Load Plus Communication Balancing of Contiguous Sparse Matrix Partitions

Peter Ahrens

Abstract—We partition to parallelize multiplication of one or more dense vectors by a sparse matrix (SpMV or SpMM). We consider contiguous partitions, where the rows (or columns) of a sparse matrix with N nonzeros are split into K parts without reordering. We propose exact and approximate contiguous partitioners that minimize the maximum runtime of any processor under a diverse family of cost models that combine work and hypergraph communication terms in symmetric or asymmetric settings. This differs from traditional partitioning models which minimize total communication, or from traditional load balancing models which only balance work. One can view our algorithms as optimally rounding one-dimensional embeddings of direct K-way noncontiguous partitioning problems.
Our algorithms use linear space. Our 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_{high} and c_{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, but is faster in practice. We empirically demonstrate that all of our algorithms efficiently produce high-quality contiguous partitions.

Index Terms—Contiguous, Partitioning, Load Balancing, Communication-Avoiding, Chains-On-Chains, Dominance Counting

1 Introduction

Scientific computing applications require efficient multiplication of one or more dense vectors by a sparse matrix. These kernels are referred to as **SpMV** or **SpMM** respectively, and often arise in situations where dense vectors are multiplied by the same sparse matrix repeatedly, such as iterative linear solvers. Parallelization can increase the efficiency of these operations, and datasets are sometimes large enough that parallelization across distributed networks of processors becomes imperative due to memory constraints. 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 order of the rows may have already been carefully optimized for numerical considerations, the natural order of the rows may already be amenable to partitioning, the solver may involve operators with special structure (like stencils or block-diagonals) that would be difficult to reorder, or reordering may simply be too complicated or expensive 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 columns which is subsequently "rounded" to produce partitions. Our algorithm can be thought of as optimally rounding one-dimensional relaxations of partitioning problems. Since noncontiguous partitioning to minimize communication costs is NP-Hard [3], [4], 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.

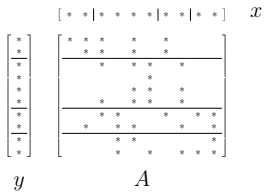

Fig. 1. Our running example matrix, together with an example symmetric partition of x and y. Nonzeros are denoted with $*$.

In many applications of SpMV or SpMM, such as iterative solvers, all processors wait for the last processor to finish. This synchronization presents a load balancing problem; our goal is to minimize the maximum time that any processor needs to compute and communicate its portion of the product. Existing contiguous partitioners optimize different objectives. Communication-aware contiguous partitioners minimize total communication [5], [6], and require quadratic time. Contiguous "Chains-On-Chains" load balancers efficiently minimize the longest-running part, but model runtime as the sum of the costs of each row in the part (typically proportional to the number of nonzeros in the row plus a constant term) [7]. Unfortunately, communication and storage effects cannot be modeled accurately as the sum of some per-row workload. Communication costs depend on the number of distinct nonlocal columns in each part; this relationship is nonlinear and depends on the sparsity structure of the matrix itself. Cache and storage effects are also nonlinear; computation in a row part is much faster when the part fits in cache, and computation is impossible if a row part does not fit in the processor's memory.

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

1.1 Contributions

In this work, we partition sparse matrices to balance the sum of work and communication on each processor. We describe families of monotonic cost models which can account for nonlocal-column-based communication costs and discontinuous cache and storage effects. We generalize state-of-the-art sublinear time algorithms for chain partitioning to optimize arbitrary monotonic cost functions (cost functions which either only increase or only decrease as more columns are added to the part). Finally, we propose efficient algorithms to evaluate our cost models. The resulting partitioners use linear space to optimize accurate cost models in linear or near-linear time. By balancing the combined cost of communication and work over each part, we can shift work burdens from extroverted processors which communicate frequently to introverted processors which communicate rarely, more efficiently utilizing computing resources [8].

Our contributions are four-fold:

- We propose families of monotonic communication-aware cost functions which target iterative solvers in distributed memory settings. 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. Our techniques can be generalized to create new families of monotonic cost functions for a wide variety of situations, accounting for factors like communication, cache, and storage effects.
- We adapt state of the art chains-on-chains partitioning algorithms (originally due to Iqbal et. al. and Nicol et. al. [9], [10] and further improved by Pinar et. al. [7]) to support arbitrary nonuniform monotonic increasing and decreasing cost functions which may be expensive to evaluate. We also propose a simpler, near-linear time version of our approximate partitioner which avoids complicated data structures and calculates cost functions on demand.
- We generalize a two-dimensional dominance counting algorithm due to Chazelle to trade construction time for query time, allowing our partitioners to run in linear time [11]. We use a slightly modified reduction from multicolored one-dimensional dominance counting to standard two-dimensional dominance counting to compute our communication terms [12].
- We evaluate our algorithms experimentally[1] on a suite of test matrices used to evaluate hypergraph partitioners [6]. We show that all of our algorithms are efficient in practice, often achieving $3\times$ improvements in quality over work-based contiguous partitioners. We also explore the performance of our algorithms on Cuthill-McKee and spectrally reordered matrices [13], [14]. Finally, we show that our partitioners are competitive with existing graph partitioning approaches when we account for the runtime of the partitioners themselves.

When load-balancing over a sublinear number of processors, we desire a data structure that can be constructed in linear time and space and queried in sublinear time. Since our dominance counting algorithm is specialized to prefix sums of sparse matrices, it may be of independent interest, especially for two-dimensional rectilinear load balancing problems with linear cost functions [10], [15], [16].

We say that f grows polynomially slower than g when f is $O(g^C)$ for some constant $C < 1$. Our dominance counting data structure uses a tree of height H, and can be constructed with H passes over the data. We suggest setting H to a small constant like 3. When partitioning the rows of an $m \times n$ matrix with N nonzeros to run on K processors, the overall runtime of our ϵ-approximate partitioner (Algorithm 2) is

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

where c_{high} and c_{low} are upper and lower bounds on the optimal cost. This runtime is sublinear when $K\log(c_{\text{high}}/(c_{\text{low}}\epsilon))$ grows polynomially slower than $N^{1-1/H} = N^{2/3}$. If the cost functions are monotonic increasing and subadditive, we can derive bounds so that $\log(c_{\text{high}}/(c_{\text{low}}\epsilon))$ is $O(\log(K/\epsilon))$. The runtime of our exact partitioner (Algorithm 3) is

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

which is sublinear when K grows polynomially slower than $N^{(1-1/H)/2} = N^{1/3}$. Up to addition of constants, the dominance counter uses at most $2m + 2N$ extra storage for our primary symmetric and simplified asymmetric cost functions, regardless of the choice of H.

We also propose a lazy version of our ϵ-approximate partitioner for our primary costs which avoids constructing a dominance counter. Up to addition of constants, this algorithm uses only $2K + n$ extra storage and runs in time

$$O\left((m + n + N)K\log\left(\frac{c_{\text{high}}}{c_{\text{low}}\epsilon}\right)\right),$$

although it is the empirically fastest serial algorithm.

In our results, we show that our communication-aware contiguous partitioners produce higher-quality partitions than simple computation-only contiguous partitioners, often producing quality improvements in excess of $3\times$. We also show that our contiguous partitioners are much faster than state-of-the-art noncontiguous approaches. We normalize our runtime to the time it takes to perform a sparse matrix-vector multiply (SpMV) on the same matrix. When $K = 2^{\lceil \log_2(m)/3 \rceil}$ and $\epsilon = 10\%$, our exact, approximate, and lazy partitioners that use dominance counters run in an average of 20.3, 18.3, and 6.95 SpMVs over our symmetric test matrices, making our algorithms practical even in situations where the matrix is not heavily reused.

2 Background

We index starting from 1. Consider the $m \times n$ **matrix** A. We refer to the entry in the i^{th} row and j^{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 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.

1. github.com/peterahrens/ChainPartitioners.jl/tree/2007.16192v3

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.

2.1 Iterative Solvers

When A is sparse, it is much faster to multiply by A (processing only the nonzero values) than it is to use direct factorization methods to solve $A \cdot x = b$. This has motivated the development of iterative solvers, which improve solutions step by step using repeated multiplication by A. We consider the case where we wish to solve for p distinct right-hand-sides, using a standard iterative solver such as the conjugate gradient method [17, Chapter 6]. The dominant cost in each step of most iterative solvers is a sparse matrix dense vector **SpMV** multiply $y = A \cdot x$ (or sparse matrix dense matrix **SpMM** multiply if $p > 1$). Often, these expensive kernels are followed by a global reduction (such as an inner product) involving the product vector y, imposing a synchronization point where all processors must wait for the last processor to receive x and multiply by A to produce y. Note that the value of x in the next step depends on the current value of y. While many iterative solvers assume symmetric matrices and multiply only by A, other solvers for asymmetric or even rectangular (for least-squares problems) matrices multiply by both A and $A^\intercal$ in each iteration [17, Chapter 8]. We must therefore distinguish between the symmetric or nonsymmetric (and possibly rectangular) cases.

2.2 Parallelism and Partitioning

We consider a distributed memory model where storage of the matrix and intermediate vectors is divided across the different processors, and communication of solver state involves sending a message between two processors. We assume that each processor can use multiple threads for asynchronous processing of communication.

A frequent parallelization strategy for SpMV and SpMM on K processors is to split the rows or columns of the matrix and corresponding vector(s) into K contiguous matching parts, where part k is stored and/or computed on processor k. When we decompose the matrix row-wise, each processor first waits to communicate input vector entries (elements of x), then computes its local portion of $y = A \cdot x$. When we decompose the matrix column-wise, each processor first computes its portion of $A \cdot x$, then communicates these partial products for the final summation to y on the destination processor. Without loss of generality, we assume a row-wise decomposition, although our cost functions apply equally to both cases. When multiplication by both A and $A^\intercal$ is required, we can simply use both interpretations of A.

A **K-partition** Π of the rows of A, assigns each row i to a single part k. The set of rows assigned to part k is denoted as π_k. Any partition Π on n elements must satisfy coverage and disjointness constraints.

$$\bigcup_k \pi_k = \{1, ..., n\}, \qquad \forall k \neq k', \pi_k \cap \pi_{k'} = \emptyset. \tag{1}$$

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 Π. 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 Φ which may differ from the output vector (row) partition Π. Since our load balancing algorithms are designed to optimize only one partition at a time, we alternate between optimizing Π or Φ, considering the other partition to be fixed. When Φ is considered fixed and our goal is only to find Π, we refer to this more restrictive problem as **primary alternate partitioning**. When Π is considered fixed and our goal is only to find Φ, 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 [18], [19]. Similar alternating approaches have been used for the related two-dimensional rectilinear partitioning regimes [10], [16].

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

$$\forall k < k', \forall (i, i') \in \pi_k \times \pi'_k, i < i'. \tag{2}$$

A contiguous partition Π can be described by its **split points** S, where $i \in \pi_k$ for $S_k \leq i < S_{k+1}$. Thus, when we parallelize our solver, processor k will be responsible for rows S_k to $S_{k+1} - 1$ of A.

2.3 Cost Models and Optimization

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.

The graph and hypergraph partitioning problems map rows of A to vertices of a graph or hypergraph, and use edges to represent communication costs. Vertices are weighted to represent computation cost or storage requirements, the weight of a part is viewed as the sum of the weights of its vertices, and the weights are constrained to be approximately balanced (balanced to a relative factor ϵ). We let W represent the vertex weights, so that w_i is the weight of vertex i. Balance constraints for graph partitioning can then be expressed as

$$\forall k, \sum_{i \in \pi_k} w_i < \frac{1+\epsilon}{K} \sum_i w_i \tag{3}$$

Graph models for symmetric partitioning of symmetric matrices typically use the **adjacency representation** $\mathrm{adj}(A)$ of a sparse matrix A. If $G = \mathrm{adj}(A)$, 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 [19]. Graph partitioning seeks to optimize

$$\arg\min_{\Pi} |\{E \cap \pi_k \times \pi_{k'} : k \neq k'\}|, \tag{4}$$

under balance constraints (3), where $G = (V, E) = \text{adj}(A)$.

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 Π, all incident vertices to e_j must belong to the same part π_k, and we can avoid communicating the input j by assigning it to processor k in our column partition Φ. In this way, we can construct a column partition Φ such that the number of cut edges in a row partition Π 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 [4]. To formalize these cost functions on a partition Π, we define $\lambda_j(A, \Pi)$ as the set of row parts which contain nonzeros in the j^{th} column. Tersely, $\lambda_j = \{k : i \in \pi_k, a_{ij} \neq 0\}$. Hypergraph partitioning with the "edge cut" metric seeks to optimize

$$\arg\min_{\Pi} |\{e_j \in E : |\lambda_j| > 1\}|, \tag{5}$$

and with the "(λ - 1) cut" metric seeks to optimize

$$\arg\min_{\Pi} \sum_{e_j \in E} |\lambda_j| - 1, \tag{6}$$

under balance constraints (3), where $G = (V, E) = \text{inc}(A)$.

While the hypergraph model better captures communication in our kernel, 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 [4], [20].

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 [19], [21]. 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 [22], [23], [24]. Other approaches modify traditional hypergraph partitioning techniques to incorporate communication balance (and other metrics) in a single phase of partitioning [25], [26], [27]. Given a row partition, Bisseling et. al. consider column partitioning to balance communication (the secondary alternate partitioning regime) [28].

While noncontiguous primary and secondary graph and hypergraph partitioning are usually NP-Hard [28], [29], [30], 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 [5], [6].

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 [31]. 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 [7] 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 Π under

$$\arg\min_{\Pi} \max_{k} \sum_{i \in \pi_k} |v_i| \tag{7}$$

where $G = (V, E) = \text{adj}(A)$.

This cost model is easily computable since the pos vector allows us to compute the total number of nonzeros in a partition from i to i' as $pos_{i'} - pos_i$ in constant time. Using pos and some additional structure in the cost function, algorithms for Chains-On-Chains partitioning 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 [32]. 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], [33].

3 Problem Formulation

The techniques used to solve Chains-On-Chains problems can be applied to other objectives in contiguous settings. In this work, we show that similar techniques apply to any objective that minimizes the modeled runtime of the longest-running processor, where the cost model on each part is **monotonic**, meaning that adding additional rows to a part will never decrease (or never increase) the cost of that part. We refer to a costliest part in a partition as a **bottleneck** processor. Put simply, monotonicity implies that adding vertices to a part can only reduce (or only increase) the cost of the other parts, which enables us to use parametric search techniques [7], [34], [35].

3.1 Monotonic Load Balancing

A cost function f on a set of vertices is defined as **monotonic increasing** if for any two sets of vertices $\pi, \pi' \subseteq V$ where $\pi \subseteq \pi'$,

$$f(\pi) \leq f(\pi'). \tag{8}$$

Our cost function f is defined as **monotonic decreasing** if for any two sets of vertices $\pi, \pi' \subseteq V$ where $\pi \subseteq \pi'$,

$$f(\pi) \geq f(\pi'). \tag{9}$$

Monotonic costs can be composed; if we have two monotonic increasing or decreasing functions f_1 and f_2, and a is a nonnegative constant, then the following functions are also monotonic increasing or decreasing, respectively:

$$\begin{aligned} g_1(\pi) &= \min(f_1(\pi), f_2(\pi)) \\ g_2(\pi) &= \max(f_1(\pi), f_2(\pi)) \\ g_3(\pi) &= f_1(\pi) + f_2(\pi) \\ g_4(\pi) &= f_1(\pi) \cdot a \end{aligned} \tag{10}$$

We are ready to state our main partitioning problem, which will provide an optimization framework for the rest of the work. By changing the cost function, our problem will address the symmetric partitioning regime and both of our nonsymmetric regimes to varying degrees of accuracy. Formally, we seek to optimize the feasible (constraint (1)) contiguous (constraint (2)) partition Π under

$$\arg\min_{\Pi} \max_{k} f_k(\pi_k) \tag{11}$$

where all f_k are monotonic increasing (constraint (8)) or all f_k are monotonic decreasing (constraint (9)).

3.2 Cost Modeling

Since the runtime of parallel programs depends on the longest-running processor, it is more accurate to partition to minimize the maximum (rather than the sum) of the modeled runtimes of each part. 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) [17, 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 $c_{\textbf{row}}$, and the per-entry computation costs (due to matrix multiplication) with the scalar $c_{\textbf{entry}}$.

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 [28]. 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 $c_{\textbf{message}}$.

3.2.1 Monotonic Nonsymmetric Cost Modeling

Since it admits a more accurate cost model, we consider the alternating partitioning regime first. This regime considers only one of the row (output) space partition Π or the column (input) space partition Φ, 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 for the alternating regime is

$$f_k(\pi_k, \phi_k) = c_{\textbf{row}}|\pi_k| + c_{\textbf{entry}} \sum_{i \in \pi_k} |v_i| + c_{\textbf{message}} \left| \left(\bigcup_{i \in \pi_k} v_i \right) \setminus \phi_k \right|. \tag{12}$$

When Φ is fixed, each f_k is a linear combination of monotonic increasing factors and is therefore monotonic increasing (adding rows to a part can only increase the work and the set of columns which potentially need communication). When Π is fixed, each f_k is monotonic decreasing (adding columns to a part can only increase the number of columns that are local, decreasing communication on that part).

While we require the opposite partition to be fixed, we will only require the partition we are currently constructing to be contiguous. Requiring both Π and Φ 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 [4]. 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 Π or Φ, we need an initial partition. We propose starting by constructing Π because this partition involves more expensive tradeoffs between work and communication. Since we would have no Φ to start with, we assume no locality, upper-bounding the cost of communication and removing Φ 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

$$f(\pi_k) = c_{\textbf{row}}|\pi_k| + c_{\textbf{entry}} \sum_{i \in \pi_k} |v_i| + c_{\textbf{message}} \left| \bigcup_{i \in \pi_k} v_i \right|. \tag{13}$$

Note that any column partition Φ will achieve or improve on the modeled cost (13).

3.2.2 Monotonic Symmetric Cost Modeling

Recall that the symmetric case asks us to produce a single partition Π 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 (11). Simply replacing Φ with Π in cost (12), we obtain

$$f(\pi_k) = c_{\textbf{row}}|\pi_k| + c_{\textbf{entry}} \sum_{i \in \pi_k} |v_i| + c_{\textbf{message}} \left| \left(\bigcup_{i \in \pi_k} v_i \right) \setminus \pi_k \right| \tag{14}$$

Unfortunately, the nonlocal communication factor $|(\bigcup_{i \in \pi_k} v_i) \setminus \pi_k|$ is not necessarily monotonic. However, notice that $|(\bigcup_{i \in \pi_k} v_i) \setminus \pi_k| + |\pi_k| = |(\bigcup_{i \in \pi_k} v_i) \cup \pi_k|$ 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 (14) in the following form,

$$f(\pi_k) = (c_{\textbf{row}} + w_{\min} \cdot c_{\textbf{entry}} - c_{\textbf{message}})|\pi_k| + c_{\textbf{entry}} \sum_{i \in \pi_k} (|v_i| - w_{\min}) + c_{\textbf{message}} \left| \left(\bigcup_{i \in \pi_k} v_i \right) \cup \pi_k \right|. \tag{15}$$

Since $|\pi_k|$, $|(\bigcup_{i \in \pi_k} v_i) \cup \pi_k|$, and $\sum_{i \in \pi_k} (|v_i| - w_{\min})$ are all monotonic, we can say cost (14) is monotonic when the coefficients on these terms are positive. We therefore require

$$c_{\textbf{row}} + w_{\min} c_{\textbf{entry}} \geq c_{\textbf{message}} \tag{16}$$

Informally, constraint (16) 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 (14) will be additively inaccurate by at most $m' \cdot w_{\min} c_{\textbf{entry}}$.

3.3 Capturing Constraints and Discontinuities

Traditional partitioners often use balance constraints to reflect per-node storage limits, rather than to balance work. Fortunately, threshold functions are monotonic, and we can use thresholds in our cost functions to reflect constraints. If our maximum storage size is $w_{\max}$, then we can define

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

Since τ is monotonic in w, we may simply take the minimum of $\tau(|\pi_k|) \cdot \infty$ (where ∞ is a suitably large value) and our original cost functions to produce a new monotonic cost which enforces balance constraints.

We can also use thresholds to capture discontinuous phenomena like cache effects. For example, one might use thresholds to switch to a more expensive model when the input or output vectors no longer fit in cache.

3.4 Bounding the Costs

While our contiguous partitioners for objective (11) will make no assumptions on the cost other than nonnegativity and monotonicity, our approximate partitioner needs an upper and lower bound on the cost to search within, and both algorithms perform better when given better bounds.

We can use **subadditivity** in increasing cost functions to provide good lower bounds on the cost. A cost function f is subadditive if for any two sets of vertices, π and π',

$$f(\pi) + f(\pi') \geq f(\pi \cup \pi') \tag{17}$$

Like monotonicity, subadditivity can be composed. Unlike monotonicity, subadditivity cannot be composed under the $\min$ function, and is not preserved under the threshold τ. Given subadditive functions f_1 and f_2 and a positive constant a, the following functions are subadditive:

$$\begin{aligned} g_1(\pi) &= \max(f_1(\pi), f_2(\pi)) \\ g_2(\pi) &= f_1(\pi) + f_2(\pi) \\ g_3(\pi) &= f_1(\pi) * a \end{aligned} \tag{18}$$

Given a monotonic increasing cost function f over a set of vertices V, it is clear that $f(V)$ is an upper bound on the cost of a K-partition. Therefore,

$$\max_k f(\pi_k) \leq f(V). \tag{19}$$

One may use subadditivity in f to create a lower bound on a K-partition. Applying the definition of subadditivity to some function f, we obtain

$$\sum_k f(\pi_k) \geq f(V),$$

and therefore,

$$\max_k f(\pi_k) \geq \frac{f(V)}{K}. \tag{20}$$

Finding lower bounds on partition costs is more difficult than finding upper bounds. Costs like (13) and (14) are subadditive and use the same cost for each part, so they obey (20). However, other costs such as (12) use different functions f_k on each part, and therefore we cannot apply equation (20).

We can work around this limitation by lower-bounding all our functions by some uniform subadditive one, then applying equation (20). 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 $\max_{k,i} f_k(i,i)$ and $\max_k f_k(1,m)$ as upper and lower bounds on the cost.

3.5 Atoms and Molecules

The monotonic "atoms", $|\pi_k|$, $\sum_{i \in \pi_k} |v_i|$, $|(\bigcup_{i \in \pi_k} v_i \setminus \phi_k)|$, $|(\bigcup_{i \in \pi_k} v_i)|$, and $|(\bigcup_{i \in \pi_k} v_i) \cup \pi_k|$, can be combined with operations like $+$, positive scalar $\cdot$, $\min$, $\max$, and τ, to produce complex "molecules" like costs (12), (13), and (14), and even new cost functions which we have yet to consider. Figure 2 displays two example partitions and the value of the corresponding atoms.

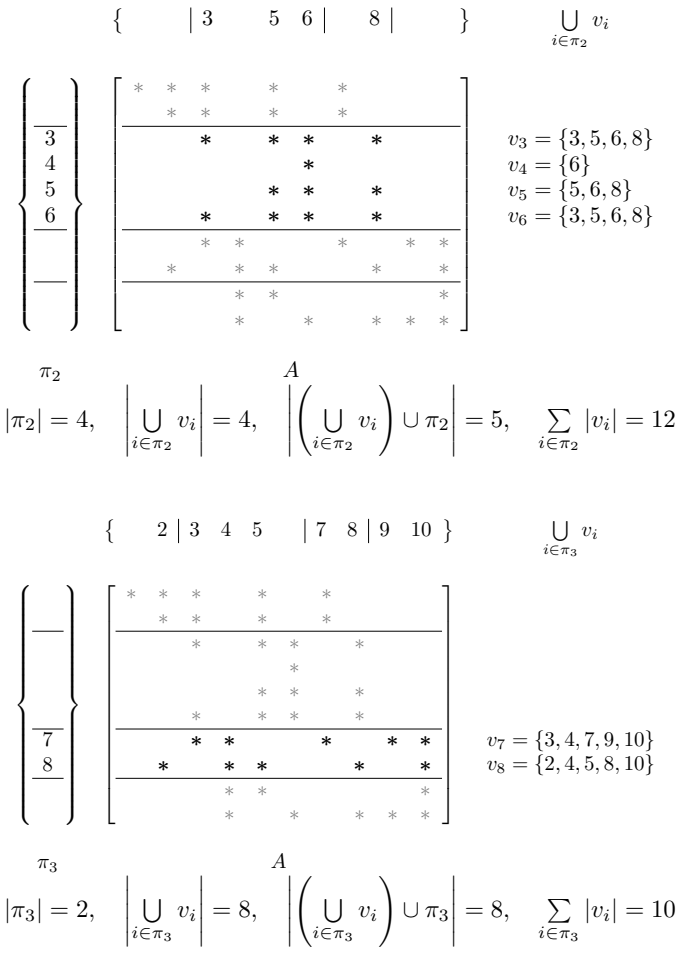

Fig. 2. Two parts in a symmetric partition of our example matrix, and the values of some of our various "atoms" for each part. Although part 2 (shown on top) contains more nonzeros than part 3 (shown on bottom), part 3 contains more distinct nonzero column locations, and will therefore require more communication. Again, nonzeros in the matrix A are denoted with $*$.

4 Monotonic Oracle Partitioners

Pinar et. al. present a multitude of algorithms for optimizing linear cost functions [7]. 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 "ϵ-BISECT+" algorithm (originally due to Iqbal et. al. [9]) and the exact "NICOL+" algorithm (originally due to Nicol et. al. [10]) 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 of the key ideas needed to understand the exact algorithm, we start with our adaptation of the "ϵ-BISECT+" partitioner, which produces a K-partition within ϵ 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. We can use such a procedure to search the space of possible costs, stopping when our lower bound on the cost is within ϵ of the upper bound. Note that if we had the optimal value of a K-partition, we could use this search procedure to find the optimal split points using only $K \log_2(m)$ evaluations of the cost function. 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 [7].

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 2. Algorithm 2 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.

Since Algorithm 2 considers at most $\log_2(c_{\text{high}}/(c_{\text{low}}\epsilon))$ candidate partition costs, the number of cost function evaluations is bounded by

$$K \log_2(m) \log_2\left(\frac{c_{\text{high}}}{c_{\text{low}}\epsilon}\right). \tag{21}$$

If our cost is subadditive and we use equation (20) to set $c_{\text{high}} = f(1, m+1)$, $c_{\text{low}} = f(1, m+1)/K$, then the number of cost function evaluations is bounded by

$$K \log_2(m) \log_2(K/\epsilon). \tag{22}$$

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 [10]. 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 [7] 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 "ϵ-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+" ([7]) for general monotonic part costs is presented in Algorithm 3.

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

$$K^2 \log_2(m)^2. \tag{23}$$

Since this number of probe sequences is sublinear in the size of the input, we will use this as our theoretical upper bound.

5 Computing The Atoms

If we can efficiently compute the value of our atoms $|\pi_k|$, $\sum_{i\in\pi_k} |v_i|$, $|(\bigcup_{i\in\pi_k} v_i \setminus \phi_k)|$, $|(\bigcup_{i\in\pi_k} v_i)|$, and $|(\bigcup_{i\in\pi_k} v_i) \cup \pi_k|$, for arbitrary contiguous parts π_k or ϕ_k, then we can efficiently evaluate any combination of these atoms, including all of the proposed cost functions in Section 3.

We start with the observation that any algorithm to efficiently evaluate costs in the primary alternating regime (when Φ is fixed) can be used to evaluate costs in the secondary alternating regime (when Π is fixed). We only need to permute the matrix so that Π is contiguous and use the primary regime algorithms to efficiently compute the value of all atoms which don't depend on Φ. All that would be left to compute is

$$\left|\bigcup_{i\in\pi_k} v_i \setminus \phi_k\right| = \left|\bigcup_{i\in\pi_k} v_i\right| - \left|\left(\bigcup_{i\in\pi_k} v_i\right) \cap \phi_k\right|,$$

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 for the boundaries of the contiguous region in our list composed of elements of ϕ_k.

The remainder of the section will focus on the symmetric and primary alternating regimes where Φ is held constant and our primary partition Π is contiguous, and therefore specified by split points S. Thus, we can compute $|\pi_k|$ as $s_{k+1} - s_k$ in constant time. Since CSR format requires pos to be a prefix sum of the number of nonzeros in each row, we can compute $\sum_{i\in\pi_k} |v_i|$ as $pos_{s_k+1} - pos_{s_k}$ in constant time.

Efficient computation of $|(\bigcup_{i\in\pi_k} v_i \setminus \phi_k)|$, $|(\bigcup_{i\in\pi_k} v_i)|$, and $|(\bigcup_{i\in\pi_k} v_i) \cup \pi_k|$ presents a challenge. These quantities concern the size of the set of distinct nonzero column locations in some row part. Prior works scan the rows of the matrix from top to bottom, using a hash table to record nonzero column locations that have been seen before, and perhaps tally when each has been seen before [2], [6], [33], [36], [37]. These approaches are efficient in serial settings when we wish to compute the cost of parts for all starting points corresponding to a fixed end point, or for all end points corresponding to a fixed start point, making them a good fit for our near-linear time algorithm in Section 7, or for dynamic programming approaches that evaluate cost functions for all possible parts in a structured, exhaustive pattern. Dynamic programming approaches often take $O(m^2+N)$ time or similar, where each of the approximately m^2 evaluations of the cost function occur in amortized constant time. However, if we wish to evaluate the cost for arbitrary start *and* end points using this approach, each evaluation of the cost function would run in linear time to the size of the considered part, which would lead to a superlinear runtime overall. To achieve a truly linear runtime, we need a data structure that supports efficient, arbitrary, evaluation of our communication terms.

Fortunately, computing these communication terms can be reduced to evaluating $|\bigcup_{i\in\pi_k} v_i|$ over special hypergraphs. In the symmetric regime, if we define A' as A with a full diagonal, and consider the corresponding hypergraph $(V', E') = \mathrm{inc}(A')$, then we have,

$$\left(\bigcup_{i\in\pi_k} v_i\right) \cup \pi_k = \bigcup_{i\in\pi_k} v'_i.$$

In the primary alternate regime when Φ is fixed, if we define $A^{(k)}$ as A where all columns other than ϕ_k are zeroed out,

$$\left|\bigcup_{i\in\pi_k} v_i \setminus \phi_k\right| = \left|\bigcup_{i\in\pi_k} v_i\right| - \left|\bigcup_{i\in\pi_k} v_i^{(k)}\right|.$$

Thus, if we can efficiently count $|\bigcup_{i\in\pi_k} v_i|$ for arbitrary A, we can compute all of our atoms. The remainder of this discussion therefore focuses on computing $|\bigcup_{i\in\pi_k} v_i|$.

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 (potentially redundant) nonzero entry in the row as a **link**. If a nonzero occurs at row i in some column and the closest following nonzero occurs at i' in that column, we call this a (i, i') link. A (i, i') link is contained in a part if both of its nonzero entries are. We can think of the second occurrence in each contained link as an overcounted nonzero in our upper bound $\sum_{i\in\pi_k} |v_i|$. Thus, by subtracting the number of links contained in our

Algorithm 1. *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 $i \le i'$, $f_k(i,i') \le c$, and $i'_{\text{low}} \le i' \le i'_{\text{high}}$. 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 ▷.*

```
function SEARCH(f_k, i, i'_low, i'_high, c)
    i'_low ← max(i, i'_low)
    while i'_low ≤ i'_high do
        i' = ⌊(i'_low + i'_high)/2⌋
        if f_k(i, i') ≤ c then
            i'_low = i' + 1                          ▷ i'_high = i' − 1
        else
            i'_high = i' − 1                         ▷ i'_low = i' + 1
        end if
    end while
    return i'_high                                   ▷ return i'_low
end function
```

Algorithm 2 (BISECT Partitioner)**.** *Given monotonic increasing cost function(s) f defined on pairs of split points, find a contiguous K-partition Π which minimizes*

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

to a relative accuracy of ϵ within the range $0 < c_{\text{low}} \le c \le c_{\text{high}}$, if such a partition exists. This is an adaptation of the "ϵ-BISECT+" algorithm by Pinar et. al. [7], which was a heuristic improvement on the algorithm proposed by Iqbal et. al. [9]. We assume that PROBE *shares scope with* BISECTPARTITION*. Changes needed for the case where all f are monotonic decreasing are marked with ▷.*

```
function BISECTPARTITION(f, m, K, c_low, c_high, ϵ)
    (s_high1, ..., s_highK+1) ← (1, m + 1, ..., m + 1)
    (s_low1, ..., s_lowK+1) ← (1, ..., 1, m + 1)
    (s_1, ..., s_K+1) ← (1, #, ..., #, m + 1)
    while c_low(1 + ϵ) < c_high do
        c ← (c_low + c_high)/2
        if PROBE(c) then
            c_high ← c
            S_high ← S                               ▷ S_low ← S
        else
            c_low ← c
            S_low ← S                                ▷ S_high ← S
        end if
    end while
    return S_high                                    ▷ return S_low
end function
function PROBE(c)
    for k = 1, 2, ..., K − 1 do
        s_k+1 ← SEARCH(f_k, s_k, s_lowk+1, s_highk+1, c)
        if s_k+1 < s_k then                          ▷ if s_k+1 > m + 1 then
            s_k+1, ..., s_K ← s_k                    ▷ s_k+1, ..., s_K ← m + 1
            return false
        end if
    end for
    return f_K(s_K, s_K+1) ≤ c
end function
```

Algorithm 3 (NICOL Partitioner)**.** *Given monotonic increasing cost function(s) f defined on pairs of split points, find a contiguous K-partition Π which minimizes*

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

within the range $c_{\text{low}} \le c \le c_{\text{high}}$, if such a partition exists. This is an adaptation of the "NICOL+" algorithm by Pinar et. al. [7], which was a heuristic improvement on the algorithm proposed by Nicol et. al. [10]. We assume that PROBEFROM *shares scope with* NICOLPARTITION*. Changes needed for the case where all f are monotonic decreasing are marked with ▷.*

```
function NICOLPARTITION(f, m, K, c_low, c_high)
    (s_high1, ..., s_highK+1) ← (1, m + 1, ..., m + 1)
    (s_low1, ..., s_lowK+1) ← (1, ..., 1, m + 1)
    (s_1, ..., s_K+1) ← (1, #, ..., #, m + 1)
    for k ← 1, 2, ..., K do
        i ← s_k
        i'_high ← s_highk+1
        i'_low ← max(s_k, s_lowk+1)
        while i'_low ≤ i'_high do
            i' ← ⌊(i'_low + i'_high)/2⌋
            c ← f_k(i, i')
            if c_low ≤ c < c_high then
                s_k+1 ← i'
                if PROBEFROM(c, k) then
                    c_high ← c
                    i'_high ← i' − 1                 ▷ i'_low ← i' + 1
                    S_high ← S                       ▷ S_low ← S
                else
                    c_low ← c
                    i'_low ← i' + 1                  ▷ i'_high ← i' − 1
                    S_low ← S                        ▷ S_high ← S
                end if
            else if c ≥ c_high then
                i'_high = i' − 1                     ▷ i'_low = i' + 1
            else
                i'_low = i' + 1                      ▷ i'_high = i' − 1
            end if
        end while
        if i'_high < s_k then                        ▷ if i'_low > m + 1 then
            break
        end if
        s_k+1 ← i'_high                              ▷ s_k+1 ← i'_low
    end for
    return S_high                                    ▷ return S_low
end function
function PROBEFROM(c, k)
    for k' = k + 1, k + 2, ..., K − 1 do
        s_k'+1 ← SEARCH(f_k', s_k', s_lowk'+1, s_highk'+1, c)
        if s_k'+1 < s'_k then                        ▷ if s_k'+1 > m + 1 then
            s_k'+1, ..., s_K ← s_k'                  ▷ s_k'+1, ..., s_K ← m + 1
            return false
        end if
    end for
    return f_K(s_K, s_K+1) ≤ c
end function
```

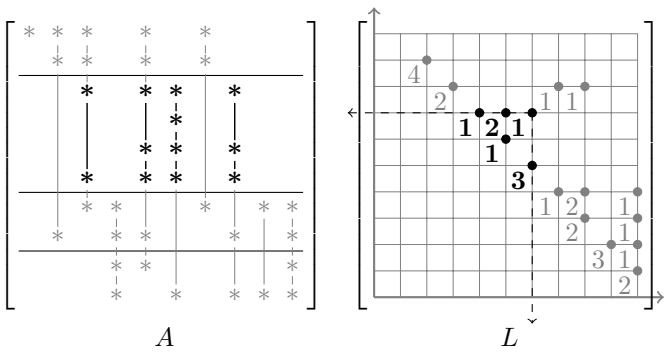

Fig. 3. Links of our example matrix A are illustrated as line segments connecting elements of A on the top, and as points (with labeled multiplicities) in the link matrix L on the bottom. Links residing entirely within part 2 are shown in bold. Part 2 contains two links starting at $i = 3$ and terminating 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.

part from the number of nonzeros in our part, we obtain the number of distinct nonzero columns in the part. If we define a matrix L where $l_{ii'}$ is equal to the number of (i, i') links in A, then we have,

$$\left| \bigcup_{i \in \pi_k} v_i \right| = \sum_{i \in \pi_k} |v_i| - \sum_{(i,i') \in \pi_k \times \pi_k} l_{ii'}. \tag{24}$$

Recall that we can compute the number of nonzeros in our part in constant time using the pos array of CSR format, and only link counting remains.

Consider each link (i, i') as a point $(-i, i')$ 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 $i'_1 \geq i'_2$. Thus, the number of links contained in some part k which extends from $s_k = i$ to $s_{k+1} = i' + 1$ equals the number of points dominated by $(-i, i')$. Figure 3 illustrates this relationship. The **dominance counting** problem in two dimensions asks for a data structure to count the number of points dominated by each query point. 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 [12]. We have re-used the values in the pos array of the link matrix in CSR format to avoid the second dominance query (these queries can be expensive in practice).

Dominance counting in two dimensions has been the subject of intensive theoretical study [11], [38], [39]. 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 original dominance counting algorithm to allow us to trade construction time for query time [11].

At this point, the reader may ask whether it is necessary to reduce our problem to the (seemingly more complicated) dominance counting. However, the problems are roughly equivalent. If we are given an arbitrary set of points on a bounded grid, we can construct a sparse matrix A with links corresponding to each point on the grid. Then we could compute dominance queries using only the values $|\bigcup_{i \in \pi_k} v_i|$ and the number of nonzeros in rows of A.

Dominance counting (and the related semigroup range sum problem [38], [40], [41]) 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. The baseline approach is to fill an $m + 1 \times m + 1$ dense matrix with the required values in the sum, taking at least $o(m^2)$ time. Improved structures tile the sparse matrix with partially dense data structures, imposing superlinear storage costs with respect to the size of the original matrix. We refer curious readers to [42] for an overview of existing approaches to the sparse prefix sum problem, with the caution that most of these works reference the size of a dense representation of the summed area table when they use words like "sublinear" and "superlinear."

5.1 Dominance Counting In A Bounded Integer Grid

Chazelle's original formulation of the 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 4 and 5.

Algorithm 4. *Construct the dominance counting data structure over N points $(i_1, j_1), ..., (i_N, j_N)$, ordered on i initially. We assume we are given the j coordinates in a vector idx. We require that $2^{Hb} > n$.*

function CONSTRUCTDOMINANCE(N, idx)
  $qos \leftarrow$ *zeroed vector of length* $n + 2$
  $qos_1 \leftarrow 1$
  $qos_{n+2} \leftarrow N + 1$
  $tmp \leftarrow$ *uninitialized vector of length* $2^b + 1$
  $cnt \leftarrow$ *zeroed 3-tensor of size* $2^b + 1 \times \lfloor N/2^{b'} \rfloor + 1 \times H$
  $byt \leftarrow$ *uninitialized vector of length* N
  for $h \leftarrow H, H - 1, ..., 1$ ***do***
    for $J \leftarrow 1, 1 + 2^{hb}, ..., n + 1$ ***do***
      Fill tmp with zeros.
      for $q \leftarrow qos_J, qos_J + 1, ..., qos_{J+2^{hb}}$ ***do***
        $d \leftarrow key_h(idx_q)$
        $tmp_{d+1} \leftarrow tmp_{d+1} + 1$
      end for
      $tmp_1 \leftarrow qos_J$
      for $d \leftarrow 1, 2, ..., 2^b$ ***do***
        $tmp_{d+1} \leftarrow tmp_d + tmp_{d+1}$
      end for
      for $q \leftarrow qos_J, qos_J + 1, ..., qos_{J+2^{hb}}$ ***do***
        $d \leftarrow key_h(idx_q)$
        $q' \leftarrow tmp_d$
        $byt_{q'} \leftarrow 2^{hb} \lfloor idx_{q'}/2^{hb} \rfloor + idx_q \bmod 2^{hb}$
        $tmp_d \leftarrow q' + 1$

```
            end for
            for d ← 1, 2, ..., 2^b do
                qos_{J+d2^{(h−1)b}} ← tmp_d
            end for
        end for
        Fill tmp with zeros.
        for Q ← 1, 1 + 2^{b'}, ..., N do
            for q ← Q, Q + 1, ..., Q + 2^{b'} do
                d ← key_h(idx_q)
                tmp_d ← tmp_d + 1
            end for
            for d ← 1, 2, ..., 2^b do
                cnt_{(d+1)Qh} ← tmp_d + cnt_{dQh}
            end for
        end for
        (idx, byt) ← (byt, idx)
    end for
    byt ← idx
    return (qos, byt, cnt)
end function
```

Algorithm 5. *Query the dominance counting data structure for the number of points dominated by (i, j).*

```
function QueryDominance(i, j)
    Δq ← pos_i − 1
    j ← j − 1
    c ← 0
    for h ← H, H − 1, ..., 1 do
        j' ← 2^{hb} ⌊j/2^{hb}⌋
        q_1 ← qos_{j'} − 1
        q_2 ← q_1 + Δq
        d ← key_h(j)
        Q_1 ← ⌊q_1/2^{b'}⌋ + 1
        Q_2 ← ⌊q_2/2^{b'}⌋ + 1
        c ← cnt_{dQ_2h} − cnt_{dQ_1h}
        Δq ← (cnt_{(d+1)Q_2h} − cnt_{dQ_2h})
        Δq ← Δq − (cnt_{(d+1)Q_1h} − cnt_{dQ_1h})
        for q ← 2^{b'}(Q_1 − 1) + 1, 2^{b'}(Q_1 − 1) + 2, ..., q_1 do
            d' ← key_h(byt_q)
            if d' < d then
                c ← c − 1
            else if d' = d then
                Δq ← Δq − 1
            end if
        end for
        for q ← 2^{b'}(Q_2 − 1) + 1, 2^{b'}(Q_2 − 1) + 2, ..., q_2 do
            d' ← key_h(byt_q)
            if d' < d then
                c ← c + 1
            else if d' = d then
                Δq ← Δq + 1
            end if
        end for
    end for
    return c
end function
```

In this section, assume we have been given N points $(i_1, j_1), ..., (i_N, j_N)$ in the range $[1, ..., m] \times [1, ..., n]$. 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 σ_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^{th} most significant group of b bits (the h^{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_j will record the starting position for points (i_q, j_q) where $j_q \geq j$. Although we only use certain entries of qos_j during our construction phase, it will be completely filled by the end of the algorithm. Note that qos is of size $n + 2$ instead of $n + 1$, as one might expect. This is because we assume that 0 is a possible value of j during construction, and we subtract one from j when we query. This is because (i_1, j_1) dominates (i_2, j_2) when $i_1 > i_2$ and $j_1 > j_2$, which is equivalent to $i_1 - 1 \geq i_2$ and $j_1 - 1 \geq j_2$.

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^{th} group of b bits in byt corresponds to the h^{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 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.

Thus, construction takes time

$$O(n + HN(1 + 2^{b-b'})). \tag{25}$$

Each query needs to traverse through H levels, each level taking $O(2^{b'})$ time, so queries require time

$$O(H2^{b'}). \tag{26}$$

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. Thus, the structure uses

$$m + n + N + NH2^{b-b'} \tag{27}$$

words (up to addition of constants).

A careful reader may notice that our complexity differs slightly from Chazelle's original complexity. Since Chazelle only considered the case where $b = 2$, each key was one bit, and Chazelle opted to store the byt array as a set of bit-packed vectors (If we think of bits as a dimension, this would be analogous to the transpose of our storage format). Because the size of a word bounded the size of the input and Chazelle assumed that we could count the number of set bits in a word in constant time, this saved another log factor at query time. Since we will choose $b > 2$ in most cases, it is likely that such bit counting instructions will not benefit us significantly even if they existed for 64-bit integers.

5.2 Balancing the Tree

In the primary alternate regime, we optimize Π under

$$\left|\bigcup_{i\in\pi_k} v_i \setminus \phi_k\right| = \left|\bigcup_{i\in\pi_k} v_i\right| - \left|\bigcup_{i\in\pi_k} v_i^{(k)}\right|.$$

where Φ is fixed. We need to create K separate dominance counters for the link matrix over each set of columns ϕ_k, splitting our links among the dominance counters. Unfortunately, the runtime and storage of our dominance counting data structure as stated depends on the number of possible columns (which, somewhat confusingly, is m in our link-counting reduction). Thus, if each processor required a dominance counter, then the total runtime and storage would be $o(mK)$. To avoid this contingency, we choose to first transform our points into "rank space" [11], [43]. Recall that our points $(i_1, j_1), ..., (i_N, j_N)$ are given to us in i-major, j-minor order. We start by resorting the points into a j-major, i-minor order $(i_{\sigma(1)}, j_{\sigma(1)}), ..., (i_{\sigma(N)}, j_{\sigma(N)})$. 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 binary search to find the transformed point at query time.

Not counting resorting, our new construction time is

$$O(N + HN(1 + 2^{b-b'})) = O(HN(1 + 2^{b-b'})) \tag{28}$$

Since we need to perform binary searches to transform query points, the new query time would be

$$O(\log(N) + H2^{b'}). \tag{29}$$

Since the i and j in the new structure are the integers $1, ..., N$, pos and qos are the identity and we no longer need to store them. However, we do need to store our orderings of i and j values, and our storage cost becomes

$$N(3 + H2^{b-b'}) \tag{30}$$

words (up to addition of constants).

In the primary alternate partitioning problem when we wish to create separate dominance counters corresponding to points in each column part ϕ_k of the matrix, we can sort the entire link matrix of all of the columns to both i-major and j-major orders in linear time with, for example, a counting sort (transposition). We can then stratify the links in the overall ordering by their corresponding column parts in linear time, producing the K requisite lists of sorted points required to count dominance within each column part ϕ_k. Since the storage and construction runtime of the new dominance counters depends multiplicatively on the number of points, and the total number of points over all of the dominance counters is N, we can construct all the dominance counters in linear time.

While transforming to rank space simplifies our algorithm, we present the non-transformed (index space) counter first because it combines the act of sorting the points with the act of constructing the dominance counter, which is likely to be practically faster than the transformed version which essentially sorts the points twice. In our results, we do not use any partitioners which require rank transformation, although it is implemented in the codebase.

5.3 Sparse Prefix Sums

Like Chazelle's algorithm, our dominance counting algorithm can be extended to compute prefix sums of non-Boolean values over a bounded integer grid. At each level, we simply need to store the values associated with the current ordering of points. When we query the structure, in the same way we count c, the number of points in our bucket where $d' < d$, we would also need to sum the values associated with these points. In order to maintain the same asymptotic runtime, this necessitates the use of an array similar to cnt which records a prefix sum of the values of previous points (rather than their count) in the ordering at each level. These modifications would not increase the runtime of construction or queries, but would increase the storage by a factor of H, so that the storage requirement for a sparse prefix sum would be (up to addition of a constant),

$$m + n + N(1 + H + H2^{b-b'}). \tag{31}$$

We also implemented this algorithm in our codebase.

TABLE 1
Relevant quantities involved in runtime tradeoffs. Recall that the link matrix is an $m \times m$ matrix with N nonzeros.

(21) Algorithm 2 Max Queries	$K \log_2(m) \log_2\left(\frac{c_{\text{high}}}{c_{\text{low}}\epsilon}\right)$
(22) Algorithm 2 (Subadditive Cost)	$K \log_2(m) \log_2(K/\epsilon)$
(23) Algorithm 3 Max Queries	$K^2 \log_2(m)^2$
(25) Dominance Construct Time (Indices)	$O(m + HN(1 + 2^{b-b'}))$
(28) Dominance Construct Time (Ranks)	$O(HN(1 + 2^{b-b'}))$
(26) Dominance Query Time (Indices)	$O(H2^{b'})$
(29) Dominance Query Time (Ranks)	$O(\log(N) + H2^{b'})$
(27) Dominance Storage (Indices)	$2m + N + NH2^{b-b'}$
(30) Dominance Storage (Ranks)	$N(3 + H2^{b-b'})$

6 Partitioning With Dominance Counters

Having described our partitioners and routines to compute our cost functions, we can combine the two and evaluate the runtime of the overall algorithm. Our analysis balances the runtime of constructing the dominance counting data structure and completing the queries.

As we explain in the beginning of Section 5, all of our cost functions can be evaluated with at most two dominance counting data structures. In the primary alternate case (the most expensive case), we need to construct two dominance counters over N points each. One of these dominance counters is transformed to rank space, the other does not need to be. While our analysis will assume both counters are used, only the unranked structure is needed to optimize costs (14) or (13), the common cost functions. Table 1 describes the relevant quantities.

To simplify our runtime analysis, we assume that $N \geq m$ and $N \geq n$. Combining the construction time, number of queries, and time per query (assuming that the query time is dominated by dominance counting), our approximate partitioner (Algorithm 2) runs in time

$$O\Big(HN(1 + 2^{b-b'}) + K \log(N) \log\left(\frac{c_{\text{high}}}{c_{\text{low}}\epsilon}\right)\left(\log(N) + H2^{b'}\right)\Big), \tag{32}$$

and our exact partitioner (Algorithm 3) runs in time

$$O\Big(HN(1 + 2^{b-b'}) + K^2 \log(N)^2\left(\log(N) + H2^{b'}\right)\Big). \tag{33}$$

There are several ways to set H, b, and b'. Chazelle proposed an algorithm in rank space which set $H = \log_2(N)$, $b = 2$, and $b' = \log_2(\log_2(N))$. While storage would be linear and query time would be polylogarithmic, constructing a dominance counter with these settings would require $\log_2(N)$ passes, an onerous 10 passes over the data for $1,024$ nonzeros, and 20 passes for $1,048,576$ nonzeros.

Thus, we recommend 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 $2^{Hb} \geq m$ in index space, and subject to $2^{Hb} \geq N$ in rank space. 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. With these settings, our approximate partitioner (Algorithm 2) runs in time

$$O\left(HN + K \log(N) \log\left(\frac{c_{\text{high}}}{c_{\text{low}}\epsilon}\right) H^2 N^{1/H}\right), \tag{34}$$

and our exact partitioner (Algorithm 3) runs in time

$$O\left(HN + K^2 \log(N)^2 H^2 N^{1/H}\right). \tag{35}$$

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.

7 Lazy Partitioning

While our experiments show our exact, linear-time contiguous partitioner to be competitive in practice, we also note that by sacrificing our strictly linear runtime guarantees, we can create a simpler approximate bisection partitioner which is even faster in serial settings. If we swap our binary search procedure for a linear search, then the resulting structured access pattern to our cost function allows us to compute our cost function on demand. Given a target cost c in Algorithm 2, we can construct a partition that attains that cost (if it exists) by simply adding rows to the current part until the cost exceeds the target. Then we move on to the next part. Since we add rows to the partition one by one from top to bottom, we can calculate the various "atoms" that make up our cost function on-the-fly.

We focus on primary alternate and symmetric partitioning, since secondary alternate partitioning doesn't require dominance counting. Calculating atoms like $|\pi_k|$ and $\sum_{i \in \pi_k} |v_i|$ in linear time is trivial. We can calculate $|\cup_{i \in \pi_k} v_i|$, $|\cup_{i \in \pi_k} v_i \cap \phi_k|$, and $|\cup_{i \in \pi_k} v_i \cup \pi_k|$ in linear time by updating a hash of which nonlocal columns we have already seen in the current part as we add each new row. To avoid reinitializing our hash when we move to a new part, we instead record the last time each column was seen, a trick used in several similar algorithms [6], [33], [36], [44]. Because we need to be able to update the value of atoms in constant time when we switch to a new part, we also compute $v_i \cap \phi_{k'}$ for all $k' > k$ as we process each new row. Overall, runtime and storage of our lazy probe is linear in the number of rows, columns and nonzeros of the matrix. Since Algorithm 2 performs $\log(c_{\text{high}}/(c_{\text{low}}\epsilon))$ probes, the lazy version runs in time

$$O\left((m + n + N) \log\left(c_{\text{high}}/(c_{\text{low}}\epsilon)\right)\right). \tag{36}$$

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

Symmetric Matrices

Group / Matrix	$m \times n$	N
Boeing/ ct20stif	52.3K×52.3K	2.7M
Boeing/ pwtk	218K×218K	11.6M
Chen/ pkustk03	63.3K×63.3K	3.13M
Chen/ pkustk14	152K×152K	14.8M
Cunningham/ qa8fk	66.1K×66.1K	1.66M
DIMACS10/ delaunay_n20	1.05M×1.05M	6.29M
Gupta/ gupta2	62.1K×62.1K	4.25M
HB/ bcsstk30	28.9K×28.9K	2.04M
HB/ bcsstk32	44.6K×44.6K	2.01M
HB/ dwt_607	607×607	5.13K
HB/ sherman3	5K×5K	20K
Hamm/ bcircuit	68.9K×68.9K	376K
Mulvey/ finan512	74.8K×74.8K	597K
Nasa/ nasasrb	54.9K×54.9K	2.68M
PARSEC/ H2O	67K×67K	2.22M
Rothberg/ cfd1	70.7K×70.7K	1.83M
Schenk_IBMNA/ c-64	51K×51K	718K
Schmid/ thermal2	1.23M×1.23M	8.58M
Simon/ venkat01	62.4K×62.4K	1.72M
TKK/ smt	25.7K×25.7K	3.75M

Asymmetric Matrices

Group / Matrix	$m \times n$	N
ATandT/ onetone1	36.1K×36.1K	341K
ATandT/ onetone2	36.1K×36.1K	228K
Averous/ epb3	84.6K×84.6K	464K
Bomhof/ circuit_4	80.2K×80.2K	308K
Grund/ bayer01	57.7K×57.7K	278K
HB/ gemat11	4.93K×4.93K	33.2K
Hollinger/ g7jac180	53.4K×53.4K	747K
Hollinger/ mark3jac140sc	64.1K×64.1K	400K
LPnetlib/ lp_cre_b	9.65K×77.1K	261K
LPnetlib/ lp_cre_d	8.93K×73.9K	247K
LPnetlib/ lp_ken_11	14.7K×21.3K	49.1K
LPnetlib/ lp_ken_13	28.6K×42.7K	97.2K
LPnetlib/ lpi_gosh	3.79K×13.5K	100K
Mallya/ lhr07	7.34K×7.34K	157K
Mallya/ lhr14	14.3K×14.3K	308K
Mallya/ lhr17	17.6K×17.6K	382K
Mallya/ lhr34	35.2K×35.2K	764K
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
Qaplib/ lp_nug30	52.3K×379K	1.57M
Shyy/ shyy161	76.5K×76.5K	330K

8 Results

We implemented our exact, approximate, and lazy approximate partitioners in Julia and compared their performance on a set of sparse matrices. We include the matrices used by Pinar and Aykanat to evaluate solutions to the Chains-On-Chains problem [7] (minimizing the maximum work without reordering), the matrices used by Çatalyurek and Aykanat to evaluate multilevel hypergraph partitioners (minimizing the sum of communication) [4], and the matrices used by Grandjean and Uçar to evaluate contiguous hypergraph partitioners (minimizing the sum of communication without reordering) [6]. 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. A summary of the test suite is provided 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) [13] and spectral [14] orderings, which have a history of application to partitioning [18], [45]. The RCM ordering uses a breadth-first traversal of the graph, prioritizing lower degree vertices first. We follow the linear-time implementation described in [46]. 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), 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. [47].

Our results are presented as performance profiles, allowing us to compare our partitioners on the entire dataset at once [48]. The performance of each partitioner is measured as the relative deviation from the best performing partitioner on each matrix individually. We then display, for each partitioner, the fraction of test instances that achieve each target deviation from the best partitioner.

Although some of our partitioners may optimize different cost models, we always measure the quality of the partitions produced with cost (12). Since the coefficients should depend on which linear solver is to be partitioned for which parallel machine, we instead use coefficients 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 $c_{\textbf{entry}} = 1$. 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; $c_{\textbf{row}} = 10$. The peak MPI bandwidth on Cori, a supercomputer at the National Energy Research Scientific Computing Center, has been measured at approximately 8 GB/s [49]. 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 $c_{\textbf{message}} = 100$.

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 use Julia 1.5.1 and normalize against the Julia Standard Library SpMV, since 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 use the average quality over 100 trials. Some partitioners, such as KaHyPar, have superlinear asymptotic runtime and take a long time to partition big matrices. We therefore instituted a cutoff of 100 seconds per trial. While all of our contiguous partitioners were able to complete within our cutoff, some executions of KaHyPar and spectral reordering were not.

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 same matrix. We normalize the modeled cost of a partition against the modeled cost of the serial partition (the partition with all rows in one part). Since both the partitioner runtime and partition quality are normalized against the same kernel, we can model the combined runtime by multiplying our normalized partition quality by an expected number of solver iterations and adding the normalized partitioning

TABLE 3
Partitioners we consider in our results, described under their shorthand labels. Partitioners proposed in this work are underlined.

Split Equally (Symmetric): Assigns an equal number of contiguous rows to each part.
Split Equally: Assigns an equal number of contiguous rows and columns to each part.
Balance Work (Symmetric): Use Algorithm 3 to balance a contiguous row partition under $c_{\text{row}}|\pi_k| + c_{\text{entry}} \sum_{i \in \pi_k} |v_i|$.
Balance Work, Assign Local: Balance rows as above, then assign columns to the part of a randomly selected incident row.
Balance Comm (Symmetric): Use Algorithm 2 with the lazy probe described in Section 7 to 10%-approximately balance a contiguous row partition under cost (15).
Balance Comm (Symmetric)(Exact): Use Algorithm 3 and a dominance counter with $H = 3$ to balance a contiguous row partition under cost (15).
Balance Comm, Balance Comm: Use Algorithm 2 with the lazy probe described in Section 7 to 10%-approximately balance a contiguous row partition under cost (13), then form the binary search structure described at the beginning of Section 5 and use Algorithm 2 to 10%-approximately balance a contiguous column partition under cost (12).
Balance Comm, Assign Greedy: Partition rows as above, then in a random order, assign each column to the currently most expensive part under cost (12).
Balance Comm, Balance Comm (Exact): Use Algorithm 3 and a dominance counter with $H = 3$ to balance a contiguous row partition under cost (13), then form the binary search structure described at the beginning of Section 5 and use Algorithm 3 to balance a contiguous column partition under cost (12).
Balance Comm, Assign Greedy (Exact): Partition rows as above, then in a random order, assign each column to the currently most expensive part under cost (12).
Metis (Symmetric): Use direct, K-way Metis to optimize total edge cut with a maximum of 10% nonzero imbalance [3].
Metis, Assign Greedy: Partition rows using symmetrized bipartite representation of matrix as above, then in a random order, assign each column to the currently most expensive part under cost (12).
KaHyPar (Symmetric): Use direct, K-way KaHyPar to optimize $\lambda - 1$ edge cut with a maximum of 10% nonzero imbalance [50].
KaHyPar, Assign Greedy: Partition rows as above, then in a random order, assign each column to the currently most expensive part under cost (12).
Spectral, $*$: Spectrally reorder, then apply a contiguous partitioner.
CuthillMcKee, $*$: Reorder with Cuthill-McKee, then apply a contiguous partitioner.

time. This allows us to evaluate partitioning time in context.

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. In general, we found that constructing a dominance counting data structure and using linear-time Algorithms 2 or 3 was efficient enough to be practical. However, for approximate partitioning, we chose the lazy variation described in Section 7 since it was empirically faster in almost all cases, even though it is asymptotically slower by a factor of $\log(c_{\text{high}}/(c_{\text{low}}\epsilon))$. When $H = 3$, the exact algorithm is expected to run in linear time when K grows slower than $m^{1/3}$. When $K = 2$, on average over our symmetric matrices, the exact, approximate, and lazy approximate algorithms were able to partition under cost (15) in 16.3, 16.4, and 3.85 SpMVs, respectively. When $K = 2^{\lceil \log_2(m)/3 \rceil}$, they used 20.3, 18.3, and 6.95 SpMVs. When $K = 2^{\lceil \log_2(m)/2 \rceil}$, well outside of the range of our linear runtime guarantees, they used 72.4, 24.2, and 8.19 SpMVs, respectively.

Figure 4 compares the performance of contiguous partitioners on our symmetric and asymmetric matrices. In the symmetric cases, partitioning the natural order using our communication-aware cost models produced higher-quality results than splitting equally or balancing just the computational load, sometimes by more than $2\times$, even accounting for partition time in realistic reuse situations. These benefits were less pronounced when partitioning Cuthill-McKee or spectrally reordered symmetric matrices, so balancing the work may also be competitive in low-reuse symmetric bandwidth-reduced situations.

Communication-aware partitioning was especially effective on asymmetric matrices. On natural orderings, communication-aware splitting was frequently as many as $3\times$ better than the work-based splittings, even after accounting for partitioning time. Note that the "Balance Comm, Assign Greedy" and "Balance Work, Assign Local" partitioners allow the column partition to be noncontiguous. Noncontiguity appears to be beneficial for secondary partitioning, since "Balance Comm, Balance Comm" constructs the optimal contiguous secondary partition, but does not perform nearly as well as the greedy noncontiguous strategy, especially on naturally ordered matrices. This is to be expected because in naturally ordered matrices, there is no guarantee that the nonzero columns incident to one part will generally occur before the nonzero columns incident to the second. However, forcing column partitions to be contiguous has much less of an impact on our reordered matrices because these reorderings attempt to reduce the profile of the matrix, clustering nonzeros near the diagonal. In a similar trend to the symmetric case, balancing the work is somewhat competitive on reordered matrices when we include partitioning time, as these reordered sparsity patterns are more uniform across rows and columns.

Figure 5 compares symmetric and asymmetric partitioners 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 reuse.

In symmetric cases, these results show that Cuthill-McKee reordering before partitioning is a good strategy when the number of parts is low, but that it takes too long to compute the Fiedler vector for spectral reordering. The KaHyPar hypergraph partitioner produced the best quality solutions, but took far too long to partition.

Asymmetric problems were less amenable to reordering-based approaches. Additionally, we found that Metis was less applicable to the asymmetric case for large numbers of parts. When the number of parts was high, communication-aware contiguous partitioning without reordering produced similar quality partitions to those of Metis in far less time. Seeing contiguous partitioning become so effective when there are many parts should come as no surprise; as the number of parts increases, the natural structure of the matrix becomes easier to utilize because the parts can be smaller. Again, when the long-running KaHyPar was able to finish in the allotted time, it produced the highest quality partitions. However, for most situations with realistic partition reuse, contiguous partitioning is shown to be the best option.

Figure 6 is meant to help give intuition for how the

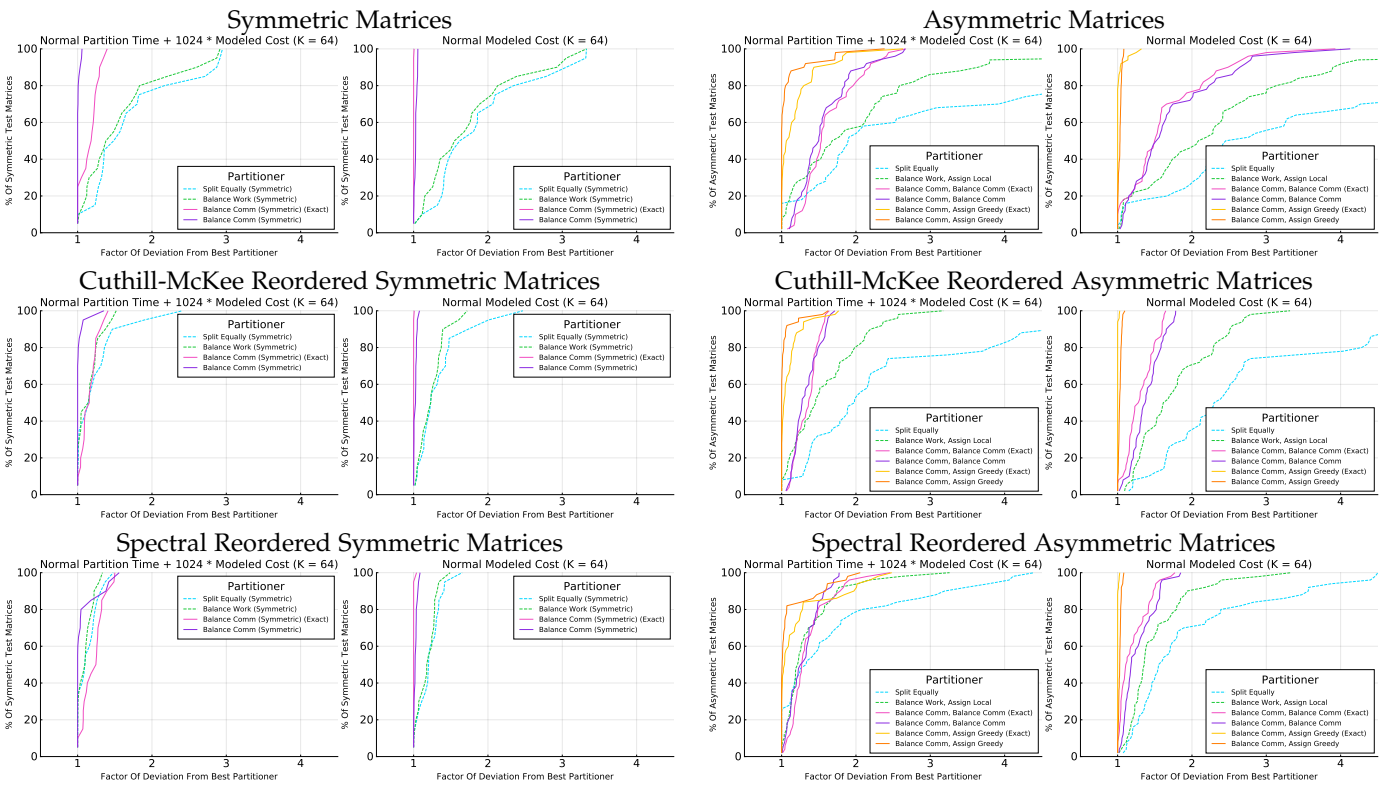

Fig. 4. Performance profiles comparing normalized modeled quality of our contiguous partitioners (Table 3) on symmetric and asymmetric test matrices (Table 2) in realistic and infinite reuse situations. Quality is measured with cost (12), using the coefficients $c_{\textbf{entry}} = 1$, $c_{\textbf{row}} = 10$, and $c_{\textbf{message}} = 100$. 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. Since our partitioners do not reorder matrices, we measure their performance both on the natural ordering and on RCM and spectral orderings. Partitioning time does not include the time to produce the orderings.

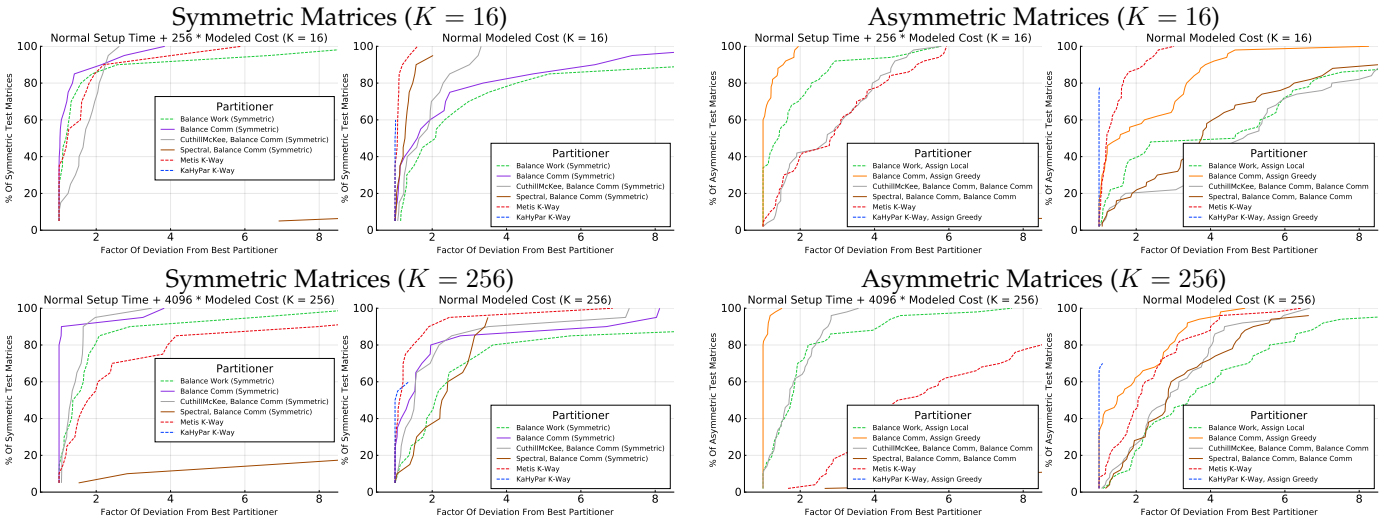

Fig. 5. Performance profiles comparing normalized modeled quality of our general (possibly noncontiguous) partitioners (Table 3) on symmetric and asymmetric test matrices (Table 2) in realistic and infinite reuse situations. Quality is measured with cost (12), using the coefficients $c_{\textbf{entry}} = 1$, $c_{\textbf{row}} = 10$, and $c_{\textbf{message}} = 100$. 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.

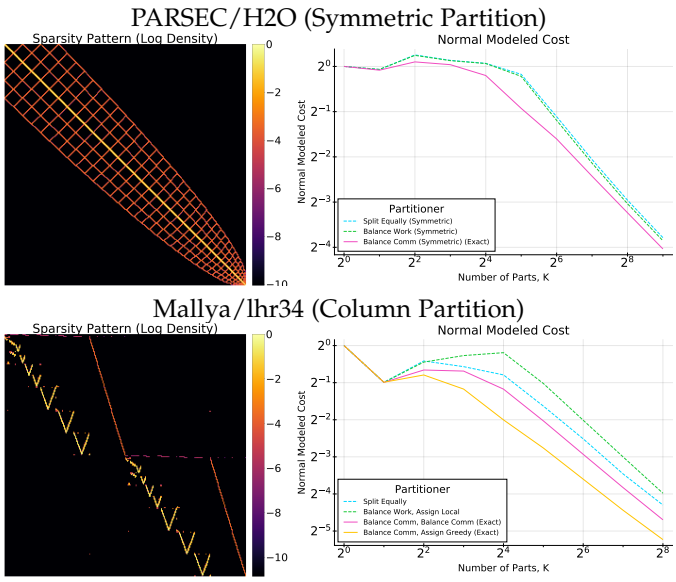

Fig. 6. Matrix sparsity patterns and the resulting partition quality of contiguous partitioners. Quality is measured with cost (12), using the coefficients $c_{\textbf{entry}} = 1$, $c_{\textbf{row}} = 10$, and $c_{\textbf{message}} = 100$. Sparsity patterns show logarithmic density of nonzeros within each pixel in a 256 by 256 grid. Partitioners and matrices are described in Tables 2 and 3.

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 2×2 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.

9 Conclusion

Traditional graph partitioning approaches have two main limitations. The cost models are highly simplified, and the problems are NP-Hard. While the ordering of the rows and columns of a matrix does not affect the meaning of the described linear operation, it often 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 time and space, provably optimizing cost models which are closer 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 both work and communication factors. [8], [21], [23], [28]. We show that, in the contiguous partitioning case, we can efficiently minimize the maximum runtime under cost models which include communication factors.

We present a rich framework for constructing and optimizing expressive cost models for contiguous decompositions of iterative solvers. Our only constraints are 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 algorithms to optimize monotonic increasing or decreasing costs. In order to efficiently compute our communication costs so the whole algorithm uses linear time and space, we 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 also provide a simpler, near-linear time algorithm, and demonstrate that all of our algorithms efficiently produce high-quality partitions in practice.

Future work includes 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 2 and 3) consists of dominance counting subroutines. As our dominance counters (Algorithms 4 and 5) are composed of decorated histogram sorts and one-dimensional prefix sums, parallelizing our dominance counters with similar strategies looks promising.

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] 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.

[4] U. 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.

[5] B. W. Kernighan, "Optimal Sequential Partitions of Graphs," *Journal of the ACM (JACM)*, vol. 18, no. 1, pp. 34–40, Jan. 1971.

[6] 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.

[7] A. Pınar 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.

[8] B. Hendrickson, "Load balancing fictions, falsehoods and fallacies," *Applied Mathematical Modelling*, vol. 25, no. 2, pp. 99–108, Dec. 2000.

[9] M. A. Iqbal, "Approximate algorithms for partitioning problems," *International Journal of Parallel Programming*, vol. 20, no. 5, pp. 341–361, Oct. 1991.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] 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.

[16] A. Yaşar, M. F. Balin, X. An, K. Sancak, and Ü. V. Çatalyürek, "On Symmetric Rectilinear Matrix Partitioning," *arXiv:2009.07735 [cs]*, Sep. 2020.

[17] Y. Saad, *Iterative methods for sparse linear systems*, 2nd ed. SIAM, 2003.

[18] T. G. Kolda, "Partitioning sparse rectangular matrices for parallel processing," in *Solving Irregularly Structured Problems in Parallel*. Springer, 1998, pp. 68–79.

[19] 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.

[20] K. D. Devine, E. G. Boman, R. T. Heaphy, R. H. Bisseling, and U. V. Çatalyürek, "Parallel hypergraph partitioning for scientific computing," in *Proceedings 20th IEEE International Parallel Distributed Processing Symposium*, Apr. 2006.

[21] 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.

[22] A. Pinar and B. Hendrickson, "Partitioning for complex objectives," in *2001 IEEE International Parallel and Distributed Processing Symposium*, Apr. 2001, pp. 1232–1237.

[23] 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.

[24] 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.

[25] M. Deveci, K. Kaya, B. Uçar, and U. V. Çatalyürek, "Hypergraph Partitioning for Multiple Communication Cost Metrics," *J. Parallel Distrib. Comput.*, vol. 77, no. C, pp. 69–83, Mar. 2015.

[26] 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.

[27] 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.

[28] R. H. Bisseling and W. Meesen, "Communication balancing in parallel sparse matrix-vector multiplication." *ETNA. Electronic Transactions on Numerical Analysis*, vol. 21, pp. 47–65, 2005.

[29] 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.

[30] T. Lengauer, *Combinatorial algorithms for integrated circuit layout*. John Wiley & Sons, Inc., 1990.

[31] G. Dósa, "The Tight Bound of First Fit Decreasing Bin-Packing Algorithm Is FFD(I) $\leq$ 11/9OPT(I) + 6/9," in *Combinatorics, Algorithms, Probabilistic and Experimental Methodologies*. Springer, 2007.

[32] 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.

[33] 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.

[34] M. A. Iqbal and S. H. 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.

[35] 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.

[36] P. Ahrens and E. G. Boman, "On Optimal Partitioning For Sparse Matrices In Variable Block Row Format," *arXiv:2005.12414 [cs]*, May 2020.

[37] 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.

[38] 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.

[39] 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.

[40] 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.

[41] S. Alstrup, G. S. 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.

[42] 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.

[43] 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.

[44] B. Jackson, J. D. Scargle, D. Barnes, S. Arabhi, A. Alt, P. Gioumousis, E. Gwin, P. Sangtrakulcharoen, L. Tan, and T. T. Tsai, "An Algorithm for Optimal Partitioning of Data on an Interval," *IEEE Signal Processing Letters*, vol. 12, no. 2, pp. 105–108, Feb. 2005.

[45] 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.

[46] 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.

[47] 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.

[48] E. D. Dolan and J. J. Moré, "Benchmarking optimization software with performance profiles," *Mathematical Programming*, vol. 91, no. 2, pp. 201–213, Jan. 2002.

[49] 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.

[50] L. Gottesbüren, M. Hamann, S. Schlag, and D. Wagner, "Advanced Flow-Based Multilevel Hypergraph Partitioning," in *18th International Symposium on Experimental Algorithms*, vol. 160. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2020, pp. 11:1–11:15.

Peter Ahrens is 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. Peter completed his B.S. in computer science and minor in mathematics at the University of California, Berkeley. His 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 7. 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 $x_{\textbf{\textit{row}}} = |\pi_k|$, $x_{\textbf{\textit{entry}}} = \sum_{i\in\pi_k} |v_i|$, $x_{\Delta_{\textbf{\textit{entry}}}} = \sum_{i\in\pi_k} \max(|v_i| - w_{\min}, 0)$, $x_{\textbf{\textit{incident}}} = |\cup_{i\in\pi_k} v_i|$, $x_{\textbf{\textit{local}}} = |\cup_{i\in\pi_k} v_i \cap \phi_k|$, and $x_{\textbf{\textit{diagonal}}} = |\cup_{i\in\pi_k} v_i \cup \pi_k|$, which is monotonic increasing in π_k, find a contiguous K-partition Π which minimizes*

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

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

This algorithm differs from Algorithm 2 only in the probe function, so we only describe the new LAZYPROBE *pseudocode. We assume that $c_{\text{low}} \ge \max_k f_k(\emptyset)$.*

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*
  $i \leftarrow 1$, $k \leftarrow 1$, $x... \leftarrow 0$
  for $i' \leftarrow 1, 2, ..., m$ ***do***
    $x_{\textbf{\textit{row}}} \leftarrow x_{\textbf{\textit{row}}} + 1$
    $x_{\textbf{\textit{entry}}} \leftarrow x_{\textbf{\textit{entry}}} + pos_{i'+1} - pos_{i'}$
    $x_{\Delta_{\textbf{\textit{entry}}}} \leftarrow x_{\Delta_{\textbf{\textit{entry}}}} + \max(pos_{i'+1} - pos_{i'} - w_{\min}, 0)$
    for $q \leftarrow pos_{i'}, pos_{i'} + 1, ..., pos_{i'+1} - 1$ ***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***
        $x_{\textbf{\textit{incident}}} \leftarrow x_{\textbf{\textit{incident}}} + 1$
        if $k' = k$ ***then***
          $x_{\textbf{\textit{local}}} \leftarrow x_{\textbf{\textit{local}}} + 1$
        end if
      end if
      if $(j < i$ ***or*** $i \le j)$ ***and*** $hst_j < i$ ***then***
        $x_{\textbf{\textit{diagonal}}} \leftarrow x_{\textbf{\textit{diagonal}}} + 1$
      end if
      $hst_j \leftarrow i'$
    end for
    if $i' \le n$ ***and*** $hst_{i'} < i$ ***then***
      $x_{\textbf{\textit{diagonal}}} \leftarrow x_{\textbf{\textit{diagonal}}} + 1$
    end if
    while $f(x..., k) > c$ ***do***
      if $k = K$ ***then***
        return false
      end if
      $s_{k+1} \leftarrow i'$
      $i \leftarrow i'$
      $k \leftarrow k + 1$
      $x_{\textbf{\textit{row}}} \leftarrow 1$
      $x_{\textbf{\textit{entry}}} \leftarrow pos_{i'+1} - pos'_i$
      $x_{\Delta_{\textbf{\textit{entry}}}} \leftarrow \max(pos_{i'+1} - pos'_i - w_{\min}, 0)$
      $x_{\textbf{\textit{incident}}} \leftarrow pos_{i'+1} - pos'_i$
      $x_{\textbf{\textit{diagonal}}} \leftarrow pos_{i'+1} - pos'_i$
      if $i' \le n$ ***and*** $hst_{i'} < i'$ ***then***
        $x_{\textbf{\textit{diagonal}}} \leftarrow x_{\textbf{\textit{diagonal}}} + 1$
      end if
      $x_{\textbf{\textit{local}}} \leftarrow 0$
      if $drt_k = i'$ ***then***
        $x_{\textbf{\textit{local}}} \leftarrow x_{\textbf{\textit{local}}} + lcl_k$
      end if
    end while
  end for
  while $k \le K$ ***do***
    $s_{k+1} \leftarrow m + 1$
  end while
  return true
end function

APPENDIX B
A WORD ON TOTAL COMMUNICATION VOLUME

Our suggested initial cost (13) for primary alternate partitioning simply assumes that no columns are local to any part. If we included only this communication term, our partitioner would find a partition Π to minimize

$$\max_k \left| \bigcup_{i\in\pi_k} v_i \right|$$

At a first glance, this appears to disagree with the $(\lambda - 1)$ hypergraph communication objective (6), which minimizes the total number of nonlocal columns (where we must also optimize Φ)

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

However, it is worth pointing out that the total number of nonlocal columns differs from the total number of incident columns by the constant $|E|$, and thus, $(\lambda - 1)$ hypergraph partitioning also minimizes

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

where the last equality comes from counting incidences over parts rather than over edges.

Thus, when switching from minimizing the sum of communication to minimizing the maximum communication, while the number of nonlocal columns is clearly the more accurate cost model, the number of incident columns is also a natural quantity to consider from an algorithmic perspective.