) because they are easy to understand and enjoy strong guarantees, but use dynamic split point bounds and other optimizations based on problem structure, resulting in empirically reduced calls to the cost function.

Since the approximate algorithm introduces many key ideas which are expanded upon in the exact algorithm, we start with our adaptation of the “ $\epsilon$ -BISECT+” partitioner, which produces a  $K$ -partition within  $\epsilon$  of the optimal cost when it lies within the given bounds.

If our cost is monotonic increasing, the optimal  $K$  partition of a contiguous subset of rows cannot cost more than the optimal  $K$  partition of the whole matrix, since we could simply truncate a partition of the entire set of rows to the subset in question and achieve the same or lesser cost. Therefore, if we know that there exists a  $K$ -partition of cost  $c$ , and we can set the endpoint of the first part to the largest  $i'$  such that  $f_1(1, i') \leq c$ , there must exist  $K - 1$ -partition of the remaining columns that starts at  $i'$  and costs at most  $c$ . This observation implies a procedure that determines whether a partition of cost  $c$  is feasible by attempting to construct the partition, maximizing split points at each part in turn. This procedure is called a PROBE. Clearly, using linear search for split points, PROBE only uses  $O(m)$  evaluations of the cost function. If PROBE uses binary search for split points, it only needs  $K \log_2(m)$  evaluations of the cost function. We can repeatedly call PROBE to search the space of possible costs, stopping when our lower bound on the cost is within  $\epsilon$  of the upper bound. Note that if we had the optimal value of a  $K$ -partition, a single call to PROBE can recover the split points. While Pinar et. al. use this fact to simplify their algorithms and return only the optimal partition value, our cost function is more expensive to evaluate than theirs, so our algorithms have been modified to compute the split points themselves without increasing the number of evaluations of the cost function [22].

The intuition in the monotonic decreasing case is the opposite of the monotonic increasing case. Instead of searching for the last split point less than a candidate cost, we search

for the first. Instead of looking for a candidate partition which can reach the last row without exceeding the candidate cost, we look for a candidate partition which does not need to exceed the last row to achieve the candidate cost. The majority of changes reside in the small details, which we leave to the pseudocode.

Our adapted bisection algorithm is detailed in Algorithm 4. Algorithm 4 differs from the algorithm presented by Pinar et. al. in that it allows for decreasing functions, is stated in terms of possibly different  $f$  for each part, does not assume  $f_k(i, i)$  to be zero, allows for an early exit to the probe function, returns the partition itself instead of the best cost (this avoids extra probes needed to construct the partition from the best cost), and constructs the dynamic split index bounds in the algorithm itself, instead of using more complicated heuristics (which may not apply to all cost functions) to initialize the split index bounds. Note that we assume  $0 < c_{\text{low}}$  only for the purposes of providing a relative approximation guarantee.

Algorithm 4 considers at most  $\log_2(c_{\text{high}}/(c_{\text{low}}\epsilon))$  candidate partition costs. If we use a linear search and online (rank space) dominance counters, PROBE runs in linear time and Algorithm 4 runs in time

$$O\left(\log\left(\frac{c_{\text{high}}}{c_{\text{low}}\epsilon}\right)N\right). \quad (19)$$

If we use binary search and offline (rank space) dominance counters with constant  $H$ , Algorithm 4 runs in time

$$O\left(H \cdot N + K \log(m) \log\left(\frac{c_{\text{high}}}{c_{\text{low}}\epsilon}\right) H^2 N^{1/H}\right), \quad (20)$$

While the binary-searching algorithm has a better asymptotic runtime, the linear-search algorithm can take advantage of the simpler linear dominance counters, and may perform better on smaller problems when  $\epsilon$  is high and therefore the number of probes is low.

The key insight made by Nicol et. al. which allows us to improve our bisection algorithm into an exact algorithm was that there are only  $m^2$  possible costs which could be a bottleneck in our partition, corresponding to  $m^2$  possible pairs of split points that might define a part [15]. Thus, Nicol’s algorithm searches the split points instead of searching the costs, and achieves a strongly polynomial runtime. We will reiterate the main idea of the algorithm, but refer the reader to [22] for more detailed analysis.

Assume that we know the starting split point of processor  $k$  to be  $i$ . Consider the ending point  $i'$  in a partition of minimal cost. If  $k$  were a bottleneck (longest running processor) in such a partition, then  $f_k(i, i')$  would be the overall cost of the partition, and we could use this cost to bound that of all other processors. If  $k$  were not a bottleneck, then  $f_k(i, i')$  should be strictly less than the minimum feasible cost of a partition, and it would be impossible to construct a partition of cost  $f_k(i, i')$ . Thus, Nicol’s algorithm searches for the first bottleneck processor, examining each processor in turn. When we find a processor where the cost  $f_k(i, i')$  is feasible, and less than the best feasible cost seen so far, we record the resulting partition in the event this was the first bottleneck processor. Then, we set  $i'$  so that  $f_k(i, i')$  is the greatest infeasible cost and continue searching, assuming that processor  $k$  was not a bottleneck.

We have made similar modifications in our adaptation of “NICOL+” as we did for our adaptation of “ $\epsilon$ -BISECT+.” Primarily, the algorithm now handles monotonic decreasing functions. We also phrase our algorithm in terms of potentially multiple  $f$ , construct our dynamic split point bounds inside the algorithm instead of using additional heuristics, make no assumptions on the value of  $f_k(i, i)$ , allow for early exits to the probe function, and return a partition instead of an optimal cost. We also consider bounds on the cost of a partition to be optional in this algorithm. Our adaptation of “NICOL+” ([22]) for general monotonic part costs is presented in Algorithm 5.

Although “NICOL+” uses outcomes from previous searches to bound the split points in future searches, a simple worst-case analysis of the algorithm shows that the number of calls to the cost function is bounded by

$$K^2 \log_2(m)^2. \quad (21)$$

Using offline (rank space) dominance counters with constant  $H$ , Algorithm 4 runs in time

$$O\left(H \cdot N + K^2 \log(N)^2 H^2 N^{1/H}\right). \quad (22)$$

We say that  $f$  grows polynomially slower than  $g$  when  $f$  is  $O(g^C)$  for some constant  $C < 1$ . Thus, the approximate partitioner runs in linear time if  $K \log(c_{\text{high}}/(c_{\text{low}}\epsilon))$  grows polynomially slower than  $N^{1-1/H}$ . If costs are subadditive, then we only need  $K \log(K/\epsilon)$ , and therefore  $K \log(1/\epsilon)$ , to grow polynomially slower than  $N^{1-1/H}$ . The exact partitioner runs in linear time if  $K^2$  grows polynomially slower than  $N^{1-1/H}$ . Our algorithms can run in linear time precisely when our partitioners use polynomially sublinear queries, since we are able to offset polynomial query time decreases with logarithmic construction time increases.

For our practical choice of  $H = 3$ ,  $K \log(c_{\text{high}}/(c_{\text{low}}\epsilon))$  needs to grow polynomially slower than  $N^{2/3}$  for linear time approximate partitioning and  $K$  needs to grow polynomially slower than  $N^{1/3}$  for linear time exact partitioning. However, both algorithms use dynamic bounds on split indices to reduce the number of probes, so they are likely to outperform these worst-case estimates. Furthermore,  $K$ , the number of processors, is often a relatively small constant.

## 5.1 Bounding the Costs

Algorithm 4, our approximate bottleneck partitioner, requires upper and lower bounds on the cost function, and both Algorithm 4 and 5 should perform better when given better initial bounds.

Given a monotonic increasing cost function  $f$  over a set of vertices  $V$ ,  $\max_i f(\{i\})$  and  $f(V)$  are naive lower and upper bounds on the bottleneck cost of a  $K$ -partition. These become upper and lower bounds when  $f$  is decreasing, respectively.

However, we can use subadditivity in increasing cost functions to increase the lower bound on the bottleneck cost of a  $K$ -partition. Using subadditivity,  $\sum_k f(\pi_k) \geq f(V)$ , and therefore,

$$\max_k f(\pi_k) \geq \frac{f(V)}{K}. \quad (23)$$

Finding good lower bounds on partition costs can be difficult. Costs like (14) and (15) are subadditive and use the

**Algorithm 3.** Given a monotonic increasing (or decreasing) cost function  $f_k$  defined on pairs of split points, a starting split point  $i$ , and a maximum cost  $c$ , find the greatest (least, respectively)  $i'$  such that  $i \leq i'$ ,  $f_k(i, i') \leq c$ , and  $i'_{\text{low}} \leq i' \leq i'_{\text{high}}$ . Returns  $\max(i, i'_{\text{low}}) - 1$  (returns  $i'_{\text{high}} + 1$ , respectively) if no cost at most  $c$  can be found. Changes needed for decreasing functions are marked with  $\triangleright$ .

```

function SEARCH( $f_k, i, i'_{\text{low}}, i'_{\text{high}}, c$ )
   $i'_{\text{low}} \leftarrow \max(i, i'_{\text{low}})$ 
  while  $i'_{\text{low}} \leq i'_{\text{high}}$  do
     $i' \leftarrow \lfloor (i'_{\text{low}} + i'_{\text{high}})/2 \rfloor$ 
    if  $f_k(i, i') \leq c$  then
       $i'_{\text{low}} = i' + 1$   $\triangleright i'_{\text{high}} = i' - 1$ 
    else
       $i'_{\text{high}} = i' - 1$   $\triangleright i'_{\text{low}} = i' + 1$ 
    end if
  end while
  return  $i'_{\text{high}}$   $\triangleright$  return  $i'_{\text{low}}$ 
end function

```

**Algorithm 4** (BISECT Partitioner). Given monotonic increasing cost function(s)  $f$  defined on pairs of split points, find a contiguous  $K$ -partition  $\Pi$  which minimizes

$$c = \max_k f_k(s_k, s_{k+1})$$

to a relative accuracy of  $\epsilon$  within the range  $0 < c_{\text{low}} \leq c \leq c_{\text{high}}$ , if such a partition exists. This is an adaptation of the “ $\epsilon$ -BISECT+” algorithm by Pinar et. al. [22], which was a heuristic improvement on the algorithm proposed by Iqbal et. al. [21]. We assume that PROBE shares scope with BISECTPARTITION. Changes needed for the case where all  $f$  are monotonic decreasing are marked with  $\triangleright$ .

```

function BISECTPARTITION( $f, m, K, c_{\text{low}}, c_{\text{high}}, \epsilon$ )
  ( $s_{\text{high}1}, \dots, s_{\text{high}K+1}$ )  $\leftarrow (1, m+1, \dots, m+1)$ 
  ( $s_{\text{low}1}, \dots, s_{\text{low}K+1}$ )  $\leftarrow (1, \dots, 1, m+1)$ 
  ( $s_1, \dots, s_{K+1}$ )  $\leftarrow (1, \#, \dots, \#, m+1)$ 
  while  $c_{\text{low}}(1 + \epsilon) < c_{\text{high}}$  do
     $c \leftarrow (c_{\text{low}} + c_{\text{high}})/2$ 
    if PROBE( $c$ ) then
       $c_{\text{high}} \leftarrow c$ 
       $S_{\text{high}} \leftarrow S$   $\triangleright S_{\text{low}} \leftarrow S$ 
    else
       $c_{\text{low}} \leftarrow c$ 
       $S_{\text{low}} \leftarrow S$   $\triangleright S_{\text{high}} \leftarrow S$ 
    end if
  end while
  return  $S_{\text{high}}$   $\triangleright$  return  $S_{\text{low}}$ 
end function
function PROBE( $c$ )
  for  $k = 1, 2, \dots, K - 1$  do
     $s_{k+1} \leftarrow \text{SEARCH}(f_k, s_k, s_{\text{low}k+1}, s_{\text{high}k+1}, c)$ 
    if  $s_{k+1} < s_k$  then  $\triangleright$  if  $s_{k+1} > m+1$  then
       $s_{k+1}, \dots, s_K \leftarrow s_k$   $\triangleright s_{k+1}, \dots, s_K \leftarrow m+1$ 
    return false
  end if
end for
  return  $f_K(s_K, s_{K+1}) \leq c$ 
end function

```

**Algorithm 5** (NICOL Partitioner). Given monotonic increasing cost function(s)  $f$  defined on pairs of split points, find a contiguous  $K$ -partition  $\Pi$  which minimizes

$$c = \max_k f_k(s_k, s_{k+1})$$

within the range  $c_{\text{low}} \leq c \leq c_{\text{high}}$ , if such a partition exists. This is an adaptation of the “NICOL+” algorithm by Pinar et. al. [22], which was a heuristic improvement on the algorithm proposed by Nicol et. al. [15]. We assume that PROBEFROM shares scope with NICOLPARTITION. Changes needed for the case where all  $f$  are monotonic decreasing are marked with  $\triangleright$ .

```

function NICOLPARTITION( $f, m, K, c_{\text{low}}, c_{\text{high}}$ )
  ( $s_{\text{high}1}, \dots, s_{\text{high}K+1}$ )  $\leftarrow (1, m+1, \dots, m+1)$ 
  ( $s_{\text{low}1}, \dots, s_{\text{low}K+1}$ )  $\leftarrow (1, \dots, 1, m+1)$ 
  ( $s_1, \dots, s_{K+1}$ )  $\leftarrow (1, \#, \dots, \#, m+1)$ 
  for  $k \leftarrow 1, 2, \dots, K$  do
     $i \leftarrow s_k$ 
     $i'_{\text{high}} \leftarrow s_{\text{high}k+1}$ 
     $i'_{\text{low}} \leftarrow \max(s_k, s_{\text{low}k+1})$ 
    while  $i'_{\text{low}} \leq i'_{\text{high}}$  do
       $i' \leftarrow \lfloor (i'_{\text{low}} + i'_{\text{high}})/2 \rfloor$ 
       $c \leftarrow f_k(i, i')$ 
      if  $c_{\text{low}} \leq c < c_{\text{high}}$  then
         $s_{k+1} \leftarrow i'$ 
        if PROBEFROM( $c, k$ ) then
           $c_{\text{high}} \leftarrow c$ 
           $i'_{\text{high}} \leftarrow i' - 1$   $\triangleright i'_{\text{low}} \leftarrow i' + 1$ 
           $S_{\text{high}} \leftarrow S$   $\triangleright S_{\text{low}} \leftarrow S$ 
        else
           $c_{\text{low}} \leftarrow c$ 
           $i'_{\text{low}} \leftarrow i' + 1$   $\triangleright i'_{\text{high}} \leftarrow i' - 1$ 
           $S_{\text{low}} \leftarrow S$   $\triangleright S_{\text{high}} \leftarrow S$ 
        end if
      else if  $c \geq c_{\text{high}}$  then
         $i'_{\text{high}} = i' - 1$   $\triangleright i'_{\text{low}} = i' + 1$ 
      else
         $i'_{\text{low}} = i' + 1$   $\triangleright i'_{\text{high}} = i' - 1$ 
      end if
    end while
    if  $i'_{\text{high}} < s_k$  then  $\triangleright$  if  $i'_{\text{low}} > m+1$  then
      break
    end if
     $s_{k+1} \leftarrow i'_{\text{high}}$   $\triangleright s_{k+1} \leftarrow i'_{\text{low}}$ 
  end for
  return  $S_{\text{high}}$   $\triangleright$  return  $S_{\text{low}}$ 
end function
function PROBEFROM( $c, k$ )
  for  $k' = k+1, k+2, \dots, K-1$  do
     $s_{k'+1} \leftarrow \text{SEARCH}(f_{k'}, s_{k'}, s_{\text{low}k'+1}, s_{\text{high}k'+1}, c)$ 
    if  $s_{k'+1} < s_{k'}$  then  $\triangleright$  if  $s_{k'+1} > m+1$  then
       $s_{k'+1}, \dots, s_K \leftarrow s_{k'}$   $\triangleright s_{k'+1}, \dots, s_K \leftarrow m+1$ 
    return false
  end if
end for
  return  $f_K(s_K, s_{K+1}) \leq c$ 
end function

```

same cost for each part, so they obey (23). However, other costs such as (13) use different functions  $f_k$  on each part, and therefore we cannot apply equation (23).

We can work around this limitation by lower-bounding all the functions by the same subadditive one, then applying equation (23). For example, one might lower bound only the work terms in the cost function, since these are uniform across partitions. As another example, one might assume that all threshold functions are zero when calculating a lower bound.

Subadditivity does not help us bound monotonic decreasing cost functions. For these costs, it may make sense to simply use  $\max_k f_k(i, i)$  and  $\max_k f_k(1, m)$  as upper and lower bounds on the cost.

## 6 TOTAL PARTITIONERS

Total partitioning under Problem (3) is quite similar to the **least weight subsequence** problem, which we phrase as evaluating the dynamic program

$$c_{i'} = \min_{1 \leq i < i'} d_i + f(i : i' - 1)$$

$$p_{i'} = \arg \min_{1 \leq i < i'} d_i + f(i : i' - 1) \quad (24)$$

for  $1 < i' < n$  when  $d_i$  is readily computable from  $c_i$ . When there is no constraint on the number of parts, we can minimize the sum (3) by setting  $d_i = c_i$ , and following the back pointers  $p$  to construct  $\pi$ . If, however, we wish to constrain the number of parts to be  $K$ , then we must apply Problem (24)  $K$  times, producing  $K$  separate cost and pointer vectors  $c_k$  and  $p_k$ , where in the  $k^{\text{th}}$  application of (24),  $d_i = c_{k-1, i}$ . The naive dynamic programming algorithm for variable  $K$  makes  $O(m^2)$  queries to the cost function, and runs in  $O(m^2 + N)$  time using our online dominance counters in rank space (we assume the inner loop over  $i$  is evaluated in reverse). The naive dynamic programming algorithm for fixed  $K$  makes  $O(Km^2)$  queries and runs in  $O(K(m^2 + N))$  time using our online dominance counters in rank space. If the cost function  $f$  is uniform across parts, we can reduce this to  $O(Km^2 + N)$  by evaluating  $f$  once for each interval  $i : i'$  and using that value to update  $K$  simultaneous LWS recurrences. We refer to this as the “simultaneous” dynamic programming approach.

Previous approaches point out that when the weight of each part is monotonic and constrained, we can use lower and upper bounds on split points in our dynamic program to obtain heuristic improvements [36], [45]. First, when evaluating the recurrence of (24), if we evaluate  $i$  starting at  $i' - 1$  and work backwards towards 1, the weight of the candidate part will only increase, and we can stop the recurrence as soon as the weight limit is exceeded. Note also that if  $i'_1 > i'_2$ , the first feasible  $i_1$  will be greater than the first feasible  $i_2$ .

Additionally, when  $K$  is fixed, we can bound the range of feasible  $i'$  by considering the least and greatest  $i'$  might be given that the previous  $k$  parts and following  $K - k$  parts are weight constrained. If  $s_{\text{low}, k} < s_k < s_{\text{high}, k}$  are the bounds on the split points for the  $k^{\text{th}}$  partition, then we can define  $s_{\text{low}, k}$  as the least index satisfying  $w(s_{\text{low}, k} : s_{\text{low}, k+1} - 1) \leq w_{\text{max}}$ , and  $s_{\text{high}, k}$  as the greatest

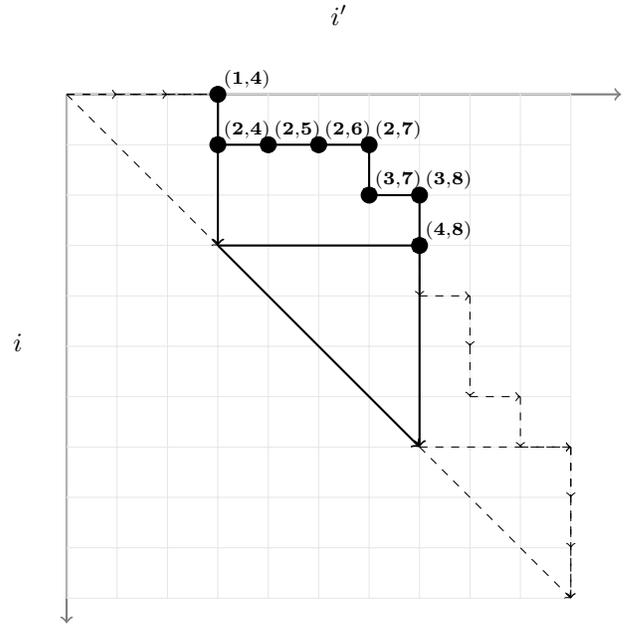


Fig. 3. The feasible regions of setup and cleanup phases under a weight limit of 12 nonzeros per part. The first (nontrivial) setup phase (irregular upper region) and cleanup phase (triangular lower region) are displayed in bold. The pairs  $(\sigma_1, \sigma'_1), \dots, (\sigma_8, \sigma'_8)$  are shown from the upper left to the lower right of the feasible region of the setup phase. Compare this figure to the strictly triangular phases of [18, Figure 2].

index satisfying  $w(s_{\text{high}, k+1} : s_{\text{high}, k} - 1) \leq w_{\text{max}}$ . We can compute these split point bounds in linear time using online dominance counters and simply walking the matrix once from top to bottom and once from bottom to top. If the weight is simply a limit  $w_{\text{max}}$  on the number of rows in each part, these heuristic improvements would bring the running time down to  $O(w_{\text{max}} * m + N)$  and  $O(K(w_{\text{max}} * m + N))$ , respectively. The “simultaneous” version would run in time  $O(K(w_{\text{max}} * m) + N)$ .

Unfortunately, these techniques still require roughly quadratic time in general. We now show how the techniques described in this paper can bring the runtime down to log-linear time using convexity or concavity in the cost function.

Efficient algorithms exist to solve (24) when  $f$  is concave or convex [52], [53], [54]. In this work, we focus on linearithmic solutions because they are simple to implement and work well in practice. The solutions of Galil and Giancarlo can solve (24) for convex or concave cost functions in  $m \log(m)$  steps, using only one query to the cost function in each step. Unfortunately, these queries are unstructured, so we must use offline dominance counters. Using Chazelle’s parameterization in rank space where  $b = 2$  in Table 1, the runtime of the resulting least weight subsequence algorithm becomes  $O(N \log(N) + m \log(m) \log(N))$ , and when  $K$  is fixed, the runtime becomes  $O(N \log(N) + K * m \log(m) \log(N))$ .

However, while concave cost functions are still concave after adding monotonic balance constraints, convex functions are no longer convex after adding such constraints. Since most of our atoms are convex, we must modify our least weight subsequence algorithm to handle such cases. We take a similar approach to Eppstein, who formulated an

algorithm for cost functions which are alternately convex, then concave, then convex, et cetera [18]. While the alternations considered by Eppstein were constrained to fixed width intervals, we consider only the case where we have a convex cost subject to an upper limit on a monotonic weight function. The width of the feasible region may therefore be irregular. While we will not consider an arbitrary number of alternations in the cost, our transition from a convex cost to the concave constraint is more general than the fixed-width intervals considered by Eppstein.

Let  $p_{\text{low},i'}$  be the least index such that  $w(p_{\text{low},i'} : i' - 1) \leq w_{\text{max}}$ . Notice that if  $i'_1 \leq i'_2$ ,  $p_{\text{low},i'_1} \leq p_{\text{low},i'_2}$ . Consider the sequence of at most  $m$  split points  $t_1, t_2, \dots$  such that  $t_l = p_{\text{low},t_{l+1}}$ . We can construct  $t$  in reverse starting at  $m$  and following the pointers  $p_{\text{low}}$ . Our algorithm to compute (24) works in batches corresponding to each range  $t_k : t_{k+1} - 1$ .

Given  $c_1, \dots, c_{t_l}$ , consider the problem of computing

$$c_{i'} = \min_{1 \leq i < i'} d_i + f(i : i' - 1)$$

for  $i' \in \{t_l, \dots, t_{l+1} - 1\}$ . We split the feasible range of the indices into two subproblems, a more complicated setup phase,

$$c_{\text{setup},i'} = \min_{p_{\text{low},i'} \leq i < t_l} d_i + f(i : i' - 1),$$

and a simpler cleanup phase,

$$c_{i'} = \min(c_{\text{setup},i'}, \min_{t_l \leq i < i'} d_i + f(i : i' - 1)).$$

The cleanup phase can be computed with any standard algorithm for (24), but we need to transform the setup phase first before reducing to (24). If  $a, b = t_l, t_{l+1}$ , let  $\sigma, \sigma'$  be defined as

$$\begin{aligned} \sigma_1 &= p_{\text{low},a}, & \sigma'_1 &= a, \\ \sigma_2 &= p_{\text{low},a} + 1, & \sigma'_2 &= a, \\ & \vdots & & \vdots \\ \sigma_{\dots} &= p_{\text{low},a+1}, & \sigma'_{\dots} &= a + 1, \\ \sigma_{\dots} &= p_{\text{low},a+1} + 1, & \sigma'_{\dots} &= a + 1, \\ & \vdots & & \vdots \\ \sigma_{2(b-a)} &= p_{\text{low},b-1}, & \sigma'_{2(b-a)} &= b - 1, \end{aligned}$$

Since  $p_{\text{low},b-1} < a$ , every element of  $\sigma$  is less than every element of  $\sigma'$ . When  $j \leq j'$ , the interval  $\sigma_j : \sigma'_j$  is feasible. Finally, if  $a \leq i' < b$  and  $i : i'$  is feasible, there exists  $j < j'$  where  $\sigma_j = \sigma'_j$ . Thus, we can compute the setup phase with the transformed least weight subsequence problem

$$c_{\text{transformed},j'} = \min_{1 \leq j \leq 2(b-a)} d_{\sigma_j} + f(\sigma_j : \sigma'_{j'} - 1),$$

and finally setting  $c_{\text{setup},i'} = c_{\text{transformed},j'}$ , where  $j'$  is the greatest index such that  $\sigma'_{j'} = i'$ . We pick the greatest such index so that the full feasible range of  $i$  is included.

Each least weight subsequence subproblem can be computed with Galil and Giancarlo's algorithm using  $O(l \log l)$  steps, where  $l$  is the length of the problem. Thus, the setup phase of each segment of length  $(b - a)$  can be computed in  $O(2(a - b) \log(2(a - b)))$  steps and the cleanup phase takes  $O((b - a) \log(b - a))$  steps. Since  $b - a < m$  and

TABLE 2  
Symmetric and asymmetric test matrices used in our test suite. The SI prefixes “ $\mu$ ,” “ $m$ ,” “ $K$ ,” and “ $M$ ” represent factors of  $10^{-6}$ ,  $10^{-3}$ ,  $10^3$ , and  $10^6$ , respectively.

Symmetric Matrices			Asymmetric Matrices		
Group / Matrix	$m \times n$	$N$	Group / Matrix	$m \times n$	$N$
Boeing/ct20stif	52.3K×52.3K	2.7M	ATandT/onetone1	36.1K×36.1K	341K
Boeing/pwtk	218K×218K	11.6M	ATandT/onetone2	36.1K×36.1K	228K
Chen/pkustk03	63.3K×63.3K	3.13M	Averous/epb3	84.6K×84.6K	464K
Chen/pkustk14	152K×152K	14.8M	Bombhof/circuit_4	80.2K×80.2K	308K
Cunningham/qa8fk	66.1K×66.1K	1.66M	Grund/bayer01	57.7K×57.7K	278K
Gupta/gupta2	62.1K×62.1K	4.25M	HB/gemat11	4.93K×4.93K	33.2K
HB/bcsstk30	28.9K×28.9K	2.04M	Hollinger/g7jac180	53.4K×53.4K	747K
HB/bcsstk32	44.6K×44.6K	2.01M	Hollinger/mark3jac140sc	64.1K×64.1K	400K
HB/dwt_607	607×607	5.13K	LPnetlib/lp_cre_b	9.65K×77.1K	261K
HB/sherman3	5K×5K	20K	LPnetlib/lp_cre_d	8.93K×73.9K	247K
Hamm/bcircuit	68.9K×68.9K	376K	LPnetlib/lp_ken_11	14.7K×21.3K	49.1K
Mulvey/finan512	74.8K×74.8K	597K	LPnetlib/lp_ken_13	28.6K×42.7K	97.2K
Nasa/nasasrb	54.9K×54.9K	2.68M	LPnetlib/lp_l_gosh	3.79K×13.5K	100K
PARSEC/H2O	67K×67K	2.22M	Mallya/lhr07	7.34K×7.34K	157K
Rothberg/cfd1	70.7K×70.7K	1.83M	Mallya/lhr14	14.3K×14.3K	308K
Schenk_IBMNA/c-64	51K×51K	718K	Mallya/lhr17	17.6K×17.6K	382K
Simon/venkat01	62.4K×62.4K	1.72M	Mallya/lhr34	35.2K×35.2K	764K
TKK/smt	25.7K×25.7K	3.75M	Mallya/lhr71c	70.3K×70.3K	1.53M
			Meszaros/co9	10.8K×22.9K	110K
			Meszaros/cq9	9.28K×21.5K	96.7K
			Meszaros/mod2	34.8K×66.4K	200K
			Meszaros/nl	7.04K×15.3K	47K
			Meszaros/world	34.5K×67.1K	199K
			Shyy/shyy161	76.5K×76.5K	330K

there are at most  $m$  segments, the overall algorithm again runs in  $O(m \log(m))$  steps. Each step involves a constant number of calls to the cost function. While linearithmic LWS algorithms obviate the need for feasible range optimizations within each LWS subproblem, we can still use the feasible range to restrict the size of each LWS subproblem in our  $K$ -step solution to the fixed- $K$  partitioning problem.

## 7 RESULTS

We compare our partitioners on a set of 42 sparse matrices. We include matrices used by Pinar and Aykanat to evaluate solutions to the Chains-On-Chains problem [22] (minimizing the maximum work without reordering), matrices used by Çatalyurek and Aykanat to evaluate multilevel hypergraph partitioners (minimizing the sum of communication) [8], and matrices used by Grandjean and Uçar to evaluate contiguous hypergraph partitioners (minimizing the sum of communication without reordering) [36]. Since we intend to partition without reordering, we also chose some additional matrices of our own to introduce more variety in the tested sparsity patterns. We exclude “Qaplib/lp\_nug30” as KaHyPar could not partition it in under 24 hours. The test suite is summarized in Table 2.

In addition to partitioning natural orderings, we also evaluate our algorithms on two of the most popular bandwidth-reducing algorithms, the Reverse Cuthill-McKee (RCM) [10] and spectral [11] orderings, which have a history of application to partitioning [23], [55]. The RCM ordering uses a breadth-first traversal of the graph, prioritiz-

ing lower degree vertices first. We follow the linear-time implementation described in [56]. The spectral reordering orders vertices by their values in the Fiedler vector, or the second-smallest eigenvector of the Laplacian matrix. We used the default eigensolver given in the Laplacians.jl library ([github.com/danspielman/Laplacians.jl](https://github.com/danspielman/Laplacians.jl)), the only eigensolver we tested that was able to solve all our largest, worst-conditioned problems in a reasonable time. In order to apply these reorderings to asymmetric matrices, we instead reorder a bipartite graph where nodes correspond to rows and columns in our original matrix, which are connected by edges when nonzeros lie at the intersection of a row and a column, as described by Berry et. al. [57].

Several results are presented as performance profiles, allowing us to compare our partitioners on the entire dataset at once [58]. Partitioners are first measured by the relative deviation from the best performing partitioner for each matrix individually. We then display, for each partitioner, the fraction of test instances that achieve each target deviation from the best partitioner.

Our partitioners optimize different cost models, but we use the same coefficients whenever possible. Since the coefficients should depend on which linear solver is to be partitioned for which parallel machine, we use powers of 10 which are easy to understand and correspond to the likely orders of magnitude involved. We assume that the cost of processing one nonzero in an SpMV is  $c_{\text{entry}} = 1$ . There are overheads to starting the multiplication loop in each row and several per-row costs incurred by dot products in linear solvers. We assume that the per-row cost is an order of magnitude larger;  $c_{\text{row}} = 10$ . The peak MPI bandwidth on Cori, a supercomputer at the National Energy Research Scientific Computing Center, has been measured at approximately 8 GB/s [59]. Cori uses two Intel®Xeon®Processor E5-2698 v3 CPUs on each node (one per socket). The peak computational throughput of one CPU is advertised as 1.1 TB/s (256-bit SIMD lane at 2.3 GHz on each of 16 cores). Since this is approximately two orders of magnitude faster than the communication bandwidth, we set  $c_{\text{message}} = 100$ .

The partitioners we tested are described in Table 3. Our asymmetric alternating partitioners partition the primary dimension first, then partition the secondary dimension. We found that continued alternation did not significantly improve solution quality, so we do not consider partitioners which attempt to refine the initial partitioning.

All experiments were run on a single core of an Intel®Xeon®Processor E5-2695 v2 CPU running at 2.4GHz with 30MB of cache and 128GB of memory. We implemented our partitioners in Julia 1.5.1 and normalize against the Julia Standard Library SpMV; we do not expect much variation across memory-bound single-core SpMV implementations. To measure runtime, we warm up by running the kernel first, then take the minimum of at most 10,000 executions or 5 seconds worth of sampling, whichever happens first. Since some of our algorithms use randomization, we measure the average quality over 100 trials.

Partitioning represents a tradeoff between partitioning time and partition quality. Since the runtime of the partitioner must compete with the runtime of the solver, we normalize the measured serial runtime of our partitioners against the measured serial runtime of an SpMV on the

TABLE 3

Partitioner components, described under their shorthand labels. Algorithms proposed in this work are underlined. Steps may be combined, with the convention that the first partitioning step is performed on rows, and the second on columns. As an example, “CuthillMcKee, Balance Conn., Balance Conn.” corresponds to reordering the matrix, then optimally partitioning the rows under (14), then the columns under (13). Labels in parenthesis differentiate which algorithm was used, when applicable.

Split Equally: Assign an equal number of rows and columns to each part. Contiguous.

Balance Work: Bottleneck partition to balance load (12). Contiguous.

Assign Local: Assign each column to the part of a randomly selected incident row. Noncontiguous.

Assign Greedy Conn.: Working in random order, assign each column to a part to greedily balance bottleneck load + communication (13). Noncontiguous.

Balance Mono. Conn.: Bottleneck partition to balance monotonized symmetric load + communication (16). Contiguous.

Balance Conn.: Bottleneck partition to balance load + communication (14) or (13). Contiguous.

Minimize Simple Cut: Total partition to minimize simple edge cut (8). Contiguous.

Minimize Hyper. Cut: Total partition to minimize hyperedge cut cost (9). Contiguous.

Minimize Conn.: Total partition to minimize  $\lambda - 1$  cut cost (10). Contiguous.

Block Equally: Split equally into parts of cardinality  $C$ . Contiguous.

Block Conn.: Split into parts of cardinality at most  $C$  to minimize total cache misses (13). Contiguous.

Metis: Use direct,  $K$ -way Metis with sorted heavy edge matching to optimize total edge cut (8) with a maximum of 10% work imbalance under (12) [7]. If matrix is asymmetric, use the symmetrized bipartite representation of matrix to produce separate row and column partitions. Noncontiguous.

KaHyPar: Use direct,  $K$ -way KaHyPar to optimize  $\lambda - 1$  cut (10) with a maximum of 10% nonzero imbalance [60]. Noncontiguous.

Spectral: Spectrally reorder. Noncontiguous.

CuthillMcKee: Reorder with Cuthill-McKee. Noncontiguous.

(Exact): Algorithm 5 (Nicol’s Algorithm) was used to exactly bottleneck partition, and an offline dominance counter with  $H = 3$  was employed to calculate costs.

(Approx): Algorithm 4 (Bisection) was used to 10%-approximately bottleneck partition, and an offline dominance counter with  $H = 3$  was employed to calculate costs.

(Lazy): Algorithm 4 (Bisection) was used to 10%-approximately bottleneck partition, and a fused online dominance counter was employed to calculate costs.

(Dynamic): Dynamic programming with split point bounds and constraint-based early search termination was used to solve least-weight subsequence problems for each part separately. An online dominance counter was used to calculate costs.

(Dynamic’): Dynamic programming with split point bounds and constraint-based early search termination was used to solve  $K$  simultaneous least-weight subsequence problems. An online dominance counter was used to calculate costs.

Quadrangle: Our quadrangle-accelerated algorithm with split point bounds was used to solve least-weight-subsequence subproblems for each part separately, and an offline dominance counter with  $H = 3$  was employed to calculate costs.

same matrix. This also allows us to use aggregate runtime statistics over the whole test set.

Figure 5 compares the quality of contiguous partitions under the several cost metrics we have addressed in this work. This figure shows that even in the contiguous regime, cost-aware partitioners can improve significantly over simpler strategies such as equal splitting or work balancing. The benefits were especially pronounced for bottleneck partitioning, often improving by more than a factor of  $3\times$ . Similar benefits were observed when partitioning Cuthill-

McKee or spectrally reordered symmetric matrices, but were less pronounced when  $K$  was smaller, since these algorithms cluster nonzeros more evenly along the diagonal. The asymmetric bottleneck partitioner “Balance Comm, Balance Comm” constrains the second partition to be contiguous, and is therefore more competitive on these bandwidth-reduced reordered matrices where nonzeros occur in a roughly diagonal pattern. The asymmetric “Balance Comm, Assign Greedy” and “Balance Work, Assign Local” partitioners allow the column partition to be noncontiguous, and are therefore more robust to natural orderings. While our partitioners significantly improved on the quality of the equal splitting strategy, no such improvements were found when blocking to minimize cache misses.

Figure 6 compares the normalized runtime of all of our partitioners. In general, the relative runtimes of these partitioners agrees with their asymptotic descriptions. Contiguous bottleneck partitioning is by far the fastest partitioning strategy, and while contiguous total hyperedge cut partitioning is much faster than general total hyperedge cut partitioning (KaHyPar), it is also slower than general simple edge cut partitioning (Metis).

When total partitioning, using offline dominance counting with  $b = 2$  and our quadrangle-accelerated algorithm was significantly faster than the dynamic programming approach with online dominance counting when the number of parts  $K$  was small, but the opposite was true when  $K$  was large. Note that both algorithms reduce the  $K$ -partitioning problem to a sequence of  $K$  least-weight subsequence problems. As  $K$  becomes smaller, the average weight constraint on each part grows, and our dynamic programming algorithm endures longer split point searches. When the average weight constraint is long enough, the quadrangle algorithm’s asymptotic advantage becomes visible. Since the cache blocking problem can be solved as a single constrained least-weight subsequence problem, Figure 7 provides perspective on this effect by comparing the runtime of the dynamic programming algorithm and the quadrangle algorithm as the weight constraint grows. We see that the quadrangle algorithm begins to outperform the dynamic programming algorithm when the weight constraint exceeds about 1000 vertices. When  $K = 8$ , the dynamic, simultaneous, and quadrangle total connectivity partitioners had average runtimes of  $1.16e04$ ,  $9.24e03$ , and 667 SpMV, respectively. The quadrangle algorithm resulted in a maximum speedup of  $53\times$  and a mean speedup of  $15.1\times$  over the dynamic programming algorithm.

When bottleneck partitioning, using offline dominance counting with  $H = 3$  and our linear time Algorithms 4 or 5 was efficient enough to be practical. However, for approximate partitioning, it was empirically fastest to use online dominance counters and fuse them directly into our linear search version of Algorithm 4, even though it is asymptotically slower by a factor of  $\log(c_{\text{high}}/(c_{\text{low}}\epsilon))$ . The resulting pseudocode is described in Appendix A. When  $H = 3$ , the exact algorithm is expected to run in linear time when  $K$  grows slower than  $m^{1/3}$ . When  $K = 8$ , on average over our symmetric matrices, the exact, approximate, and lazy approximate algorithms were able to partition under cost (16) in an average of 18, 17.7, and 5.15 SpMVs, respectively. The improved runtime of the linear search algorithm

is due to the simplicity of the implementations, since the cost calculations are fully inlined.

Figure 4 compares symmetric and asymmetric partitioners on our bottleneck cost in the case where the partition need not be contiguous. The figure shows that contiguous communication-aware partitioners are often competitive with reordering partitioners when we account for setup time and realistic levels of partition reuse. Furthermore, these results show that communication-aware bottleneck contiguous partitioning is more effective when the number of parts is high (and the aspect ratio of each part emphasizes communication over computation). Cuthill-McKee reordering before partitioning is only a good strategy for symmetric matrices when the number of parts is low. The KaHyPar hypergraph partitioner produced the best quality solutions, but took far too long to partition, and fell off the drawable region of the graph.

Figure 8 is meant to help give intuition for how the sparsity pattern of a matrix affects contiguous partitioning performance. Contiguous partitioning returns for PARSEC/H2O, a quantum chemistry problem, show the scale-dependence of structure utilization. For our naturally-ordered partitions, we see that increasing the number of processors does not reduce the cost until the partitions are skinny enough to resolve the structure in the matrix. Since the pattern is roughly 32 large diamonds long along the diagonal, we can only split all the diamonds at around 64 processors, which is roughly when the cost of the partitions begins to decrease in earnest. Mallya/lhr34 is a matrix arising in a nonlinear solver in a chemical process simulation. Because this matrix has a  $2 \times 2$  macro block structure, it is easily split into two parts in its natural ordering. However, continued bisection increases communication costs until there are enough parts to effectively split the finer structures in the matrix. Since Mallya/lhr34 is asymmetric, we can compare the effectiveness of greedy and contiguous secondary partitioning. Nonzeros occur in roughly two bands; the greedy strategy can split local rows among column parts in both bands, while the contiguous strategy suffers because the nonzeros aren’t clustered on the diagonal.

## 8 CONCLUSION

Traditional graph partitioning is NP-Hard, and only a limited set of objectives have been considered. While the ordering of the rows and columns of a matrix does not affect the meaning of the described linear operation, there are many situations where it carries useful information about the problem structure. Contiguous partitioning shifts the burden of reordering onto the user, asking them to use domain-specific knowledge or known heuristics to produce good orderings. In exchange, we show that the contiguous partitioning problem can be solved optimally in linear or near linear time and space, provably optimizing cost models which are close to the realities of distributed parallel computing.

Researchers point out that traditional graph partitioning approaches are inaccurate, since they minimize the total communication, rather than the maximum runtime of any processor under combined work and communication factors. [9], [26], [28], [33]. We show that, in the contiguous

## General Partitioning

### Bottleneck Load + Connectivity On Symmetric Matrices

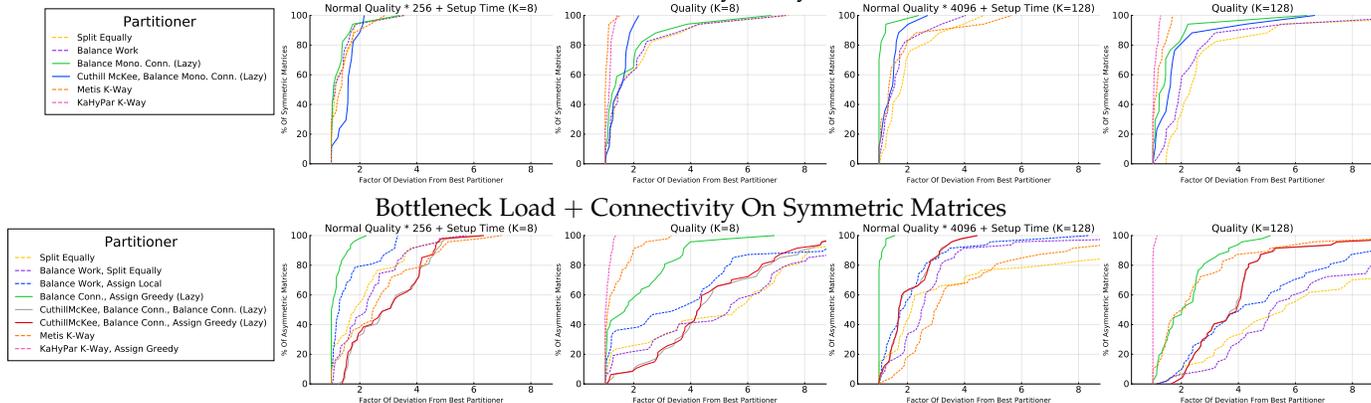
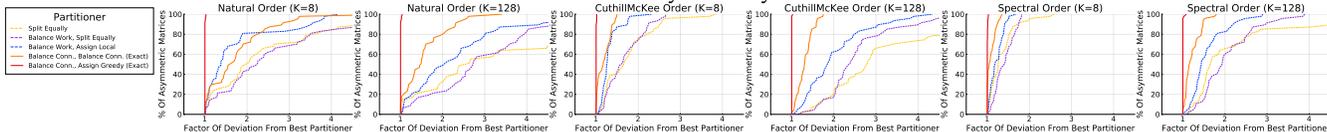


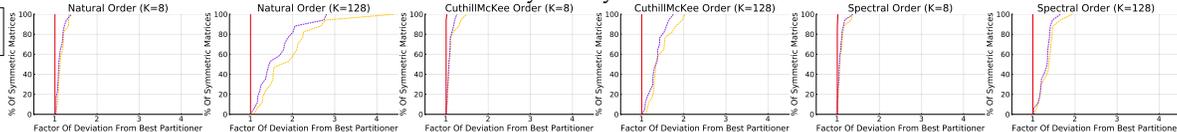
Fig. 4. Performance profiles comparing normalized modeled quality of our general (possibly noncontiguous) partitioners (Table 3) on symmetric and asymmetric test matrices (Table 2) in realistic and infinite reuse situations. Quality is measured with cost (13), using the coefficients  $c_{\text{entry}} = 1$ ,  $c_{\text{row}} = 10$ , and  $c_{\text{message}} = 100$ . For symmetric matrices, we require that the associated partitions be symmetric (we use the same partition for rows and columns). Our asymmetric test matrices also include their transposes. Some of the partitioners may reorder the matrix; setup time includes reordering operations.

## Contiguous Partition Quality Improvements

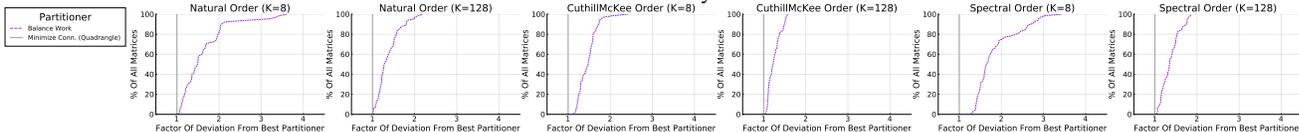
### Bottleneck Load + Connectivity On Asymmetric Matrices



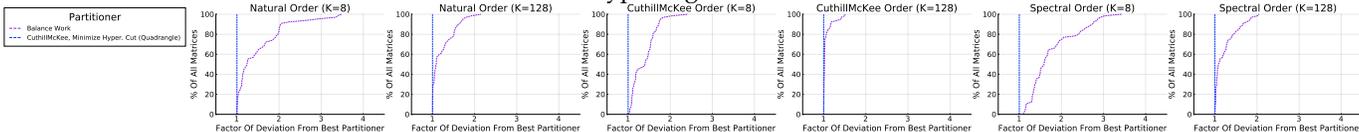
### Bottleneck Load + Connectivity On Symmetric Matrices



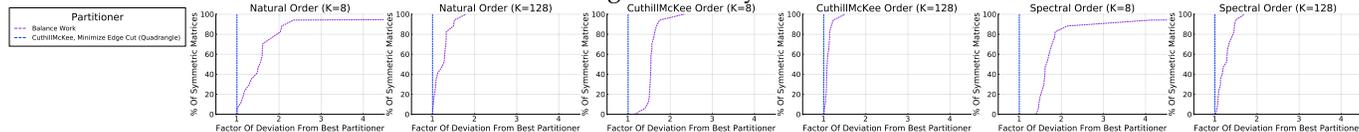
### Load Balanced Total Connectivity On All Matrices



### Load Balanced Hyperedge Cut On All Matrices



### Load Balanced Edge Cut On Symmetric Matrices



### Cache Blocked Connectivity On All Matrices

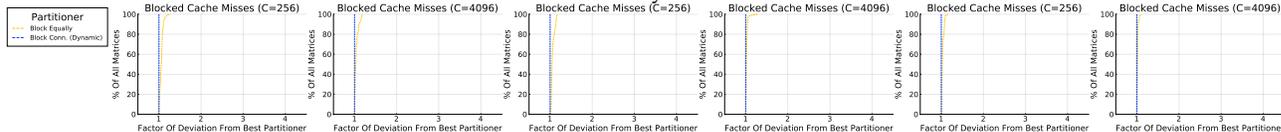


Fig. 5. Performance profiles comparing normalized modeled quality of our general (possibly noncontiguous) partitioners (Table 3) on symmetric and asymmetric test matrices (Table 2) in realistic and infinite reuse situations. Quality is measured with cost (13), using the coefficients  $c_{\text{entry}} = 1$ ,  $c_{\text{row}} = 10$ , and  $c_{\text{message}} = 100$ . For symmetric matrices, we require that the associated partitions be symmetric (we use the same partition for rows and columns). Our asymmetric test matrices also include their transposes. Some of the partitioners may reorder the matrix; setup time includes reordering operations.

## Contiguous Partitioner Runtimes

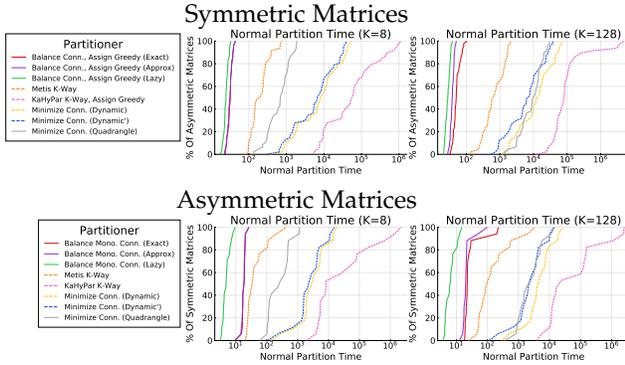


Fig. 6. Performance profiles comparing normalized partitioning runtime of our partitioners (Table 3) on symmetric and asymmetric test matrices (Table 2) across contiguous and noncontiguous, total and bottleneck regimes.

## Cache Blocking Runtimes

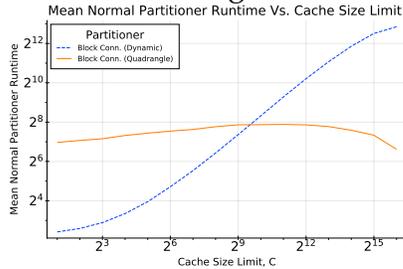


Fig. 7. Mean normalized runtime of our cache blocking algorithm as a function of the size constraint on blocks.

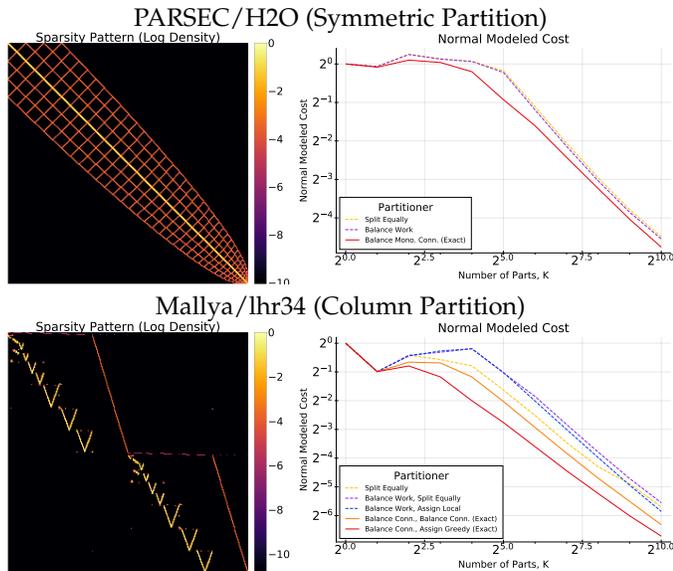


Fig. 8. Matrix sparsity patterns and the resulting partition quality of contiguous partitioners. Quality is measured with cost (13), using the coefficients  $c_{\text{entry}} = 1$ ,  $c_{\text{row}} = 10$ , and  $c_{\text{message}} = 100$ . Sparsity patterns show logarithmic density of nonzeros within each pixel in a 256 by 256 grid. Partitioners and matrices are described in Tables 2 and 3.

partitioning case, we can efficiently minimize the maximum runtime under the more accurate combined cost models.

We present a rich framework for constructing and optimizing expressive cost models for contiguous decompositions of iterative solvers. We describe a taxonomy of cost model properties, including convexity, monotonicity, and perhaps subadditivity. Using a set of efficiently computable “atoms”, we can construct complex “molecules” of cost functions which express complicated nonlinear dynamics such as cache effects, memory constraints, and communication costs. We adapt state-of-the-art load balancing and least-weight subsequence algorithms to optimize our costs. In order to efficiently compute our communication costs, we reduce our cost queries to dominance count queries and generalize a classical dominance counting algorithm to reduce construction time by increasing query time. Our new data structure can also be used to compute sparse prefix sums. We demonstrate that there are significant quality variations among partitioners in the contiguous setting, and that all of our algorithms efficiently produce high-quality partitions in practice.

There are several opportunities for future work. First, we discuss parallel implementations of our partitioners. Because our lazy partitioner relies on linear search to construct partitions, it does not appear to be amenable to parallelization. On the other hand, the bulk of the runtime for our binary-search-based partitioners (Algorithms 4 and 5) consists of dominance counting subroutines. As our dominance counters (Algorithms 1 and 2) are composed of decorated histogram sorts and one-dimensional prefix sums, parallelizing our dominance counters with similar strategies looks promising.

Our approaches might also be accelerated with algorithmic improvements. Since the dominance counters recursively decompose the index space, there may be opportunities to integrate dominance counting with binary search subroutines. Additionally, one might be able to approximate the computation by sampling edges or nonzeros.

Finally, we hope to see investigations applying these techniques to embedding-based or geometric multi-jagged partitioning approaches [1], [2], [3], [4]. Our dominance-counters may also have applications for other geometric load-balancing problems [15], [16], [17].

## REFERENCES

- [1] T. F. Chan, P. Ciarlet, and W. K. Szeto, “On the Optimality of the Median Cut Spectral Bisection Graph Partitioning Method,” *SIAM Journal on Scientific Computing*, vol. 18, no. 3, pp. 943–948, May 1997.
- [2] K. Aydin, M. Bateni, and V. Mirrokni, “Distributed Balanced Partitioning via Linear Embedding †,” *Algorithms*, vol. 12, no. 8, p. 162, Aug. 2019.
- [3] S. Acer, E. G. Boman, C. A. Glusa, and S. Rajamanickam, “Sphinx: A parallel multi-GPU graph partitioner for distributed-memory systems,” *Parallel Computing*, p. 102769, Apr. 2021.
- [4] M. Deveci, S. Rajamanickam, K. D. Devine, and Ü. V. Çatalyürek, “Multi-Jagged: A Scalable Parallel Spatial Partitioning Algorithm,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 27, no. 3, pp. 803–817, Mar. 2016.
- [5] J. Herrmann, J. Kho, B. Uçar, K. Kaya, and Ü. V. Çatalyürek, “Acyclic Partitioning of Large Directed Acyclic Graphs,” in *2017 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, May 2017, pp. 371–380.

- [6] J. Herrmann, M. Y. Özkaya, B. Uçar, K. Kaya, and Ü. V. Çatalyürek, "Multilevel Algorithms for Acyclic Partitioning of Directed Acyclic Graphs," *SIAM Journal on Scientific Computing*, vol. 41, no. 4, pp. A2117–A2145, Jan. 2019.
- [7] G. Karypis and V. Kumar, "A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs," *SIAM Journal on Scientific Computing*, vol. 20, no. 1, pp. 359–392, Jan. 1998.
- [8] Ü. V. Çatalyürek and C. Aykanat, "Hypergraph-partitioning-based decomposition for parallel sparse-matrix vector multiplication," *IEEE Transactions on Parallel and Distributed Systems*, vol. 10, no. 7, pp. 673–693, Jul. 1999.
- [9] B. Hendrickson, "Load balancing fictions, falsehoods and fallacies," *Applied Mathematical Modelling*, vol. 25, no. 2, pp. 99–108, Dec. 2000.
- [10] E. Cuthill and J. McKee, "Reducing the bandwidth of sparse symmetric matrices," in *Proceedings of the 1969 24th national conference*. Association for Computing Machinery, Aug. 1969, pp. 157–172.
- [11] A. Pothen, H. D. Simon, and K.-P. Liou, "Partitioning sparse matrices with eigenvectors of graphs," *SIAM Journal on Matrix Analysis and Applications*, vol. 11, no. 3, pp. 430–452, May 1990.
- [12] A. Grandjean and B. Uçar, "On Partitioning Two Dimensional Finite Difference Meshes for Distributed Memory Parallel Computers," in *2014 22nd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, Feb. 2014, pp. 9–16.
- [13] B. W. Kernighan, "Optimal Sequential Partitions of Graphs," *Journal of the ACM (JACM)*, vol. 18, no. 1, pp. 34–40, Jan. 1971.
- [14] B. Chazelle, "A Functional Approach to Data Structures and Its Use in Multidimensional Searching," *SIAM Journal on Computing*, vol. 17, no. 3, pp. 427–462, Jun. 1988.
- [15] D. M. Nicol, "Rectilinear Partitioning of Irregular Data Parallel Computations," *Journal of Parallel and Distributed Computing*, vol. 23, no. 2, pp. 119–134, Nov. 1994.
- [16] E. Saule, E. Ö. Baş, and Ü. V. Çatalyürek, "Load-balancing spatially located computations using rectangular partitions," *Journal of Parallel and Distributed Computing*, vol. 72, no. 10, pp. 1201–1214, Oct. 2012.
- [17] A. Yaşar, M. F. Balin, X. An, K. Sancak, and Ü. V. Çatalyürek, "On Symmetric Rectilinear Matrix Partitioning," *arXiv:2009.07735 [cs]*, Sep. 2020.
- [18] D. Eppstein, "Sequence comparison with mixed convex and concave costs," *Journal of Algorithms*, vol. 11, no. 1, pp. 85–101, Mar. 1990.
- [19] K. Akbudak and C. Aykanat, "Exploiting Locality in Sparse Matrix-Matrix Multiplication on Many-Core Architectures," *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, no. 8, pp. 2258–2271, Aug. 2017.
- [20] N. Abubaker, K. Akbudak, and C. Aykanat, "Spatiotemporal Graph and Hypergraph Partitioning Models for Sparse Matrix-Vector Multiplication on Many-Core Architectures," *IEEE Transactions on Parallel and Distributed Systems*, vol. 30, no. 2, pp. 445–458, Feb. 2019.
- [21] M. A. Iqbal, "Approximate algorithms for partitioning problems," *International Journal of Parallel Programming*, vol. 20, no. 5, pp. 341–361, Oct. 1991.
- [22] A. Pinar and C. Aykanat, "Fast optimal load balancing algorithms for 1D partitioning," *Journal of Parallel and Distributed Computing*, vol. 64, no. 8, pp. 974–996, Aug. 2004.
- [23] T. G. Kolda, "Partitioning sparse rectangular matrices for parallel processing," in *Solving Irregularly Structured Problems in Parallel*. Springer, 1998, pp. 68–79.
- [24] B. Hendrickson and T. G. Kolda, "Partitioning Rectangular and Structurally Unsymmetric Sparse Matrices for Parallel Processing," *SIAM Journal on Scientific Computing*, vol. 21, no. 6, pp. 2048–2072, Jan. 2000.
- [25] K. D. Devine, E. G. Boman, R. T. Heaphy, R. H. Bisseling, and Ü. V. Çatalyürek, "Parallel hypergraph partitioning for scientific computing," in *Proceedings 20th IEEE International Parallel Distributed Processing Symposium*, Apr. 2006, pp. 10 pp.–.
- [26] S. Rajamanickam and E. Boman, "Parallel partitioning with Zoltan: Is hypergraph partitioning worth it?" in *Contemporary Mathematics*. American Mathematical Society, Jan. 2013, vol. 588, pp. 37–52.
- [27] A. Pinar and B. Hendrickson, "Partitioning for complex objectives," in *2001 IEEE International Parallel and Distributed Processing Symposium*, Apr. 2001, pp. 1232–1237.
- [28] B. Uçar and C. Aykanat, "Encapsulating Multiple Communication-Cost Metrics in Partitioning Sparse Rectangular Matrices for Parallel Matrix-Vector Multiplies," *SIAM Journal on Scientific Computing*, vol. 25, no. 6, pp. 1837–1859, Jan. 2004.
- [29] K. Akbudak, O. Selvitopi, and C. Aykanat, "Partitioning Models for Scaling Parallel Sparse Matrix-Matrix Multiplication," *ACM Transactions on Parallel Computing*, vol. 4, no. 3, pp. 13:1–13:34, Jan. 2018.
- [30] M. Deveci, K. Kaya, B. Uçar, and Ü. V. Çatalyürek, "Hypergraph Partitioning for Multiple Communication Cost Metrics," *J. Parallel Distrib. Comput.*, vol. 77, no. C, pp. 69–83, Mar. 2015.
- [31] S. Acer, O. Selvitopi, and C. Aykanat, "Improving performance of sparse matrix dense matrix multiplication on large-scale parallel systems," *Parallel Computing*, vol. 59, pp. 71–96, Nov. 2016.
- [32] M. O. Karsavuran, S. Acer, and C. Aykanat, "Reduce Operations: Send Volume Balancing While Minimizing Latency," *IEEE Transactions on Parallel and Distributed Systems*, vol. 31, no. 6, pp. 1461–1473, Jun. 2020.
- [33] R. H. Bisseling and W. Meesen, "Communication balancing in parallel sparse matrix-vector multiplication," *ETNA. Electronic Transactions on Numerical Analysis [electronic only]*, vol. 21, pp. 47–65, 2005.
- [34] M. R. Garey, D. S. Johnson, and L. Stockmeyer, "Some simplified NP-complete graph problems," *Theoretical Computer Science*, vol. 1, no. 3, pp. 237–267, Feb. 1976.
- [35] T. Lengauer, *Combinatorial algorithms for integrated circuit layout*. John Wiley & Sons, Inc., 1990.
- [36] A. Grandjean, J. Langguth, and B. Uçar, "On Optimal and Balanced Sparse Matrix Partitioning Problems," in *2012 IEEE International Conference on Cluster Computing*, Sep. 2012, pp. 257–265.
- [37] G. Dósa, "The Tight Bound of First Fit Decreasing Bin-Packing Algorithm Is  $\text{FFD}(I) \leq 11/9\text{OPT}(I) + 6/9$ ," in *Combinatorics, Algorithms, Probabilistic and Experimental Methodologies*. Springer, 2007, pp. 1–11.
- [38] D. Nicol and D. O'Hallaron, "Improved algorithms for mapping pipelined and parallel computations," *IEEE Transactions on Computers*, vol. 40, no. 3, pp. 295–306, Mar. 1991.
- [39] L. H. Ziantz, C. C. Özturan, and B. K. Szymanski, "Run-time optimization of sparse matrix-vector multiplication on SIMD machines," in *PARLE'94 Parallel Architectures and Languages Europe*. Springer, 1994, pp. 313–322.
- [40] M. Ashraf Iqbal and S. Bokhari, "Efficient algorithms for a class of partitioning problems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 6, no. 2, pp. 170–175, Feb. 1995.
- [41] C. J. Alpert and A. B. Kahng, "Splitting an Ordering into a Partition to Minimize Diameter," *Journal of Classification*, vol. 14, no. 1, pp. 51–74, Jan. 1997.
- [42] Y. Saad, *Iterative methods for sparse linear systems*, 2nd ed. SIAM, 2003.
- [43] P. Gupta, R. Janardan, and M. Smid, "Further Results on Generalized Intersection Searching Problems: Counting, Reporting, and Dynamization," *Journal of Algorithms*, vol. 19, no. 2, pp. 282–317, Sep. 1995.
- [44] H. N. Gabow, J. L. Bentley, and R. E. Tarjan, "Scaling and related techniques for geometry problems," in *Proceedings of the sixteenth annual ACM symposium on Theory of computing*. Association for Computing Machinery, Dec. 1984, pp. 135–143.
- [45] W. Ahrens and E. G. Boman, "On Optimal Partitioning For Sparse Matrices In Variable Block Row Format," *arXiv:2005.12414 [cs]*, May 2020.
- [46] C. Alpert and A. Kahng, "Multiway partitioning via geometric embeddings, orderings, and dynamic programming," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 14, no. 11, pp. 1342–1358, Nov. 1995.
- [47] J. JaJa, C. W. Mortensen, and Q. Shi, "Space-Efficient and fast algorithms for multidimensional dominance reporting and counting," in *Proceedings of the 15th international conference on Algorithms and Computation*. Springer-Verlag, Dec. 2004, pp. 558–568.
- [48] T. M. Chan and B. T. Wilkinson, "Adaptive and Approximate Orthogonal Range Counting," *ACM Transactions on Algorithms*, vol. 12, no. 4, pp. 45:1–45:15, Sep. 2016.
- [49] B. Chazelle, "Lower bounds for orthogonal range searching: part II. The arithmetic model," *Journal of the ACM (JACM)*, vol. 37, no. 3, pp. 439–463, Jul. 1990.
- [50] S. Alstrup, G. Stolting Brodal, and T. Rauhe, "New data structures for orthogonal range searching," in *Proceedings 41st Annual Symposium on Foundations of Computer Science*, Nov. 2000, pp. 198–207.

- [51] M. Shekelyan, A. Dignös, and J. Gamper, "Sparse prefix sums: Constant-time range sum queries over sparse multidimensional data cubes," *Information Systems*, vol. 82, pp. 136–147, May 2019.
- [52] Z. Galil and R. Giancarlo, "Speeding up dynamic programming with applications to molecular biology," *Theoretical Computer Science*, vol. 64, no. 1, pp. 107–118, Apr. 1989.
- [53] M. M. Klawe and D. J. Kleitman, "An Almost Linear Time Algorithm for Generalized Matrix Searching," *SIAM Journal on Discrete Mathematics*, vol. 3, no. 1, pp. 81–97, Feb. 1990.
- [54] R. Wilber, "The concave least-weight subsequence problem revisited," *Journal of Algorithms*, vol. 9, no. 3, pp. 418–425, Sep. 1988.
- [55] M. Manguoglu, A. H. Sameh, and O. Schenk, "PSPIKE: A Parallel Hybrid Sparse Linear System Solver," in *Euro-Par 2009 Parallel Processing*. Springer, 2009, pp. 797–808.
- [56] W. M. Chan and A. George, "A linear time implementation of the reverse Cuthill-McKee algorithm," *BIT Numerical Mathematics*, vol. 20, no. 1, pp. 8–14, Mar. 1980.
- [57] M. W. Berry, B. Hendrickson, and P. Raghavan, "Sparse Matrix Reordering Schemes for Browsing Hypertext," in *The Mathematics of Numerical Analysis*. American Mathematical Society, 1996, vol. 32, pp. 99–122.
- [58] E. D. Dolan and J. J. Moré, "Benchmarking optimization software with performance profiles," *Mathematical Programming*, vol. 91, no. 2, pp. 201–213, Jan. 2002.
- [59] D. Doerfler, B. Austin, B. Cook, J. Deslippe, K. Kandalla, and P. Mendygral, "Evaluating the networking characteristics of the Cray XC-40 Intel Knights Landing-based Cori supercomputer at NERSC," *Concurrency and Computation: Practice and Experience*, vol. 30, no. 1, p. e4297, 2018.
- [60] L. Gottesbüren, M. Hamann, S. Schlag, and D. Wagner, "Advanced Flow-Based Multilevel Hypergraph Partitioning," in *18th International Symposium on Experimental Algorithms (SEA 2020)*, vol. 160. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2020, pp. 11:1–11:15.

PLACE  
PHOTO  
HERE

**Willow Ahrens** is a Department of Energy Computational Science Graduate Fellow working towards a Ph.D. in computer science at the Massachusetts Institute of Technology under the supervision of Professor Saman Amarasinghe. Willow completed her B.S. in computer science and minor in mathematics at the University of California, Berkeley. Her research interests include graph algorithms, compilers, numerical analysis, and high performance computing with applications in scientific computing.

## APPENDIX A

### LAZY PROBE PSEUDOCODE

Algorithm 6 details pseudocode for the lazy probe algorithm described in Section 5. The algorithm computes the atoms needed for all of our cost functions in primary or symmetric partitioning settings.

**Algorithm 6** (Lazy BISECT Partitioner). *Given a cost function(s)  $f$  defined on the atoms  $x_{row} = |\pi_k|$ ,  $x_{entry} = \sum_{i \in \pi_k} |v_i|$ ,  $x_{\Delta_{entry}} = \sum_{i \in \pi_k} \max(|v_i| - w_{\min}, 0)$ ,  $x_{incident} = |\cup_{i \in \pi_k} v_i|$ ,  $x_{local} = |\cup_{i \in \pi_k} v_i \cap \phi_k|$ , and  $x_{diagonal} = |\cup_{i \in \pi_k} v_i \cup \pi_k|$ , which is monotonic increasing in  $\pi_k$ , find a contiguous  $K$ -partition  $\Pi$  which minimizes*

$$c = \max_k f_k(\pi_k)$$

to a relative accuracy of  $\epsilon$  within the range  $c_{\text{low}} \leq c \leq c_{\text{high}}$ , if such a partition exists.

This algorithm differs from Algorithm 4 only in the probe function, so we only describe the new LAZYPROBE pseudocode.

We assume that  $c_{\text{low}} \geq \max_k f_k(\emptyset)$ .

**function** LAZYPROBE( $c$ )

$hst \leftarrow$  pre-allocated length  $n$  vector filled with zero

$drt \leftarrow$  pre-allocated length  $K$  vector filled with zero

$lcl \leftarrow$  pre-allocated length  $K$  vector

$i \leftarrow 1, k \leftarrow 1, x \dots \leftarrow 0$

**for**  $i' \leftarrow 1, 2, \dots, m$  **do**

$x_{row} \leftarrow x_{row} + 1$

$x_{entry} \leftarrow x_{entry} + pos_{i'+1} - pos_{i'}$

$x_{\Delta_{entry}} \leftarrow x_{\Delta_{entry}} + \max(pos_{i'+1} - pos_{i'} - w_{\min}, 0)$

**for**  $q \leftarrow pos_{i'}, pos_{i'} + 1, \dots, pos_{i'+1} - 1$  **do**

$j \leftarrow idx_q$

Set  $k'$  such that  $j \in \phi_{k'}$

**if**  $drt_{k'} < i'$  **then**

$lcl_{k'} \leftarrow 0$

**end if**

$lcl_{k'} \leftarrow lcl_{k'} + 1$

$drt_{k'} \leftarrow i'$

**if**  $hst_j < i$  **then**

$x_{incident} \leftarrow x_{incident} + 1$

**if**  $k' = k$  **then**

$x_{local} \leftarrow x_{local} + 1$

**end if**

**end if**

**if** ( $j < i$  or  $i \leq j$ ) **and**  $hst_j < i$  **then**

$x_{diagonal} \leftarrow x_{diagonal} + 1$

**end if**

$hst_j \leftarrow i'$

**end for**

**if**  $i' \leq n$  **and**  $hst_{i'} < i$  **then**

$x_{diagonal} \leftarrow x_{diagonal} + 1$

**end if**

**while**  $f(x \dots, k) > c$  **do**

**if**  $k = K$  **then**

**return** false

**end if**

$s_{k+1} \leftarrow i'$

$i \leftarrow i'$

$k \leftarrow k + 1$

$x_{row} \leftarrow 1$

$x_{entry} \leftarrow pos_{i'+1} - pos_{i'}$

$x_{\Delta_{entry}} \leftarrow \max(pos_{i'+1} - pos_{i'} - w_{\min}, 0)$

$x_{incident} \leftarrow pos_{i'+1} - pos_{i'}$

$x_{diagonal} \leftarrow pos_{i'+1} - pos_{i'}$

**if**  $i' \leq n$  **and**  $hst_{i'} < i'$  **then**

$x_{diagonal} \leftarrow x_{diagonal} + 1$

**end if**

$x_{local} \leftarrow 0$

**if**  $drt_k = i'$  **then**

$x_{local} \leftarrow x_{local} + lcl_k$

**end if**

**end while**

**end for**

**while**  $k \leq K$  **do**

$s_{k+1} \leftarrow m + 1$

**end while**

**return** true

**end function**