Load Plus Communication Balancing in Contiguous Partitions for Distributed Sparse Matrices: Linear-Time Algorithms

Peter Ahrens

***Abstract*—We study partitioning 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 the matrix are split into K parts without reordering. We present exact and approximate contiguous partitioning algorithms that minimize the runtime of the longest-running processor under cost models that combine work factors and hypergraph communication factors. This differs from traditional graph or hypergraph partitioning models which minimize total communication under a work balance constraint. We address regimes where partitions of the row space and column space are expected to match (the symmetric case) or are allowed to differ (the nonsymmetric case).**

Our algorithms use linear space. Our exact algorithm runs in linear time when K^2 is sublinear. Our $(1+\epsilon)$-approximate algorithm runs in linear time when $K\log(1/\epsilon)$ is sublinear.

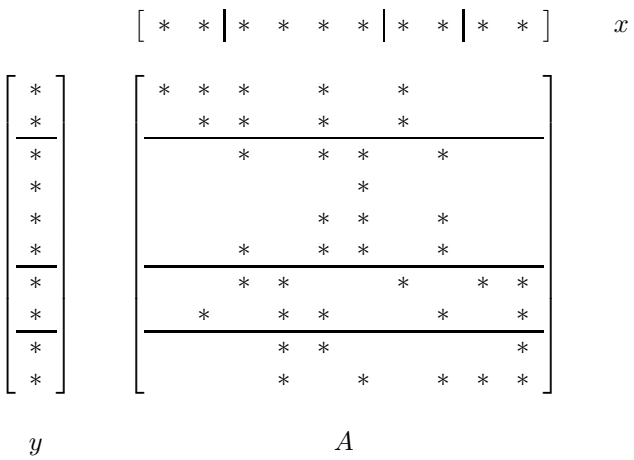
We combine concepts from high-performance computing and computational geometry. Existing load balancing algorithms optimize a linear model of per-processor work. We make minor adaptations to optimize arbitrary nonuniform monotonic increasing or decreasing cost functions which may be expensive to evaluate. We then show that evaluating our model of communication is equivalent to planar dominance counting. We specialize Chazelle's dominance counting algorithm to points in the bounded integer plane and generalize it to trade reduced construction time for increased query time, since our partitioners make very few queries.

Our algorithms split the original row (or column) ordering into parts to optimize diverse cost models. Combined with reordering or embedding techniques, our algorithms might be used to build more general heuristic partitioners, as they can optimally round one-dimensional embeddings of direct K-way noncontiguous partitioning problems.

I. 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 parts, then assign each part 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 embeddings. 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 accurately. As we will demonstrate, the contiguous partitioning techniques we develop are efficient 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, including iterative

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.

solvers, all processors wait for the last processor to finish multiplication. This synchronization presents a load balancing problem where our goal is to minimize the maximum amount of time that any processor requires to finish and communicate its portion of the product. If we assign a cost to each row (typically proportional to the number of nonzeros plus some constant term reflecting linear operations) and model the cost of a part as the sum of the costs of rows in the part, this corresponds to the "Chains-On-Chains" load balancing problem. Chains-On-Chains has received extensive study over several decades; we refer to [5] for a summary.

Unfortunately, communication costs and storage considerations cannot be modeled accurately as the sum of some per-row workload. Iterative solvers require each processor to communicate completed entries of their part of the solution vector to the other processors that need them. The size of the messages that need to be sent is proportional to the number of distinct nonzero column locations that occur in the row part, regardless of how many times a particular column appears in the part. Parts which have nonzeros in many different columns need to communicate more. Accurate communication costs depend on the number of distinct nonlocal columns; this relationship is nonlinear and depends on the sparsity structure of the matrix itself. Cache and storage effects are also nonlinear; the cost to compute with 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 processors memory.

A. 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 show how state-of-the-art sublinear time algorithms for chain partitioning can optimize 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 run in linear time, use linear space, and optimize accurate models. 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 [6].

Our contributions are three-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. [7], [8] and further improved by Pinar et. al. [5]) to support arbitrary nonuniform monotonic increasing and decreasing cost functions which may be expensive to evaluate.
- We use a slightly modified reduction from multicolored one-dimensional dominance counting to standard two-dimensional dominance counting to compute our communication terms [9]. We generalize a two-dimensional dominance counting algorithm due to Chazelle to trade construction time for query time, allowing the overall algorithm to run in linear time [10].

When load-balancing across a sublinear number of processors, we desire a data structure that can be constructed in linear time and evaluate queries in sublinear time. Since our dominance counting algorithm is specialized to the case of computing prefix sums in a sparse matrix, it may be of independent interest, especially for two-dimensional rectilinear load balancing problems with linear cost functions [8], [11], [12].

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 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 3) is

$$O(m + n + HN + K \log\left(\frac{K}{\epsilon}\right) \log(m) H^2 N^{1/H}) = O(m + n + N + K \log\left(\frac{K}{\epsilon}\right) \log(m) N^{1/3})$$

for subadditive monotonically increasing costs, which is linear when K grows strictly slower than $N^{1-1/H} = N^{2/3}$. The runtime of our exact partitioner (Algorithm 4) is

$$O(m + n + HN + K^2 \log(m)^2 H^2 N^{1/H}) = O(m + n + N + K^2 \log(m)^2 N^{1/3})$$

which is linear when K grows strictly slower than $N^{(1-1/H)/2} = N^{1/3}$. Both algorithms use at most $2N + n$ extra storage, regardless of the choice of H.

II. 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. Let N be the number of nonzeros in A. Since most of their entries are zeros, it is more efficient to store sparse matrices in **compressed** formats. For example, the **Compressed Sparse Row (CSR)** format stores a sorted vector of nonzero column coordinates in each row, and a corresponding vector of values. This is accomplished efficiently 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. Sometimes it can be convenient to use **Compressed Sparse Column (CSC)** format, the transpose of CSR format where columns are compressed instead of rows.

A. 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 solve the symmetric, positive semidefinite linear system $A \cdot x = b$ through repeated multiplication by A. We consider the case where we wish to solve for p distinct right-hand-sides.

Algorithm 1 (Serial Conjugate Gradient Method)**.** *Solve the symmetric, positive semidefinite system $A \cdot x = b$ to a tolerance of ϵ. The initial value of x is a guess. Note that in the case that there are p right-hand sides, x and b should be considered $n \times p$ matrices. Adapted from [13, Chapter 6].*

function CONJUGATEGRADIENT*(A, x, b, ϵ)*
  $r \leftarrow b - A \cdot x$
  $\delta \leftarrow r \cdot r^\intercal$
  $z \leftarrow r$
  while $\delta^{1/2} < \epsilon$ ***do***
    $y \leftarrow A \cdot z$
    $\alpha \leftarrow \delta/(y \cdot z)$
    $x \leftarrow x + \alpha z$
    $r \leftarrow r - \alpha y$
    $\delta' \leftarrow r \cdot r^\intercal$
    $\beta \leftarrow \delta'/\delta$
    $z \leftarrow r + \beta z$
    $\delta \leftarrow \delta'$
  end while
  return x
end function

Because we can expect the inner loop to execute many times, we ignore the initialization steps and focus on the loop itself. The dominant cost is the sparse matrix dense vector **SpMV** multiply $A \cdot z$ (or sparse matrix dense matrix **SpMM** multiply if $p > 1$), but there are also several dot products and pointwise additions to compute, which become more noticeable as the matrix becomes sparse.

The two dot products in Algorithm 1 impose synchronization points, where all processors locally compute their dot products, but depend on a global sum of the local dot products to compute α and β, which are needed in the next phase of computation. This isolates the dominant phase of computation (the SpMV) from when each processor receives a global value of β to when all processors have posted their local contribution to α. Once β is received, processors compute their local copies of z, receive relevant nonlocal entries of z from other processors, compute $y = A \cdot z$ and then post $y \cdot z$.

A high-level summary of the conjugate gradient method is presented in Algorithm 1. We use conjugate gradient as an example because it is a well-known algorithm which helps illustrate several of the important terms in the computational cost of iterative solvers. While conjugate gradient is our running example, the cost functions we present are formulated for more broad applications of SpMV and SpMM. For example, similar iterative algorithms exist for least-squares problems that apply to possibly rectangular matrices, multiplying by both A and $A^\intercal$ in each iteration [13]. We must therefore distinguish when we intend to partition in the symmetric or nonsymmetric (and possibly rectangular) case.

B. Parallelism and Partitioning

We consider a distributed memory model where storage of the matrix and intermediate vectors is divided across the different processors, and any 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. Without loss of generality since transposition runs in linear time, we assume our matrix is stored in CSR format.

We must decide how to break up our data among processors to compute $y = A \cdot x$. 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 processors 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 that we decompose the SpMV row-wise, 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.

In a **K-partition** Π of the rows of A, each row i is assigned to a single part k, and the set of rows in part k is denoted as π_k. We require that the parts be disjoint. Any partition Π on n elements must satisfy coverage and disjointness constraints:

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

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 [14], [15]. Similar alternating approaches have been used for the related two-dimensional rectilinear partitioning regimes [8], [12].

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 wealth of literature has been devoted to noncontiguous partitioning, where elements can be assigned to arbitrary processor locations. While contiguous partitions are less flexible, they are useful when the cost of reordering the matrix is unacceptable, when the matrix already has good local structure, or when the matrix has already been carefully ordered for numerical reasons. Contiguous partitions are also easier to conceptually understand and implement. This work demonstrates that optimal contiguous partitions can be constructed in nearly linear time, whereas noncontiguous partitioning is usually NP-Hard [16], [17]. Even noncontiguous secondary alternate partitioning to balance communication (known as the "Column Assignment Problem") is NP hard [18].

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. Figure 1 illustrates a symmetric decomposition in the context of Algorithm 1.

C. Cost Models and Optimization

The history of cost models used to optimize iterative solvers is tied to both the methods available to optimize these models and the nature of the architecture under consideration. Since communication costs are quite expensive in the distributed memory model and storage limits on each node can be a concern, classical graph and hypergraph partitioners use graph theory to model and minimize the communication costs under per-node work or storage balance constraints.

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.

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 [15].

Problem 1 (Edge Cut Graph Partitionining)**.** Optimize the feasible (1) partition Π under

$$G = (V, E) = \mathrm{adj}(A)$$
$$\arg\min_{\Pi} |\{E \cap \pi_k \times \pi_{k'} : k \neq k'\}| :$$
$$\forall k, \sum_{i \in \pi_k} w_i < \frac{1+\epsilon}{K} \sum_i w_i$$

Inaccuracies in the graph model led to the development of the hypergraph model of communication. Here we use the **incidence representation** of a hypergraph, $\mathrm{inc}(A)$. If $G = \mathrm{inc}(A)$, 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. While it is clear that this will work in the nonsymmetric regime, Reference [4] explains how the partitions may be constrained to be symmetric when A is square. 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, $\lambda_j = \{k : i \in \pi_k, a_{ij} \neq 0\}$. The former cost is known as the "edge cut" metric and the latter is known as the "(λ - 1) cut" metric.

Problem 2 (Hyperedge Cut Hypergraph Partitioning)**.** Optimize the feasible (1) partition Π under

$$G = (V, E) = \mathrm{inc}(A)$$
$$\arg\min_{\Pi} |\{e_j \in E : |\lambda_j| > 1\}| :$$
$$\forall k, \sum_{i \in \pi_k} w_i < \frac{1+\epsilon}{K} \sum_i w_i$$

Problem 3 ($(\lambda - 1)$ Cut Hypergraph Partitioning)**.** Optimize the feasible (1) partition Π under

$$G = (V, E) = \mathrm{inc}(A)$$
$$\arg\min_{\Pi} \sum_{e_j \in E} |\lambda_j| - 1 :$$
$$\forall k, \sum_{i \in \pi_k} w_i < \frac{1+\epsilon}{K} \sum_i w_i$$

While the hypergraph model better captures communication in our kernel, heuristics for these noncontiguous partitioning problems are highly specialized to the problem formulation, and can be expensive. For example, state-of-the-art multilevel partitioners recursively pair vertices and vertex groups, maintaining balance constraints and using iterative improvement algorithms at each level of grouping. The pairing step can

be thought of as a weighted maximum matching problem (often approached with greedy heuristics). Weights in the graph case are the edge weights, the weights in the hypergraph case are the vertex similarities, measured by the number (or total weight) of shared edges. The refinement is usually accomplished with the Kernighan-Lin heuristic in graphs or the FiducciaMattheyses algorithm in hypergraphs [3], [4]. A multilevel graph partitioner using a greedy coarsening algorithm and the FM heuristic runs in log-linear time. However, the added complexity of the hypergraph case is costly (especially the coarsening step), and these heuristics usually require at least quadratic time.

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 communication, rather than the sum of all communication [15]. 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 [19], [20], [21]. Other approaches modify traditional hypergraph partitioning techniques to incorporate communication balance (and other metrics) in a single phase of partitioning [22], [23], [24]. Given a row partition, Bisseling et. al. consider finding a column partition to balance communication (the secondary alternate partitioning regime) [18].

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 [25], [26].

Simplifications to the cost function lead to further reductions in runtime and stronger guarantees. Much research has been devoted to the "Chains-On-Chains" partitioning problem, where the objective is to produce a contiguous partition that minimizes the maximum amount of work performed by any processor (where work is modeled as the sum of some per-element work). The cost function doesn't involve communication, leading to algorithmic simplifications. When minimizing the maximum amount of work in a noncontiguous asymmetric (primary or secondary) partition, our partitioning problem is equivalent to bin-packing, which is approximable using straightforward heuristics that can be made to run in log-linear time [27]. 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,

Problem 4 (Chains-On-Chains)**.** Optimize the feasible (1) contiguous (2) partition Π under

$$G = (V, E) = \text{adj}(A)$$
$$\arg\min_{\Pi} \max_{k} \sum_{i \in \pi_k} |v_i|$$

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 saxpy operations in iterative linear solvers [28]. Local refinements to contiguous partitions have been proposed to take communication factors into account, but this work proposes the first globally optimal single-phase contiguous partitioner with a communication-based cost model [29], [2].

III. Problem Formulation

Although it is simpler and often less accurate, the objective of Chains-On-Chains (Problem 4) has two key advantages over the graph-based objectives of Problems 1, 2, and 3; Chains-On-Chains treats work balance as an objective, not a constraint, and the cost function is monotonic.

Since Chains-On-Chains partitioning uses partition balance as an objective, attempting to minimize the time spent by the longest-running processor, it better captures the effects of the main synchronization point in our iterative solver. The runtime of the solver doesn't depend directly on the total amount of work or communication, just on the maximum amount of work and communication any particular processor has to perform [6], [20], [18], [30]. We refer to a costliest part in a partition as a **bottleneck** processor.

Additionally, Chains-On-Chains partitioning minimizes an objective function which is **monotonic**, meaning that adding additional rows to a part will never decrease (or never increase) the cost of that part, whereas neither of the communication metrics of the graph-based objectives are monotonic. The combination of a balance objective and monotonic cost function makes the contiguous partitioning problem efficiently solvable. Put simply, monotonicity implies that adding vertices to a part can only reduce (or only increase) the cost of the other parts, which enables parametric search techniques [5].

A. Monotonic Load Balancing

This leads us to the first contribution of this work: monotonic cost models for SpMV in the context of iterative solvers on distributed memory machines. We start with a formal definition of monotonicity. 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 $\pi \subseteq \pi'$,

$$f(\pi) \leq f(\pi'). \tag{3}$$

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

$$f(\pi) \geq f(\pi'). \tag{4}$$

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:

$$\begin{aligned} g_1(\pi) &= \min(f_1(\pi), f_2(\pi)) \\ g_2(\pi) &= \max(f_1(\pi), f_2(\pi)) \\ g_3(\pi) &= f_1(\pi) + f_2(\pi) \\ g_4(\pi) &= f_1(\pi) \cdot a \end{aligned} \tag{5}$$

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.

Problem 5 (Monotonic Load Balancing)**.** Optimize the feasible (1) contiguous (2) partition Π under

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

where each f_k is monotonic increasing (3) or each f_k is monotonic decreasing (4).

B. Cost Modeling

Looking carefully at Algorithm 1, our example iterative solver, the inner loop can be divided into two phases, separated by the synchronization to compute dot products for α and β. The first phase, between α and β, which computes x, r, and β, is relatively inexpensive and requires no communication other than a scalar reduction at the beginning and end.

We focus instead on the phase between β and α, which computes z, y, and α. This phase is much more expensive, involving the key SpMV operation. It begins with each processor sending copies of their local portions of z to processors that need it, then receiving the required nonlocal copies of z from other processors. In distributed memory settings, both the sending and the receiving node must participate in transmission of the message. The accuracy to which we can model the effects of communication will depend on whether we are partitioning in the nonsymmetric or symmetric regime.

In all of our partitioning regimes, we model the per-row work (computation) cost (due to dot products, vector scaling, and vector addition) with the scalar $c_{\textbf{dot}}$, and the per-nonzero work costs (due to matrix multiplication) with the scalar $c_{\textbf{spmv}}$.

We further 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 [18]. 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 process sends and 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 assume the cost of receiving an entry is proportional to some scalar $c_{\textbf{message}}$.

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 of our sparse matrix multiplication operation and the column (input) space partition Φ, considering the other to be fixed.

We model our matrix as an incidence hypergraph. Notice that we are incentivized to pick partitions where elements of the input vector that reside on a processor are reused in computing output vector elements on the same processor, so they do not need to be communicated. 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, an expressive cost model for the alternating regime can be constructed as

$$f_k(\pi_k, \phi_k) = c_{\textbf{dot}}|\pi_k| + c_{\textbf{spmv}} \sum_{i\in\pi_k} |v_i| + c_{\textbf{message}} \left| \left(\bigcup_{i\in\pi_k} v_i \right) \setminus \phi_k \right| \tag{6}$$

When Φ is fixed, each f_k is a linear combination of monotonic increasing factors and is therefore monotonic increasing (increasing the size of a row 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 (increasing the size of a column part can only increase the number of local columns, decreasing the communication of part k).

While we require the opposite partition to be fixed, we will only require the partition we are currently constructing to be contiguous. Requiring that both Π and Φ to be contiguous 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 simply assign each column to an incident part to optimize total communication volume as suggested by Çatalyurek [4].

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, we can assume that all entries of the input vector must be communicated to processors that need them, regardless of whether they are stored locally or not, then we can decouple the costs of the two partitions, upper-bounding the cost of communication. 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

$$f(\pi_k) = c_{\textbf{dot}}|\pi_k| + c_{\textbf{spmv}} \sum_{i\in\pi_k} |v_i| + c_{\textbf{message}} \left| \bigcup_{i\in\pi_k} v_i \right|. \tag{7}$$

Note that any column partition Φ will achieve or improve on the modeled cost of (7).

2) Monotonic Symmetric Cost Modeling: In the symmetric partitioning case, we can achieve an approximation of the accuracy of the nonsymmetric model by adjusting scalar coefficients. However, the symmetric partitioning regime does not need multiple alternating steps, and we construct our partition with just one solution to Problem 5. Recall that the symmetric case asks us to produce a partition Π which will be used to partition both the row and column space simultaneously. Simply replacing Φ with Π in (6), we obtain

$$f(\pi_k) = c_{\textbf{dot}}|\pi_k| + c_{\textbf{spmv}} \sum_{i\in\pi_k} |v_i| + c_{\textbf{message}} \left| \left(\bigcup_{i\in\pi_k} v_i \right) \setminus \pi_k \right| \tag{8}$$

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 $v_{\min}$ nonzeros (in linear solvers, $v_{\min}$ should be at least two, or less occupied rows could be trivially computed from other rows.) We rewrite (8) in the following form,

$$f(\pi_k) = (c_{\textbf{dot}} + v_{\min} \cdot c_{\textbf{spmv}} - c_{\textbf{message}})|\pi_k| + c_{\textbf{spmv}} \sum_{i\in\pi_k} (|v_i| - v_{\min}) + c_{\textbf{message}} \left| \left(\bigcup_{i\in\pi_k} v_i \right) \cup \pi_k \right|.$$

Since $|\pi_k|$, $|(\bigcup_{i\in\pi_k} v_i) \cup \pi_k|$, and $\sum_{i\in\pi_k} (|v_i| - v_{\min})$ are all monotonic, we can say that (8) is monotonic when the coefficients on these terms are positive. We therefore require

$$c_{\textbf{dot}} + v_{\min} c_{\textbf{spmv}} \geq c_{\textbf{message}} \tag{9}$$

Informally, (9) asks that the rows and dot products hold "enough" local work to rival communication costs. These conditions correspond to situations where it would cheaper to communicate a summand in $y^\intercal \cdot A \cdot x$ than it would be to compute it (assuming the required entries of y are locally available). 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 how sparse our matrix is, we may approximate the modeled cost of the matrix by assuming $v_{\min}$ to be larger than it really is. If there are at most m' "underfull" rows with less than $v_{\min}$ nonzeros, then (8) will be additively inaccurate by at most $m' \cdot v_{\min} c_{\text{spmv}}$.

C. 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. If we have a more expensive cost model we wish to use when our input or output vectors do not fit in cache, we can multiply the more expensive model by a threshold and take the larger of the in-cache model and the thresholded out-of-cache model.

D. Atoms and Molecules

Together, 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 (6), (7), (8), and even new cost functions which we have yet to consider. Since Nicol's algorithm minimizes monotonic cost functions on contiguous partitions with a sublinear number of calls to the cost function, our overall algorithm will be efficient if we can ensure that our functions, and thus, their composite atoms, are efficiently computable. Figure 2 displays two example partitions and the value of the corresponding atoms.

We assume our partition 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 the 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. Storage requirements usually depend on factors such as $|\pi_k|$ and $\sum_{i\in\pi_k} |v_i|$. All that remains is to find an efficient way to compute $|(\bigcup_{i\in\pi_k} v_i \setminus \phi_k)|$, $|(\bigcup_{i\in\pi_k} v_i)|$, and $|(\bigcup_{i\in\pi_k} v_i) \cup \pi_k|$, which we will discuss later in Section V.

E. Bounding the Costs

While our partitioner for Problem 5 does not make any assumption on the cost other than nonnegativity and monotonicity, our approximate partitioner will need an upper and lower bound on the cost to search within, and both algorithms perform better when given better bounds on the cost.

We can use **subadditivity** in our 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',

$$f(\pi) + f(\pi') \geq f(\pi \cup \pi') \tag{10}$$

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:

$$\begin{aligned} g_1(\pi) &= \max(f_1(\pi), f_2(\pi)) \\ g_2(\pi) &= f_1(\pi) + f_2(\pi) \\ g_3(\pi) &= f_1(\pi) * a \end{aligned} \tag{11}$$

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,

$$\max_k f(\pi_k) \leq f(V). \tag{12}$$

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,

$$\max_k f(\pi_k) \geq \frac{f(V)}{K}. \tag{13}$$

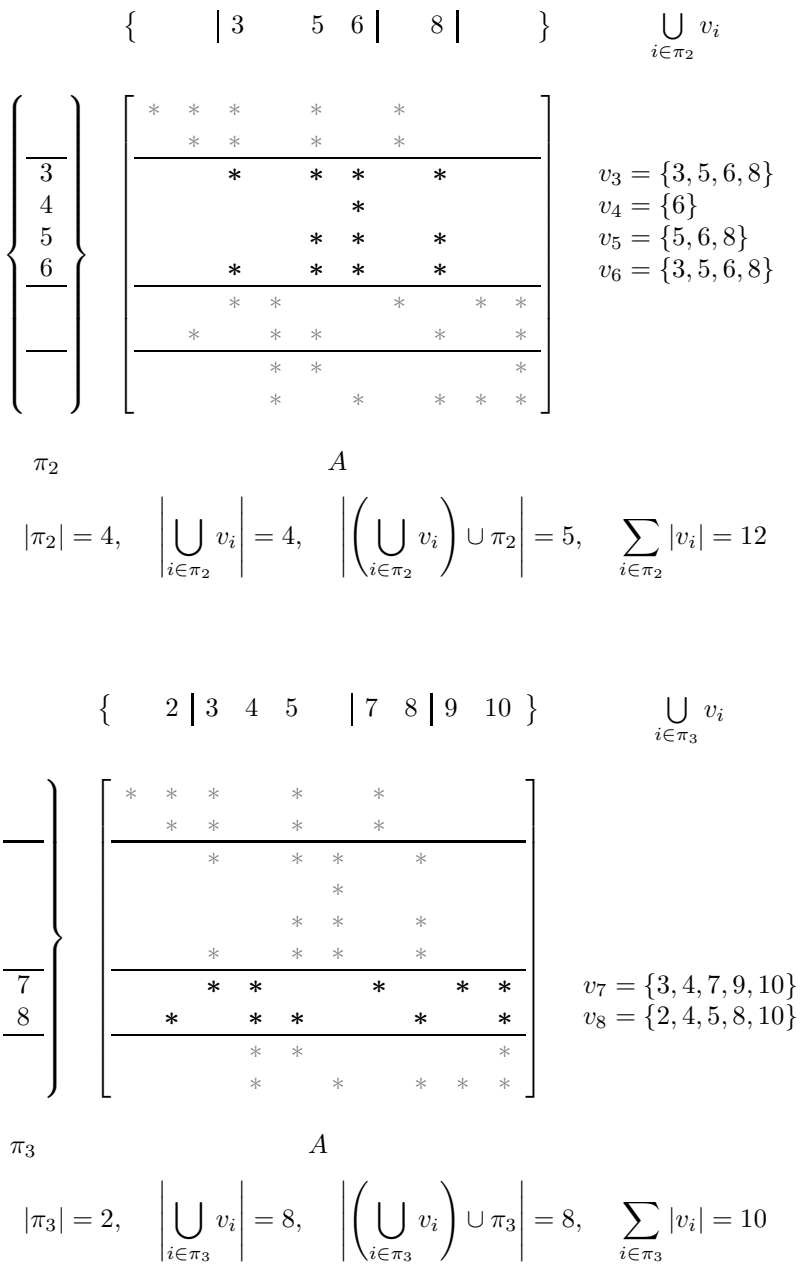


Fig. 2: Two parts in a symmetric partition of our example matrix, and the values of some of our various "atoms" for each part. Although part 2 (shown on top) contains more nonzeros than part 3 (shown on bottom), part 3 contains more distinct nonzero column locations, and will therefore require more communication.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

Finding lower bounds on partition costs is more difficult than finding upper bounds. Equations (7) and (8) are subadditive and use the same cost for each part, so they obey (13). However, Equation (6) uses different functions f_k on each part, and therefore we cannot apply (13). One workaround is to lower bound only the work terms in the cost function, since these are uniform across partitions. This is equivalent to first lower-bounding all our functions by some uniform subadditive one, then applying Equation (13).

Similarly, although subadditivity is not preserved under operations involving thresholds $\tau_{w_{\max}}$, we can often provide lower bounds by examining only the subadditive terms. For example, if our cost function is subadditive except for a finite set of monotonic constraint functions $\tau_{w_{\max}}...$, we can write it in the form

$$f(\pi) = \max(f'(\pi), \tau_{w_{\max}}...),$$

and simply use the upper and lower bound on $f'(\pi)$, since this is the only bound that matters if a feasible solution exists.

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.

IV. Monotonic Increasing Partitioners

Pinar et. al. present a multitude of algorithms for optimizing linear cost functions [5]. We examine both an approximate and an optimal algorithm. We chose the approximate "ϵ-BISECT+" algorithm (originally due to Iqbal et. al. [7]) and the exact "NICOL+" algorithm (originally due to Nicol et. al. [8]) because they were easy to understand and enjoyed strong guarantees, but used dynamic split point bounds and other optimizations based on problem structure which result 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, increasing the size of some part in a partition will not increase the cost of other parts. Therefore, if we know the cost of a K-partition to be at most c, then we can set the endpoint of the first part to the largest i' such that $f_1(1, i') \leq c$, and the remaining $K-1$-partition which starts at i' will also cost at most c. This observation provides us with a procedure that determines whether a partition of cost c is feasible. We can use such a procedure to search over 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 only return the optimal partition value, our cost function is much more expensive to evaluate than theirs, so our presentations of their algorithms have been modified to return split points with no extra evaluations of the cost functions [5].

Since parametric search is a repeated pattern, we provide pseudocode for our binary search procedure as a warm up.

Algorithm 2. *Given a monotonic increasing cost function f_k defined on pairs of split points, a starting split point i, and a maximum cost c, find the greatest i' such that $i < i'$, $f_k(i, i') \leq c$, and $i'_{\text{low}} \leq i' \leq i'_{\text{high}}$. Returns $\max(i, i'_{\text{low}}) - 1$ if no cost at most c can be found.*

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

We now state our adapted bisection algorithm as Algorithm 3. Algorithm 3 differs from Pinar et. al. in that it is stated in terms of possibly different f for each part, makes no assumptions on the value of $f_k(i, i)$, 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.

Since Algorithm 3 evaluates at most $\log_2(c_{\text{high}}/(c_{\text{low}}\epsilon))$ objectives, the number of calls to the cost function is bounded by

$$K \log_2(m) \log_2 \left(\frac{c_{\text{high}}}{c_{\text{low}}\epsilon} \right). \tag{14}$$

If our cost is subadditive and we use (13) to correctly set $c_{\text{high}} = f(1, m+1)$, $c_{\text{low}} = f(1, m+1)/K$, then the number of calls to the cost function is bounded by

$$K \log_2(m) \log_2(K/\epsilon). \tag{15}$$

Algorithm 3 (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 $c_{\text{low}} \leq c \leq c_{\text{high}}$, if such a partition exists.

This is an adaptation of the "ϵ-BISECT+" algorithm by Pinar et. al. [5], which was a heuristic improvement on the algorithm proposed by Iqbal et. al. [7].

function BISECTPARTITION*(f, n, c_{low}, c_{high}, ϵ)*
  $(s_{\text{high}\,1}, ..., s_{\text{high}\,K+1}) \leftarrow (1, m+1, m+1, ..., m+1)$
  $(s_{\text{low}\,1}, ..., s_{\text{low}\,K+1}) \leftarrow (1, 1, ..., 1)$
  $(s_1, ..., s_{K+1}) \leftarrow (\#, \#, ..., \#)$
  while $c_{\text{low}}(1+\epsilon) < c_{\text{high}}$ ***do***
    $c \leftarrow (c_{\text{low}} + c_{\text{high}})/2$
    $s_1 \leftarrow 1$
    $t \leftarrow \text{true}$
    for $k = 1, 2, ..., K$ ***do***
      $s_{k+1} \leftarrow \text{SEARCH}(f_k, s_k, s_{\text{low}\,k+1}, s_{\text{high}\,k+1}, c)$
      if $s_{k+1} < \max(s_k, s_{\text{low}\,(k+1)})$ ***then***
        $t \leftarrow \text{false}$
        $s_{k+1}, ..., s_{K+1} \leftarrow \max(s_k, s_{\text{low}\,k+1})$
        break
      end if
    end for
    if t ***and*** $s_{K+1} = m+1$ ***then***
      $c_{\text{high}} \leftarrow c$
      $S_{\text{high}} \leftarrow S$
    else
      $c_{\text{low}} \leftarrow c$
      $S_{\text{low}} \leftarrow S$
    end if
  end while
end function
return S_{high}

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 [8]. 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 [5] 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 the previous processor was not a bottleneck.

We have made similar modifications in our adaptation of "NICOL+" as we did for our adaptation of "ϵ-BISECT+." We 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+" ([5]) for general monotonic part costs is presented in Algorithm 4.

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

$$K^2 \log_2(m)^2. \tag{16}$$

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

Algorithm 4 (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 $c_{\text{low}} \leq c \leq c_{\text{high}}$, if such a partition exists.

This is an adaptation of the "NICOL+" algorithm by Pinar et. al. [5], which was a heuristic improvement on the algorithm proposed by Nicol et. al. [8].

function NICOLPARTITION(f, m, $c_{\text{low}} = -\infty$, $c_{\text{high}} = \infty$)
  $(s_{\text{high}1}, ..., s_{\text{high}K+1}) \leftarrow (1, m+1, m+1, ..., m+1)$
  $(s_{\text{low}1}, ..., s_{\text{low}K+1}) \leftarrow (1, 1, ..., 1)$
  $(s_1, ..., s_{K+1}) \leftarrow (\#, \#, ..., \#)$
  for $k \leftarrow 1, 2, ..., K$ ***do***
    $i \leftarrow s_k$
    $i'_{\text{high}} \leftarrow s_{\text{high}k+1}$
    $i'_{\text{low}} \leftarrow \max(s_k, s_{\text{low}k+1})$
    while $i'_{\text{low}} \leq i'_{\text{high}}$ ***do***
      $i' \leftarrow \lfloor (i'_{\text{low}} + i'_{\text{high}})/2 \rfloor$
      $c \leftarrow f(i, i')$
      if $c_{\text{low}} \leq c < c_{\text{high}}$ ***then***
        $s_{k+1} \leftarrow i'$
        $t \leftarrow \text{true}$
        for $k' = k+1, k+2, ..., K$ ***do***
          $s_{k'+1} \leftarrow$ SEARCH$(f_{k'}, s_{k'}, s_{\text{low}k'+1}, s_{\text{high}k'+1}, c)$
          if $s_{k'+1} < \max(s'_k, s_{\text{low}k'+1})$ ***then***
            $t \leftarrow \text{false}$
            $s_{k'+1}, ..., s_{K+1} \leftarrow \max(s_{k'}, s_{\text{low}k'+1})$
            break
          end if
        end for
        if t ***and*** $s_{K+1} = m+1$ ***then***
          $c_{\text{high}} \leftarrow c$
          $i'_{\text{high}} \leftarrow i' - 1$
          $S_{\text{high}} \leftarrow S$
        else
          $c_{\text{low}} \leftarrow c$
          $i'_{\text{low}} \leftarrow i' + 1$
          $S_{\text{low}} \leftarrow S$
        end if
      else if $c \geq c_{\text{high}}$ ***then***
        $i'_{\text{high}} = i' - 1$
      else
        $i'_{\text{low}} = i' + 1$
      end if
    end while
    if $i'_{\text{high}} < \max(s_k, s_{\text{low}k+1})$ ***then***
      break
    end if
    $s_{k+1} \leftarrow i'_{\text{high}}$
  end for
  return S_{high}
end function

Of course, some modifications are needed to apply Algorithms 3 and 4 to monotonic decreasing cost functions. Because the increasing and decreasing variants are so similar, we state the monotonic decreasing versions in Appendix A.

V. Computing The Footprint

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 major challenge in this work. 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 [31], [26], [29], [2], [32]. These approaches are efficient if 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 dynamic programming approaches that will evaluate cost functions for all possible parts in a structured, exhaustive pattern. These approaches often take $O(m^2 + N)$ time or similar, meaning that 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. Thus, we need a data structure that supports efficient, arbitrary, evaluations of our communication terms.

In the secondary alternate regime when Π is fixed,

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

By constructing sorted list representations of each set $(\bigcup_{i\in\pi_k} v_i)$ in linear time and space, we can easily query the last term $|(\bigcup_{i\in\pi_k} v_i) \cap \phi_k|$ in $O(\log(m))$ time by searching for the boundaries of the contiguous region defining ϕ_k.

In the other regimes, we can compute our communication terms by 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 $(V', E') = \text{inc}(A')$, 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 have been 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 occur in. 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 its following duplicate 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 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

$$\left|\bigcup_{i\in\pi_k} v_i\right| = \sum_{i\in\pi_k} |v_i| - \sum_{(i,i')\in\pi_k\times\pi_k} l_{ii'} \tag{17}$$

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 $s_{k+1} = i' + 1$ is equal to the number of of points dominated by $(-i, i')$. The **dominance counting** problem in two dimensions asks for a data structure to support queries on the number of dominated points. 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 [9]. 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 [10], [33], [34]. 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 [10].

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 [35], [36], [33]) 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 [37] 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."

A. Dominance Counting In A Bounded Integer Grid

Chazelle's original formulation of the dominance counting algorithm is linear in the number of points to be stored, requires 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 from CSR to CSC. Because the algorithm is so detail-oriented, we give a high-level description, but leave the specifics to the pseudocode presented in Algorithm 5.

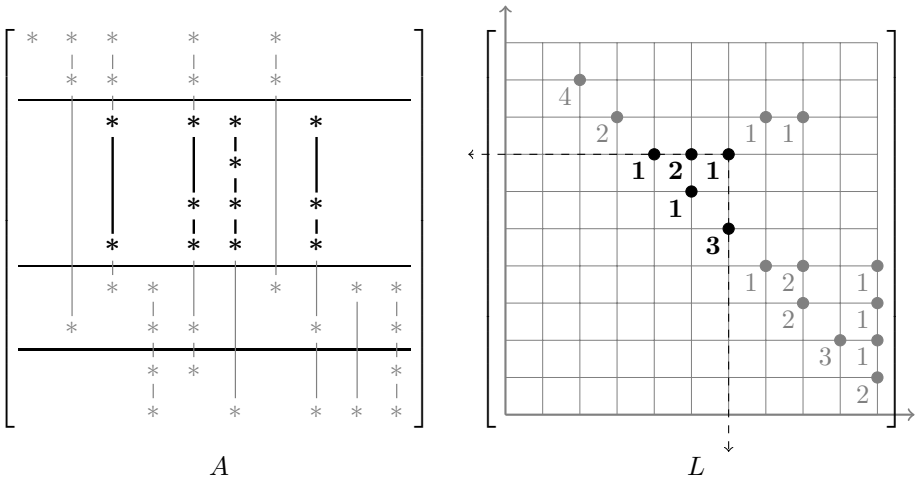


Fig. 3: Links of our example matrix A are illustrated as line segments connecting elements of A on the left, and as points (with labeled multiplicities) in the link matrix L on the right. Links residing entirely within part 2 are shown in bold. Part 2 contains two links starting at $i = 3$ and terminating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

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.

Algorithm 5. *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.*

function CONSTRUCTDOMINANCE*(N, idx)*
  $qos \leftarrow$ *zero-initialized vector of length* $n + 2$
  $qos_1 \leftarrow 1$
  $qos_{n+2} \leftarrow N + 1$
  $tmp \leftarrow$ *uninitialized vector of length* $2^b + 1$
  $cnt \leftarrow$ *zero-initialized 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 $d \leftarrow 1, 2, ..., 2^b$ ***do***
        $qos_{J+d2^{(h-1)b}} \leftarrow tmp_d$
      end for
    end for
    Fill tmp with zeros.
    for $Q \leftarrow 1, 1 + 2^{b'}, ..., N$ ***do***
      for $q \leftarrow Q, Q + 1, ..., Q + 2^{b'}$ ***do***
        $d \leftarrow key_h(idx_q)$
        $tmp_d \leftarrow tmp_d + 1$
      end for
      for $d \leftarrow 1, 2, ..., 2^b$ ***do***
        $cnt_{(d+1)Qh} \leftarrow tmp_d + cnt_{dQh}$
      end for
    end for
    $(idx, byt) \leftarrow (byt, idx)$
  end for
  $H \leftarrow idx$
  return (qos, byt, cnt)
end function

Algorithm 6. *Query 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.*

function QUERYDOMINANCE*(i, j)*
  $\Delta q \leftarrow pos_i - 1$
  $j \leftarrow j - 1$
  $c \leftarrow 0$
  for $h \leftarrow H, H - 1, ..., 1$ ***do***
    $j' \leftarrow 2^{hb} \lfloor j / 2^{hb} \rfloor$
    $q_1 \leftarrow qos_{j'} - 1$
    $q_2 \leftarrow q_1 + \Delta q$
    $d \leftarrow key_h(j)$
    $Q_1 \leftarrow \lfloor q_1 / 2^{b'} \rfloor + 1$

$Q_2 \leftarrow \lfloor q_2/2^{b'} \rfloor + 1$
$c \leftarrow cnt_{dQ_2h} - cnt_{dQ_1h}$
$\Delta q \leftarrow (cnt_{(d+1)Q_2h} - cnt_{dQ_2h})$
$\Delta q \leftarrow \Delta q - (cnt_{(d+1)Q_1h} - cnt_{dQ_1h})$
for $q \leftarrow 2^{b'}(Q_1 - 1) + 1, 2^{b'}(Q_1 - 1) + 2, ..., q_1$ ***do***
$d' \leftarrow key_h(byt_q)$
if $d' < d$ ***then***
$c \leftarrow c - 1$
else if $d' = d$ ***then***
$\Delta q \leftarrow \Delta q - 1$
end if
end for
for $q \leftarrow 2^{b'}(Q_2 - 1) + 1, 2^{b'}(Q_2 - 1) + 2, ..., q_2$ ***do***
$d' \leftarrow key_h(byt_q)$
if $d' < d$ ***then***
$c \leftarrow c + 1$
else if $d' = d$ ***then***
$\Delta q \leftarrow \Delta q + 1$
end if
end for
end for
return c
end function

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 $j_q >= j$. 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. Using our histogram sort with a scratch array of size 2^b, we can 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). Notice that in the initial ordering, $i_q < i$ is equivalent to $q < pos_i$. Thus, the dominating points reside within in 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 have seen 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), then 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 cached count array 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

$$O(n + HN(1 + 2^{b-b'})). \tag{18}$$

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

$$O(H2^{b'}). \tag{19}$$

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

$$m + n + 3 + N(1 + H2^{b-b'}) \tag{20}$$

words.

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.

B. Balancing the Tree

Our primary alternate cost function requires us to create K separate dominance counters for each set of columns ϕ_k. Unfortunately, the runtime and storage of our dominance counting data structure as stated depends on the dimension n. Thus, if each processor required a dominance counter, then the total runtime and storage would be $o(nK)$. To avoid this contingency, we may choose to first transform our points into "rank space" [38], [10]. 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 the initial resorting, our new construction time becomes

$$O(N + HN(1 + 2^{b-b'})). \tag{21}$$

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

$$O(\log(m) + \log(n) + H2^{b'}). \tag{22}$$

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 two orderings, and our storage cost becomes

$$N(3 + H2^{b-b'}) \tag{23}$$

words.

In the primary alternate case where 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.

The reader may ask why we have bothered to present the non-transformed version of our dominance counter first. We do so because the non-transformed counter 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.

TABLE I: Relevant quantities involved in runtime tradeoffs

(14)	Algorithm 3 Max Queries	$K \log_2(m) \log_2(c_{\text{high}}/(c_{\text{low}}\epsilon))$
(15)	Algorithm 3 (Subadditive Cost)	$K \log_2(m) \log_2(K/\epsilon)$
(16)	Algorithm 4 Max Queries	$K^2 \log_2(m)^2$
(21)	Dominance Construction Time	$O(N + HN(1 + 2^{b-b'})$
(22)	Dominance Query Time	$O(\log(m+n) + H2^{b'})$
(23)	Dominance Storage	$3N + HN2^{b-b'}$

C. 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

$$m + n + 3 + N(1 + H + H2^{b-b'}) \tag{24}$$

VI. Putting It Together

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 V, all of our cost functions can be evaluated with at most two calls to a dominance counting data structure. In the primary alternate case, we need to construct a dominance counter for each processor over N points total. Therefore, we assume a transformation to rank space for our runtime analysis, and include the cost of a linear-time counting sort in our construction phase.

Table I describes the relevant quantities.

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 3) runs in time

$$\begin{aligned} &O(m + n + HN(1 + 2^{b-b'}) + \\ &\quad K \log\left(\frac{K}{\epsilon}\right) \log(m)(\log(m+n) + H2^{b'})), \end{aligned} \tag{25}$$

for subadditive costs, and our exact partitioner (Algorithm 4) runs in time

$$O(m+n+HN(1+2^{b-b'})+K^2 \log(m)^2(\log(m+n)+H2^{b'})). \tag{26}$$

There are several ways to set H, b, and b'. Chazelle considered an algorithm which sets $H = \log_2(N)$, $b = 2$, and $b' = \log_2(\log_2(N))$. While storage would be linear and the query time 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 over the data 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 $2^{Hb} \geq n$. We minimize b' subject to $2^{b'} \geq H2^b$ to ensure that the footprint of our dominance counter is at most four times the size of A. Assuming H is a constant, our approximate partitioner (Algorithm 3) runs in time

$$O(m + n + HN + K \log\left(\frac{K}{\epsilon}\right) \log(m) H^2 N^{1/H}), \quad (27)$$

for subadditive costs, and our exact partitioner (Algorithm 4) runs in time

$$O(m + n + HN + K^2 \log(m)^2 H^2 N^{1/H}). \quad (28)$$

With these parameters, the approximate partitioner runs in linear time when $K \log(1/\epsilon)$ grows strictly slower than $N^{1-1/H}$ and the exact partitioner runs in linear time when K grows strictly slower than $N^{(1-1/H)/2}$.

Our algorithms can run in linear time precisely when our partitioners use strictly sublinear queries, since we are able to offset polynomial query time decreases with logarithmic construction time increases.

For our practical choice of $H = 3$, $K \log(1/\epsilon)$ needs to grow slower than $N^{2/3}$ for linear time approximate partitioning and K needs to grow 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.

VII. 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, and provably optimizes 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. [6], [20], [18], [30]. 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.

Our algorithm is the first to provably balance communication costs of contiguous partitions in linear time and use linear space. We show that we can compute our communication costs with dominance counting, and generalize a classical dominance counting algorithm to reduce construction time by increasing query time. Our new data structure can also be used to compute sparse prefix sums. We show that, with minor adaptation, state-of-the-art load balancing algorithms are capable of optimizing our cost functions in linear time.

In this version of the manuscript, we only address algorithmic challenges. Future work includes an empirical evaluation of serial and parallel implementations, as well as an exploration of the effects of contiguous communication balancing under popular row and column reorderings.

References

[1] Tony 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*, 18(3):943–948, May 1997.

[2] Kevin Aydin, MohammadHossein Bateni, and Vahab Mirrokni. Distributed Balanced Partitioning via Linear Embedding †. *Algorithms*, 12(8):162, August 2019. Number: 8 Publisher: Multidisciplinary Digital Publishing Institute.

[3] George. Karypis and Vipin. Kumar. A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs. *SIAM Journal on Scientific Computing*, 20(1):359–392, January 1998.

[4] U. V. Catalyurek and C. Aykanat. Hypergraph-partitioning-based decomposition for parallel sparse-matrix vector multiplication. *IEEE Transactions on Parallel and Distributed Systems*, 10(7):673–693, July 1999.

[5] Ali Pınar and Cevdet Aykanat. Fast optimal load balancing algorithms for 1D partitioning. *Journal of Parallel and Distributed Computing*, 64(8):974–996, August 2004.

[6] Bruce Hendrickson. Load balancing fictions, falsehoods and fallacies. *Applied Mathematical Modelling*, 25(2):99–108, December 2000.

[7] Mohammad Ashraf Iqbal. Approximate algorithms for partitioning problems. *International Journal of Parallel Programming*, 20(5):341–361, October 1991.

[8] D. M. Nicol. Rectilinear Partitioning of Irregular Data Parallel Computations. *Journal of Parallel and Distributed Computing*, 23(2):119–134, November 1994.

[9] P. Gupta, R. Janardan, and M. Smid. Further Results on Generalized Intersection Searching Problems: Counting, Reporting, and Dynamization. *Journal of Algorithms*, 19(2):282–317, September 1995.

[10] Bernard. Chazelle. A Functional Approach to Data Structures and Its Use in Multidimensional Searching. *SIAM Journal on Computing*, 17(3):427–462, June 1988.

[11] Erik Saule, Erdeniz Ö. Baş, and Ümit V. Çatalyürek. Load-balancing spatially located computations using rectangular partitions. *Journal of Parallel and Distributed Computing*, 72(10):1201–1214, October 2012.

[12] Abdurrahman Yaşar and Ümit V. Çatalyürek. Heuristics for Symmetric Rectilinear Matrix Partitioning. *arXiv:1909.12209 [cs]*, September 2019. arXiv: 1909.12209.

[13] Yousef Saad. *Iterative methods for sparse linear systems*. SIAM, Philadelphia, 2nd edition, 2003.

[14] Tamara G. Kolda. Partitioning sparse rectangular matrices for parallel processing. In Alfonso Ferreira, José Rolim, Horst Simon, and Shang-Hua Teng, editors, *Solving Irregularly Structured Problems in Parallel*, Lecture Notes in Computer Science, pages 68–79, Berlin, Heidelberg, 1998. Springer.

[15] Bruce Hendrickson and Tamara G. Kolda. Partitioning Rectangular and Structurally Unsymmetric Sparse Matrices for Parallel Processing. *SIAM Journal on Scientific Computing*, 21(6):2048–2072, January 2000. Publisher: Society for Industrial and Applied Mathematics.

[16] M. R. Garey, D. S. Johnson, and L. Stockmeyer. Some simplified NP-complete graph problems. *Theoretical Computer Science*, 1(3):237–267, February 1976.

[17] Thomas Lengauer. *Combinatorial algorithms for integrated circuit layout*. John Wiley & Sons, Inc., USA, 1990.

[18] Rob H. Bisseling and Wouter Meesen. Communication balancing in parallel sparse matrix-vector multiplication. *ETNA. Electronic Transactions on Numerical Analysis [electronic only]*, 21:47–65, 2005. Publisher: Kent State University, Department of Mathematics and Computer Science.

[19] A. Pinar and B. Hendrickson. Partitioning for complex objectives. In *Proceedings 15th International Parallel and Distributed Processing Symposium. IPDPS 2001*, pages 1232–1237, April 2001. ISSN: 1530-2075.

[20] Bora. Uçar and Cevdet. Aykanat. Encapsulating Multiple Communication-Cost Metrics in Partitioning Sparse Rectangular Matrices for Parallel Matrix-Vector Multiplies. *SIAM Journal on Scientific Computing*, 25(6):1837–1859, January 2004.

[21] Kadir Akbudak, Oguz Selvitopi, and Cevdet Aykanat. Partitioning Models for Scaling Parallel Sparse Matrix-Matrix Multiplication. *ACM Transactions on Parallel Computing*, 4(3):13:1–13:34, January 2018.

[22] Mehmet Deveci, Kamer Kaya, Bora Uçar, and Umit V. Çatalyürek. Hypergraph Partitioning for Multiple Communication Cost Metrics. *J. Parallel Distrib. Comput.*, 77(C):69–83, March 2015.

[23] Seher Acer, Oguz Selvitopi, and Cevdet Aykanat. Improving performance of sparse matrix dense matrix multiplication on large-scale parallel systems. *Parallel Computing*, 59:71–96, November 2016.

[24] M. Ozan Karsavuran, Seher Acer, and Cevdet Aykanat. Reduce Operations: Send Volume Balancing While Minimizing Latency. *IEEE Transactions on Parallel and Distributed Systems*, 31(6):1461–1473, June 2020. Conference Name: IEEE Transactions on Parallel and Distributed Systems.

[25] Brian W. Kernighan. Optimal Sequential Partitions of Graphs. *Journal of the ACM (JACM)*, 18(1):34–40, January 1971.

[26] Anael Grandjean, Johannes Langguth, and Bora Uçar. On Optimal and Balanced Sparse Matrix Partitioning Problems. In *2012 IEEE International Conference on Cluster Computing*, pages 257–265, September 2012. ISSN: 2168-9253.

[27] György Dósa. The Tight Bound of First Fit Decreasing Bin-Packing Algorithm Is FFD(I) ¡= 11/9OPT(I) + 6/9. In Bo Chen, Mike Paterson, and Guochuan Zhang, editors, *Combinatorics, Algorithms, Probabilistic and Experimental Methodologies*, Lecture Notes in Computer Science, pages 1–11, Berlin, Heidelberg, 2007. Springer.

[28] D.M. Nicol and D.R. O'Hallaron. Improved algorithms for mapping pipelined and parallel computations. *IEEE Transactions on Computers*, 40(3):295–306, March 1991.

[29] Louis H. Ziantz, Can C. Özturan, and Boleslaw K. Szymanski. Run-time optimization of sparse matrix-vector multiplication on SIMD machines. In Costas Halatsis, Dimitrios Maritsas, George Philokyprou, and Sergios Theodoridis, editors, *PARLE'94 Parallel Architectures and Languages Europe*, Lecture Notes in Computer Science, pages 313–322, Berlin, Heidelberg, 1994. Springer.

[30] Sivasankaran Rajamanickam and Erik Boman. Parallel partitioning with Zoltan: Is hypergraph partitioning worth it? In David Bader, Henning Meyerhenke, Peter Sanders, and Dorothea Wagner, editors, *Contemporary Mathematics*, volume 588, pages 37–52. American Mathematical Society, Providence, Rhode Island, January 2013.

[31] Peter Ahrens and Erik G. Boman. On Optimal Partitioning For Sparse Matrices In Variable Block Row Format. *arXiv:2005.12414 [cs]*, May 2020. arXiv: 2005.12414.

[32] C.J. Alpert and A.B. Kahng. Multiway partitioning via geometric embeddings, orderings, and dynamic programming. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 14(11):1342–1358, November 1995.

[33] Joseph JaJa, Christian W. Mortensen, and Qingmin Shi. Space-Efficient and fast algorithms for multidimensional dominance reporting and counting. In *Proceedings of the 15th international conference on Algorithms and Computation*, ISAAC'04, pages 558–568, Hong Kong, China, December 2004. Springer-Verlag.

[34] Timothy M. Chan and Bryan T. Wilkinson. Adaptive and Approximate Orthogonal Range Counting. *ACM Transactions on Algorithms*, 12(4):45:1–45:15, September 2016.

[35] Bernard Chazelle. Lower bounds for orthogonal range searching: part II. The arithmetic model. *Journal of the ACM (JACM)*, 37(3):439–463, July 1990.

[36] S. Alstrup, G. Stolting Brodal, and T. Rauhe. New data structures for orthogonal range searching. In *Proceedings 41st Annual Symposium on Foundations of Computer Science*, pages 198–207, November 2000. ISSN: 0272-5428.

[37] Michael Shekelyan, Anton Dignös, and Johann Gamper. Sparse prefix sums: Constant-time range sum queries over sparse multidimensional data cubes. *Information Systems*, 82:136–147, May 2019.

[38] Harold N. Gabow, Jon Louis Bentley, and Robert E. Tarjan. Scaling and related techniques for geometry problems. In *Proceedings of the sixteenth annual ACM symposium on Theory of computing*, STOC '84, pages 135–143, New York, NY, USA, December 1984. Association for Computing Machinery.

[39] Brad Jackson, Jeffrey D. Scargle, David Barnes, Sundararajan Arabhi, Alina Alt, Peter Gioumousis, Elyus Gwin, Paungkaew Sangtrakulcharoen, Linda Tan, and Tun Tao Tsai. An Algorithm for Optimal Partitioning of Data on an Interval. *IEEE Signal Processing Letters*, 12(2):105–108, February 2005. arXiv: math/0309285.

[40] H. A. Choi and B. Narahari. Efficient Algorithms for Mapping and Partitioning a Class of Parallel Computations. *Journal of Parallel and Distributed Computing*, 19(4):349–363, December 1993.

[41] B. Olstad and F. Manne. Efficient partitioning of sequences. *IEEE Transactions on Computers*, 44(11):1322–1326, November 1995. Conference Name: IEEE Transactions on Computers.

APPENDIX

A. Monotonic Decreasing Partitioners

Here we present modified versions of Algorithms 2, 3, and 4 which handle monotonic decreasing cost functions.

Algorithm 7. *Given a monotonic decreasing cost function f_k defined on pairs of split points, a starting split point i, and a maximum cost c, find the greatest i' such that $i < i'$, $f_k(i, i') \leq c$, and $i'_{\text{low}} \leq i' \leq i'_{\text{high}}$. Returns $\max(i, i'_{\text{low}}) - 1$ if no cost at most c can be found.*

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

Algorithm 8 (Reverse BISECT Partitioner)**.** *Given monotonic decreasing 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 $c_{\text{low}} \leq c \leq c_{\text{high}}$, if such a partition exists.

function REVERSEBISECTPARTITION(f, n, c_{low}, c_{high}, ϵ)
  $(s_{\text{high}_1}, ..., s_{\text{high}_{K+1}}) \leftarrow (1, m+1, m+1, ..., m+1)$
  $(s_{\text{low}_1}, ..., s_{\text{low}_{K+1}}) \leftarrow (1, 1, ..., 1)$
  $(s_1, ..., s_{K+1}) \leftarrow (\#, \#, ..., \#)$
  while $c_{\text{low}}(1 + \epsilon) < c_{\text{high}}$ ***do***
    $c \leftarrow (c_{\text{low}} + c_{\text{high}})/2$
    $s_1 \leftarrow 1$
    $t \leftarrow \text{true}$
    for $k = 1, 2, ..., K$ ***do***
      $s_{k+1} \leftarrow$ REVERSESEARCH$(f_k, s_k, s_{\text{low}_{k+1}}, s_{\text{high}_{k+1}}, c)$
      if $s_{k+1} > s_{\text{high}_{k+1}}$ ***then***

$t \leftarrow \text{false}$
$s_{k+1}, ..., s_{K+1} \leftarrow s_{\text{high}_{k+1}}, ..., s_{\text{high}_{K+1}}$
break
end if
end for
if t ***then***
$c_{\text{high}} \leftarrow c$
$S_{\text{low}} \leftarrow S$
else
$c_{\text{low}} \leftarrow c$
$S_{\text{high}} \leftarrow S$
end if
end while
end function
$s_{\text{high}_{K+1}} \leftarrow m + 1$
return S_{high}

Algorithm 9 (Reverse NICOL Partitioner)**.** *Given monotonic decreasing 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 $c_{\text{low}} \leq c \leq c_{\text{high}}$, if such a partition exists.

function REVERSENICOLPARTITION(f, m, $c_{\text{low}} = -\infty$, $c_{\text{high}} = \infty$)
$(s_{\text{high}_1}, ..., s_{\text{high}_{K+1}}) \leftarrow (1, m+1, m+1, ..., m+1)$
$(s_{\text{low}_1}, ..., s_{\text{low}_{K+1}}) \leftarrow (1, 1, ..., 1)$
$(s_1, ..., s_{K+1}) \leftarrow (\#, \#, ..., \#)$
for $k \leftarrow 1, 2, ..., K$ ***do***
$i \leftarrow s_k$
$i'_{\text{high}} \leftarrow s_{\text{high}_{k+1}}$
$i'_{\text{low}} \leftarrow \max(s_k, s_{\text{low}_{k+1}})$
while $i'_{\text{low}} \leq i'_{\text{high}}$ ***do***
$i' \leftarrow \lfloor (i'_{\text{low}} + i'_{\text{high}})/2 \rfloor$
$c \leftarrow f(i, i')$
if $c_{\text{low}} \leq c < c_{\text{high}}$ ***then***
$s_{k+1} \leftarrow i'$
$t \leftarrow \text{true}$
for $k' = k+1, k+2, ..., K$ ***do***
$s_{k'+1} \leftarrow$ REVERSESEARCH($f_{k'}, s_{k'}, s_{\text{low}_{k'+1}}, s_{\text{high}_{k'+1}}, c$)
if $s_{k'+1} < s_{\text{high}_{k'+1}}$ ***then***
$t \leftarrow \text{false}$
$s_{k'+1}, ..., s_{K+1} \leftarrow s_{\text{high}_{k'+1}}, ..., s_{\text{high}_{K+1}}$
break
end if
end for
if t ***then***
$c_{\text{high}} \leftarrow c$
$i'_{\text{low}} \leftarrow i' + 1$
$S_{\text{low}} \leftarrow S$
else
$c_{\text{low}} \leftarrow c$
$i'_{\text{high}} \leftarrow i' - 1$
$S_{\text{high}} \leftarrow S$
end if
else if $c \geq c_{\text{high}}$ ***then***
$i'_{\text{high}} = i' - 1$
else
$i'_{\text{low}} = i' + 1$
end if
end while
if $i'_{\text{low}} > s_{\text{high}_{k+1}}$ ***then***
break
end if
$s_{k+1} \leftarrow i'_{\text{high}}$
end for
$s_{\text{high}_{K+1}} \leftarrow m + 1$
return S_{high}
end function

The intuition behind these algorithms 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 cost, we look for a candidate partition which does not need to pass the last row without exceeding the cost. The majority of changes reside in the small details, which we leave to the pseudocode.

B. A Word On Dynamic Programming

Our complex dominance counting data structure is required for our partitioners because they require random access to the cost function. This may lead readers to wonder whether dynamic programming would offer a simpler algorithmic solution, since dynamic programming queries the cost function in a structured pattern. Indeed, it is possible to avoid the dominance counting tree entirely during dynamic programming by using a hash-based structure similar to those of [31], [26], [39], [29]. These structures compute all values of $f(i, i')$ for each i' in turn, and the dynamic programming solutions of Choi et. al., and Olstad et. al. compare the split points of all i for parts ending at each i' in turn [40], [41]. Since the pattern for computing costs matches the pattern for querying costs, we can use these algorithmic structures to provide amortized constant-time access to each query. Unfortunately, the dynamic programming solutions make K passes over the rows, or perform $O(K)$ work for each row. The total number of probes becomes $O(K(m-K))$, which would be superlinear by a factor of K when K is not a constant. The recursive bisection heuristics suggested by Pinar et. al. to provide static split point bounds to empirically accelerate the dynamic programming algorithm require random access to the cost function, and if they were implemented using an approach similar to Ziantz et. al., would require $N \log(m)$ time to execute [5], [29]. Therefore, we will not investigate dynamic programming solutions in this initial version of the manuscript.

C. A Word On Total Versus Maximum Communication

Our suggested initial cost function (7) 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 metric of Problem 3, 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 problem to consider from an algorithmic perspective.