View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Dagstuhl Research Online Publication Server

Sparse Matrix Multiplication and Triangle Listing in the Congested Clique Model

Keren Censor-Hillel
Department of Computer Science, Technion, Israel
ckeren@cs.technion.ac.il

Dean Leitersdorf
Department of Computer Science, Technion, Israel
dean.leitersdorf@gmail.com

Elia Turner
Department of Computer Science, Technion, Israel
eliaturner11@gmail.com

Abstract

We show how to multiply two $n \times n$ matrices S and T over semirings in the Congested Clique model, where n nodes communicate in a fully connected synchronous network using $O(\log n)$-bit messages, within $O(nz(S)^{1/3}nz(T)^{1/3}/n + 1)$ rounds of communication, where $nz(S)$ and $nz(T)$ denote the number of non-zero elements in S and T, respectively. By leveraging the sparsity of the input matrices, our algorithm greatly reduces communication costs compared with general multiplication algorithms [Censor-Hillel et al., PODC 2015], and thus improves upon the state-of-the-art for matrices with $o(n^2)$ non-zero elements. Moreover, our algorithm exhibits the additional strength of surpassing previous solutions also in the case where only one of the two matrices is such. Particularly, this allows to efficiently raise a sparse matrix to a power greater than 2. As applications, we show how to speed up the computation on non-dense graphs of 4-cycle counting and all-pairs-shortest-paths.

Our algorithmic contribution is a new *deterministic* method of restructuring the input matrices in a sparsity-aware manner, which assigns each node with element-wise multiplication tasks that are not necessarily consecutive but guarantee a balanced element distribution, providing for communication-efficient multiplication.

Moreover, this new deterministic method for restructuring matrices may be used to restructure the adjacency matrix of input graphs, enabling faster deterministic solutions for graph related problems. As an example, we present a new sparsity aware, *deterministic* algorithm which solves the triangle listing problem in $O(m/n^{5/3} + 1)$ rounds, a complexity that was previously obtained by a *randomized* algorithm [Pandurangan et al., SPAA 2018], and that matches the known lower bound of $\tilde{\Omega}(n^{1/3})$ when $m = n^2$ of [Izumi and Le Gall, PODC 2017, Pandurangan et al., SPAA 2018]. Naturally, our triangle listing algorithm also implies triangle counting within the same complexity of $O(m/n^{5/3} + 1)$ rounds, which is (possibly more than) a *cubic* improvement over the previously known *deterministic* $O(m^2/n^3)$-round algorithm [Dolev et al., DISC 2012].

2012 ACM Subject Classification Theory of computation → Graph algorithms analysis, Theory of computation → Distributed algorithms

Keywords and phrases congested clique, matrix multiplication, triangle listing

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2018.4

Related Version Some proofs are omitted from this paper and are presented in the full version, available online at [8], `https://arxiv.org/abs/1802.04789`.

© Keren Censor-Hillel, Dean Leitersdorf, and Elia Turner;
licensed under Creative Commons License CC-BY
22nd International Conference on Principles of Distributed Systems (OPODIS 2018).
Editors: Jiannong Cao, Faith Ellen, Luis Rodrigues, and Bernardo Ferreira; Article No. 4; pp. 4:1–4:17

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Funding This project has received funding from the European Union's Horizon 2020 Research And Innovation Programe under grant agreement no. 755839. Supported in part by ISF grant 1696/14.

Acknowledgements The authors thank Seri Khoury, Christoph Lenzen, and Merav Parter for many useful discussions and suggestions.

1 Introduction

Matrix multiplication is a fundamental algebraic task, with abundant applications to various computations. The value of the exponent ω of matrix multiplication, that is, the value ω for which $\Theta(n^\omega)$ is the complexity of matrix multiplication, is a central question in algebraic algorithms [25, 9, 26], and is currently known to be bounded by 2.3728639 [13].

The work of Censor-Hillel et al. [7] recently showed that known matrix multiplication algorithms for the *parallel* setting can be adapted to the distributed Congested Clique model, which consists of n nodes in a fully connected synchronous network, limited by a bandwidth of $O(\log n)$ bits per message. Subsequently, this significantly improved the state-of-the-art for a variety of tasks, including triangle and 4-cycle counting, girth computations, and (un)weighted/(un)directed all-pairs-shortest-paths (APSP). This was followed by the beautiful work of Le Gall [14], who showed how to efficiently multiply rectangular matrices, as well as multiple independent multiplication instances. These led to even faster algorithms for some of the tasks, such as weighted or directed APSP, as well as fast algorithms for new tasks, such as computing the size of the maximum matching.

In many cases, multiplication is required to be carried out for *sparse* matrices, and this need has been generating much effort in designing algorithms that are faster given sparse inputs, both in sequential (e.g., [27, 12, 17, 1, 15]) and parallel (e.g., [3, 5, 6, 2, 4, 20, 19, 24]) settings.

In this paper we focus our attention on the task of multiplying sparse matrices in the Congested Clique model, providing a novel *deterministic* algorithm with a round complexity which depends on the sparsity of the input matrices.

An immediate application of our algorithm is faster counting of 4-cycles. Moreover, a prime feature of our algorithm is that it speeds up matrix multiplication even if *only one* of the input matrices is sparse. The significance of this ability stems from the fact that the product of sparse matrices may be non-sparse, which in general may stand in the way of fast multiplication of more than two sparse matrices, such as raising a sparse matrix to a power that is larger than 2. Therefore, this property of our algorithm enables, for instance, a fast algorithm for computing APSP in the Congested Clique model. We emphasize that, unlike the matrix multiplication algorithms of [7], we are not aware of a similar sparse matrix multiplication algorithm existing in the literature of parallel settings.

Furthermore, we leverage our techniques to obtain a deterministic algorithm for sparsity-aware triangle listing in the Congested Clique model, in which each triangle needs to be known to some node. This problem has been tackled (implicitly) in the Congested Clique model for the first time by Dolev et al. [10], providing two deterministic algorithms. Later, [23, 16] showed a $\tilde{\Omega}(n^{1/3})$ lower bound in general graphs. Pandurangan et al. [23] showed a *randomized* triangle listing algorithm, with the same round complexity as we obtain.

1.1 Our contribution

For a matrix A, let $nz(A)$ be its number of nonzero elements. Our main contribution is an algorithm called SMM (Sparse Matrix Multiplication), for which we prove the following.

▶ **Theorem 1.** *Given two $n \times n$ matrices S and T, Algorithm SMM deterministically computes the product $P = S \cdot T$ over a semiring in the* Congested Clique *model, completing in $O(nz(S)^{1/3} nz(T)^{1/3}/n + 1)$ rounds.*[1]

An important case of Theorem 1, especially when squaring the adjacency matrix of a graph in order to solve graph problems, is when the sparsities of the input matrices are roughly the same. In such a case, Theorem 1 gives the following.

▶ **Corollary 2.** *Given two $n \times n$ matrices S and T, where $O(nz(S)) = O(nz(T)) = m$, Algorithm SMM deterministically computes the product $P = S \cdot T$ over a semiring in the* Congested Clique *model, within $O(m^{2/3}/n + 1)$ rounds.*

Notice that for $m = O(n^2)$, Corollary 2 gives the same complexity of $O(n^{1/3})$ rounds as given by the semiring multiplication of [7].

We apply Algorithm SMM to 4-cycle counting, obtaining the following.

▶ **Theorem 3.** *There is a deterministic algorithm that computes the number of 4-cycles in an n-node graph G in $O(m^{2/3}/n + 1)$ rounds in the* Congested Clique *model, where m is the number of edges of G.*

Notice that for $m = O(n^{3/2})$ this establishes 4-cycle counting in a constant number of rounds.

As described earlier, our algorithm is fast also in the case where only one of the input matrices is sparse, as stated in the following corollary of Theorem 1.

▶ **Corollary 4.** *Given two $n \times n$ matrices S and T, where $\min\{O(nz(S)), O(nz(T))\} = m$, Algorithm SMM deterministically computes the product $P = S \cdot T$ over a semiring in the* Congested Clique *model, within $O((m/n)^{1/3} + 1)$ rounds.*

This allows us to compute powers that are larger than 2 of a sparse input matrix. Although we cannot enjoy the guarantees of our algorithm when repeatedly squaring a matrix, because this may require multiplying dense matrices, we can still repeatedly increase its power by 1. This gives the following for computing APSP, whose comparison to the state-of-the-art depends on trade-off between the number of edges in the graph and its diameter.

▶ **Theorem 5.** *There is a deterministic algorithm that computes unweighted undirected APSP in an n-node graph G in $O(D((m/n)^{1/3} + 1))$ rounds in the* Congested Clique *model, where m is the number of edges of G and D is its diameter.*

For comparison, the previously known best complexity of unweighted undirected APSP is $O(n^{1-2/\omega})$, given by[7, 14], which is currently known to be bounded by $O(n^{0.158})$. For a graph with a number of edges that is $m = o(n^{4-6/\omega}/D^3)$, which is currently $o(n^{1.474}/D^3)$, our algorithm improves upon the latter.

Lastly, we leverage the routing techniques developed in our sparse matrix multiplication algorithm in order to introduce an algorithm for the triangle listing problem in the Congested Clique model.

[1] Since we minimize communication rather than element-wise multiplications, the *zero element* does not have to be the zero element of the semiring - any *single* element may be chosen to not be explicitly communicated.

▶ **Theorem 6.** *There is a deterministic algorithm for triangle listing in an n-node, m-edge graph G in $O(m/n^{5/3}+1)$ rounds in the* Congested Clique *model.*

For comparison, two deterministic algorithms by Dolev et al. [10] take $\tilde{O}(n^{1/3})$ and $O(\lceil\Delta^2/n\rceil)$ rounds, while the sparsity-aware randomized algorithm of Pandurangan et al. [23] completes in $\tilde{O}(m/n^{5/3})$, w.h.p. Notice that for general graphs, our algorithm matches the lower bound of $\tilde{\Omega}(n^{1/3})$ by[23, 16]. Additionally, our algorithm for triangle listing implies a triangle counting algorithm. A triangle counting algorithm whose complexity depends on the arboricity A of the graph is given in [10]. Their algorithm completes in $O(A^2/n+\log_{2+n/A^2} n)$ rounds. Since $A \geq m/n$, this gives a complexity of $\Omega(m^2/n^3)$, upon which our algorithm provides more than a cubic improvement. The previously known best complexity of triangle and 4-cycle counting in general graphs is $O(n^{1-2/\omega})$, given by[7], which is currently known to be bounded by $O(n^{0.158})$. For a graph with a number of edges that is $m = o(n^{8/3-2/\omega})$, which is currently $o(n^{1.824})$, our algorithm improves upon the latter.

The sections containing *triangle listing*, *APSP*, and *4-cycle counting* are in the full paper [8].

1.2 Challenges and Our Techniques

Given two $n \times n$ matrices S and T, denote their product by $P = S \cdot T$, for which $P[i][j] = \sum_{k=1}^{n} S[i][k]T[k][j]$. A common way of illustrating the multiplication is by a *3-dimensional cube* of size $n \times n \times n$, in which the entry (i, j, k) corresponds to the element-wise product $S[i][k]T[k][j]$. In other words, two dimensions of the cube correspond to the matrices S and T, and the third dimension corresponds to element-wise products. Each index of the third dimension is a *page*, and P corresponds to the element-wise summation of all n pages.

In essence, the task of distributed matrix multiplication is to assign each of the n^3 element-wise multiplications to the nodes of the network, in a way which minimizes the amount of communication that is required.[2] This motivates the goal of assigning the element-wise products to the nodes in a way that balances the number of non-zero elements in S and T that need to be communicated among the nodes, as this is the key ingredient towards minimizing the number of communication rounds. The main obstacle is that a sparse input matrix may be unbalanced, leading to the existence of nodes whose element-wise multiplication operation assignment requires them to obtain many nonzero elements of the input matrices that originally reside in other nodes, and thus necessitating much communication.

As we elaborate upon in Section 1.3, algorithms for the parallel settings, which encounter the same hurdle, typically first permute the rows and columns of the input matrices in an attempt to balance the structure of the non-zero entries. Ballard et al. [3] write: "While a priori knowledge of sparsity structure can certainly reduce communication for many important classes of inputs, we are not aware of any algorithms that dynamically determine and efficiently exploit the structure of general input matrices. In fact, a common technique of current library implementations is to randomly permute rows and columns of the input matrices in an attempt to destroy their structure and improve computational load balance."

Our high-level approach, which is *deterministic*, is threefold. The first ingredient is splitting the $n \times n \times n$ cube into n equally sized sub-cubes whose dimensions are determined dynamically, based on the sparsity of the input matrices. The second is indeed permuting the input matrices S and T into two matrices S' and T', respectively. We do so in a subtle

[2] We consider all n^3 element-wise multiplications rather than Strassen-like algorithms since we work over a semiring and not a ring.

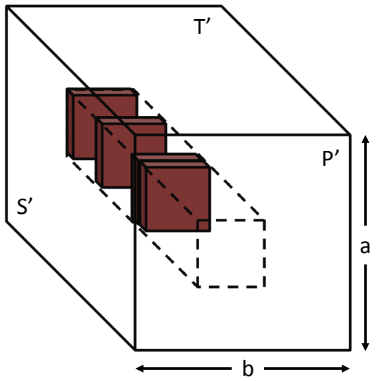

Figure 1 An illustration of the multiplication cube for $P' = S'T'$. Each sub-matrix is assigned to n/ab nodes, with a not necessarily consecutive page assignment that is computed on-the-fly to minimize communication.

manner, for which the resulting matrices exhibit some nice balancing property.[3] The third ingredient is the innovative part of our algorithm, which assigns the computation of pages of different sub-matrices across the nodes in a *non-consecutive* manner. We elaborate below about these key ingredients, with the aid of Figure 1.

Permuting the input matrices: We employ standard parallelization of the task of computing the product matrix P, by partitioning P into ab equal sized $n/a \times n/b$ sub-matrices denoted by $P_{i,j}$ for $i \in [a], j \in [b]$, and assigning n/ab nodes for computing each sub-matrix.

To this end, we leverage the simple observation that the multiplication of permutations of the rows of S and the columns of T results in a permutation of the product of S and T, which can be easily inverted. This observation underlies the first part of our algorithm, in which the nodes permute the input matrices, such that the number of non-zero entries from S and T that are required for computing each $n/a \times n/b$ sub-matrix are roughly the same across the $a \cdot b$ sub-matrices. We call the two matrices, S' and T', that result from the permutations, *sparsity-balanced matrices with respect to* (a, b). The rest of our algorithm deals with computing the product of two such matrices. This part inherently includes a computation of the best choice for a and b for minimizing the communication.

Assigning pages to nodes: To obtain each sub-matrix $P_{i,j}$, there are n sub-pages $P_{i,j,\ell}$ which need to be computed and summed. For each $P_{i,j}$, this task is assigned to distinct n/ab nodes, each of which computes some of the n sub-pages $P_{i,j,\ell}$ and sums them locally. The local sums are then aggregated and summed, for obtaining $P_{i,j}$. We utilize the commutativity and associativity properties of summation over the semiring in order to assign sub-pages to nodes in a *non-consecutive* manner, such that the nodes require receiving a roughly equal number of non-zero entries in order to compute their assigned sub-pages.

[3] Note, we do not assume that balancing the distribution of non-zero elements gives a balanced local computation. Our balancing is done for the amount of communication: we assign the amount and identity of matrix entries that should be sent and received by each node in a way that will balance the communication, not necessarily the local computation.

Assigning non-zero matrix entries to nodes: For fast communication in the Congested Clique model using Lenzen's routing scheme (see Section 1.4), it is moreover paramount that the nodes also *send* a roughly equal amount of non-zero matrix entries. However, it may be the case that a certain row, held by a node v, contains a significantly larger number of non-zero entries as compared with other rows. Therefore, we rearrange the entries held by each node such that every node holds a roughly equal amount of non-zero entries that need to be sent to other nodes for computing the n^3 products. Notice that in this step we do not rearrange the rows or columns of S or T, rather, we redistribute the entries of S and T. Thus, a node may hold values which originate from different rows.

Routing non-zero elements: Crucially, the assignments made above, for addressing the need to balance sending and receiving, are not global knowledge. That is, for every $P_{i,j}$, the corresponding n/ab nodes decide which matrix entries are received by which node, but this is unknown to the other nodes, who need to send this information. Likewise, the redistribution of entries of S and T across the nodes is not known to all nodes. Nonetheless, clearly, a node must know the destination of each message it needs to send. As a consequence, we ultimately face the challenge of communicating some of this local knowledge. In our solution, a node that needs to receive information from a certain column of S (or row of T) sends a request to the nodes holding subsequences of that column (or row) without knowing the exact partition into subsequences. The nodes then deliver the non-zero entries of this column (or row), which allow computing the required element-wise multiplications.
Our solutions to the three challenges described above, for *sending* and *receiving* as small as possible amounts of information and for resolving a corresponding *routing*, and their combination, are the main innovation of our algorithm.

1.3 Related work

Matrix multiplication in the Congested Clique model: A randomized Boolean matrix multiplication algorithm was given by Drucker et al. [11], completing in $O(n^{\omega-2})$ rounds, where ω is the exponent of sequential matrix multiplication. The best currently known upper bound is $\omega < 2.3728639$ [13], implying $O(n^{0.372})$ rounds for the above.

Later, Censor-Hillel et al. [7] gave a deterministic algorithm for (general) matrix multiplication over semirings, completing in $O(n^{1/3})$ rounds, and a deterministic algorithm for (general) matrix multiplication over rings, completing in $O(n^{1-2/\omega})$ rounds, which by the current known upper bound on ω is $O(n^{0.158})$. The latter is a Strassen-like algorithm, exploiting known schemes for computing the product of two matrices over a ring without directly computing all n^3 element-wise multiplications. Then, Le Gall [14] provided fast algorithms for multiplying rectangular matrices and algorithms for computing multiple instances of products of independent matrices.

Related graph computations in the Congested Clique model: Triangle counting in the Congested Clique model was addressed by Dolev et al. [10], who provided a deterministic $\tilde{O}(n^{d-2/d})$-round algorithm for counting the number of appearances of any d-node subgraph, giving triangle counting in $\tilde{O}(n^{1/3})$ rounds. To speed up the computation for sparse instances, [10] show that every node in a graph with a maximum degree of Δ can learn its 2-hop neighborhood within $O(\Delta^2/n)$ rounds, implying the same round complexity for triangle counting. They also showed a deterministic triangle counting algorithm completing in $\tilde{O}(A^2/n + \log_{2+n/A^2} n)$ rounds, where A is the arboricity of the input graph, i.e., the minimal number of forests into which the set of edges can be decomposed. Note that a graph with

arboricity A has at most An edges, but there are graphs with arboricity A and a significantly smaller number of edges. Since it holds that $A \geq m/n$, this implies a complexity of $\Omega(m^2/n^3)$ for their triangle counting algorithm, upon which our $O(m^{2/3}/n+1)$-round algorithm provides a cubic improvement. The deterministic matrix multiplication algorithm over rings of [7] directly gives a triangle counting algorithm with $O(n^{1-2/\omega})$ rounds.

For 4-cycle counting, the algorithm of [10] completes in $\tilde{O}(n^{1/2})$ rounds, and the matrix multiplication algorithm of [7] implies a solution in $O(n^{1-2/\omega})$ rounds.

For APSP, the matrix multiplication algorithms of [7] give $O(n^{1-2/\omega})$ for the unweighted undirected case. For weighted directed APSP, $\tilde{O}(n^{1/3})$ rounds are given in [7], and improved algorithms for weighted (directed and undirected) APSP are given in [14]. We mention that our technique could allow for computing weighted APSP, but the cost would be too large due to our iterative multiplication (as opposed to the previous algorithms that can afford iterative *squaring*). Algorithms for approximations of APSP are given in [22, 7, 14].

Note that for all graph problems, Lenzen's routing scheme [21] (see Section 1.4) implies that every node can learn the entire structure of G within $O(m/n)$ rounds, where m is the number of edges (this can also be obtained by a simpler scheme).

Sequential/Parallel matrix multiplication: Additional related work on sequential and parallel matrix multiplication can be found in the full paper [8].

1.4 Preliminaries

Model: The Congested Clique model consists of a set $[n] = \{1, \ldots, n\}$ of nodes in a fully connected synchronous network, limited by a bandwidth of $O(\log n)$ bits per message.

In an instance of multiplication of two matrices S and T, the input to each node v is row v of each matrix and its output should be row v of $P = S \cdot T$. For a graph problem over a graph G of n nodes, we identify the nodes of the Congested Clique model with the nodes of G, and the input to node v in the Congested Clique model is its input in G.

As defined earlier, for a matrix A we denote by $nz(A)$ the number of non-zero elements of A. Throughout the paper, we also need to refer to the number of non-zero elements in certain sub-matrices or sequences. We will therefore overload this notation, and use $nz(X)$ to denote the number of non-zero elements in any object X.

A pair of integers (a, b) is *n-split* if $a, b \in [n]$, both a and b divide n, and $n/ab \geq 1$. The requirement that a and b divide n is for simplification only and could be omitted. Eventually, the n-split pair that will be chosen is $a = n \cdot nz(S)^{1/3}/nz(T)^{2/3}$ and $b = n \cdot nz(T)^{1/3}/nz(S)^{2/3}$.

For a given n-split pair (a, b), it will be helpful to associate each node $v \in [n]$ with three indices, two indicating the $P_{i,j}$ sub-matrix to which the node is assigned, and one distinguishing it from the other nodes assigned to $P_{i,j}$. Hence, we denote each node v also as $v_{i,j,k}$, where $i \in [a]$, $j \in [b]$, and $k \in [n/ab]$. The assignment of indices to the nodes can be any arbitrary one-to-one function from $[n]$ to $[a] \times [b] \times [n/ab]$.

Throughout our algorithms, we implicitly comply with the following: (I) no information is sent for matrix entries whose value is zero, and (II) when the value of a non-zero entry is sent, it is sent alongside its location in the matrix. Since sending the location within an $n \times n$ matrix requires $O(\log n)$ bits, the overhead in the complexity is constant.

Lenzen's routing scheme: A useful tool in designing algorithms for the Congested Clique model is *Lenzen's routing scheme* [21]. In this scheme, each of the n nodes can send and receive $n-1$ messages (of $O(\log n)$ bits each) in a constant number of rounds. While this is simple to see for the simplest case where each node sends a single message to every other

node, the power of Lenzen's scheme is that it applies to any (multi)set of source-destination pairs, as long as each node is source of at most $n-1$ messages and destination of at most $n-1$ messages. Moreover, the multiset of pairs does not need to be known to all nodes in advance, rather each sender only needs to know the recipient of its messages. Employing this scheme is what underlies our incentive for balancing the number of messages that need to be sent and received by all the nodes.

Useful combinatorial claims: The following are simple combinatorial claims that we use for routing messages in a load-balanced manner.

▶ Claim 1. Let $A = (a_1, \ldots, a_t)$ be a finite set and let $1 \leq c \leq t$ be an integer. There exists a partition of A into $\lceil t/c \rceil$ subsets of size at most $c+1$ each.

▶ Claim 2. Given any n finite sets, $A^i = (a^i_1, \ldots, a^i_{t_i})$ for $1 \leq i \leq n$. Let $avg = (\sum_{1 \leq i \leq n} t_i)/n$. There exists a partition of each A^i into $\lceil t_i/avg \rceil$ subsets of size at most $avg+1$ each, such that the total number of subsets is at most $2n$.

▶ Claim 3. Given a sorted finite multiset $A = (a_1, \ldots, a_n)$ of natural numbers, an integer $x \in \mathbb{N}$ such that for all $i \in [n]$ it holds that $a_i \leq x$, and an integer k that divides n, there exists a partition $A = \cup_{i=j}^{k} A_j$ into k multisets A_j, $1 \leq j \leq k$, of equal size n/k, such that for all $1 \leq j \leq k$ it holds that $sum(A_j) \leq sum(A)/k + x$.

2 Fast Sparse Matrix Multiplication

Our main result is Theorem 1, stating the guarantees of our principal algorithm SMM (Sparse Matrix Multiplication) for fast multiplication of sparse matrices. Algorithm SMM first manipulates the structure of its input matrices and then calls algorithm SBMM (Sparse Balanced Matrix Multiplication), which solves the problem of fast sparse matrix multiplication under additional assumptions on the distributions of non-zero elements in the input matrices, which are defined next. In Section 2.1, we show how SMM computes general matrix multiplication $P = ST$, given Algorithm SBMM and Theorem 8. Algorithm SBMM, and Theorem 8 which states its guarantees, are deferred to Section 2.2.

Theorem 1 (restated). *Given two $n \times n$ matrices S and T, Algorithm SMM deterministically computes the product $P = S \cdot T$ over a semiring in the* Congested Clique *model, completing in $O(nz(S)^{1/3} nz(T)^{1/3}/n + 1)$ rounds.*

We proceed to presenting Theorem 8 which discusses SBMM. SBMM multiplies matrices S' and T' in which the non-zero elements are roughly balanced between portions of the rows of S' and columns of T'. In what follows, for a matrix A, the notation $A[x:y][*]$ refers to rows x through y of A and the notation $A[*][x:y]$ refers to columns x through y of A. In the following definition we capture the needed properties of well-balanced matrices.

▶ **Definition 7.** Let S and T be $n \times n$ matrices and let (a, b) be an n-split pair. For every $i \in [a]$ and $j \in [b]$, denote $S_i = S[(i-1)(n/a)+1 : i(n/a)][*]$ and $T_j = T[*][(j-1)(n/b)+1 : j(n/b)]$. We say that S and T are a *sparsity-balanced pair of matrices with respect to* (a, b), if:

- S-condition: For every $i \in [a]$, $nz(S_i) \leq nz(S)/a + n$.
- T-condition: For every $j \in [b]$, $nz(T_j) \leq nz(T)/b + n$.

These conditions ensure that *bands* of adjacent rows of S and columns of T contain roughly the same number of non-zero elements. We can now state our theorem for multiplying sparsity-balanced matrices, which summarizes our algorithm SBMM.

▶ **Theorem 8.** *Given two $n \times n$ matrices S and T and an n-split pair (a, b), if S and T are a sparsity-balanced pair with respect to (a, b), then Algorithm SBMM deterministically computes the product $P = S \cdot T$ over a semiring in the* Congested Clique *model, completing in $O(nz(S) \cdot b/n^2 + nz(T) \cdot a/n^2 + n/ab)$ rounds.*

We show that $O(1)$ rounds are sufficient in the Congested Clique for transforming any two general matrices S and T to sparsity-balanced matrices S' and T' by invoking standard matrix permutation operations. Therefore, in essence, Algorithm SMM performs permutation operations on S and T, generating the matrices S' and T', respectively, invokes SBMM on S' and T' to compute $P' = S'T'$, and finally recovers P from P'.

2.1 Fast General Sparse Matrix Multiplication - Algorithm SMM

Algorithm Description: First, each node distributes the entries in its row of T to other nodes in order for each node to obtain its column in T. Then, the nodes broadcast the number of non-zero elements in their respective row of S and column of T, in order for all nodes to compute $nz(S)$ and $nz(T)$. Having this information, the nodes locally compute the n-split pair (a, b) that minimizes the expression $nz(S) \cdot b/n^2 + nz(T) \cdot a/n^2 + n/ab$, which describes the round complexities of each of the three parts of Algorithm SBMM. It can be shown that the pair $(n \cdot nz(S)^{1/3}/nz(T)^{2/3}, n \cdot nz(T)^{1/3}/nz(S)^{2/3})$ minimizes this expression. Then, the nodes permute the rows of S and columns of T so as to produce matrices S' and T' which have the required balance. Subsequently, Algorithm SBMM is executed on the permuted matrices S' and T', followed by invoking the inverse permutations on the product $P' = S'T'$ in order to obtain the product $P = S \cdot T$ of the original matrices. A pseudocode of SMM is given in Algorithm 1.

Proof of Theorem 1. To prove correctness, we need to show that the matrices S' and T' computed in Line 10 are a sparsity-balanced pair of matrices with respect to the n-split pair (a, b) that is determined in Line 8. Once this is proven, the correctness of the algorithm is as follows. In Lines 1-12 the matrices S' and T' are computed and are distributed among the nodes such that each node $v \in [n]$ holds row v of S and column v of T. The loop of Line 13 is only for consistency, having the input to SBMM be the respective rows of both S' and T'. Assuming the correctness of algorithm SBMM given in Theorem 8, the matrix P' computed in Line 15 is the product $P' = S'T'$. Finally, in the last loop, node v receives row v of $P = A_\sigma^{-1} P' A_\tau^{-1}$, completing the correctness of the Algorithm SMM.

We now show that S' and T' are indeed a sparsity-balanced pair of matrices with respect to (a, b). To this end, we first need to show that for all $i \in [a]$, the number of non-zero elements in S'_i is at most $nz(S')/a + n$. By construction, the number of non-zero elements in $S'_i = S'[(i-1)(n/a) + 1 : i(n/a)][*]$ is exactly $sum(A_i^S)$ of the partition computed in Line 9. By Claim 3 this is bounded by $sum(A)/k + x$, which in our case is $nz(S)/a + n = nz(S')/a + n$. Thus, S' satisfies the S-condition of Definition 7. A similar argument shows that T' satisfies the T-condition of Definition 7.

For the complexity, we sum the number of rounds as follows. The first loop allows every node v to obtain column v of T, while in the second loop the nodes exchange the sums of non-zero elements in rows and columns of S and T, respectively. Even without the need to resort to Lenzen's routing scheme, both of these loops can be completed within $O(1)$ rounds. A similar argument shows that $O(1)$ rounds suffice for permuting S and T into S' and T', and for permuting P' back into P. Thus, all lines of the pseudocode excluding Line 15 complete in $O(1)$ rounds. This implies that the complexity of Algorithm SMM equals that of Algorithm SBMM when given S', T', a, and b as input. By Theorem 8 and due to the choice of a and

Algorithm 1: SMM (S,T): Computing the product $P = S \cdot T$. Code for node $v \in \{1,\ldots,n\}$.

1 **foreach** $u \in [n]$, $u \neq v$ **do**
2 **send** $T[v][u]$ to node u
3 **foreach** $u \in [n]$, $u \neq v$ **do**
4 **send** $nz(S[v][*])$ to node u
5 **send** $nz(T[*][v])$ to node u
6 $nz(S) \leftarrow \sum_{u\in[n]} nz(S[u][*])$
7 $nz(T) \leftarrow \sum_{u\in[n]} nz(T[*][u])$
8 $(a,b) \leftarrow argmin_{n\text{-split pairs } (a,b)}\{nz(S)\cdot b/n^2 + nz(T)\cdot a/n^2 + n/ab\}$
9 Let $A_1^S,\ldots,A_a^S$ be the partition of the sorted multiset of $\{nz(S[u][*])|u\in[n]\}$ into a multisets, and $A_1^T,\ldots,A_b^T$ be the partition of the sorted multiset of $\{nz(T[*][u])|u\in[n]\}$ into b multisets, both proven to exist in Claim 3 (with $x=n$).
10 Let σ be a permutation for which its $n\times n$ permutation matrix A_σ is such that the rows of the matrix $S' = A_\sigma S$ that correspond to any single A_u^S are adjacent, and let τ be a permutation for which its $n\times n$ permutation matrix A_τ is such that the columns of the matrix $T' = TA_\tau$ that correspond to any single A_u^T are adjacent.
11 **send** $S[v][*]$ to node $\sigma(v)$
12 **send** $T[*][v]$ to node $\tau(v)$
13 **foreach** $u \in [n]$, $u \neq v$ **do**
14 **send** $T'[u][v]$ to node u
15 $P' \leftarrow \text{SBMM}(S',T',a,b)$
16 **foreach** $u \in [n]$, $u \neq v$ **do**
17 **send** $P'[\sigma^{-1}(v)][\tau^{-1}(u)]$ to node u

b in Line 8, this complexity is $O(\min_{n\text{-split pairs } (a,b)}\{nz(S)\cdot b/n^2 + nz(T)\cdot a/n^2 + n/ab\})$. Choosing $a = n\cdot nz(S)^{1/3}/nz(T)^{2/3}$ and $b = n\cdot nz(T)^{1/3}/nz(S)^{2/3}$ gives a complexity of $O(nz(S)^{1/3}nz(T)^{1/3}/n+1)$ rounds, which can be shown to be optimal. ◀

2.2 Fast Sparse Balanced Matrix Multiplication - Algorithm SBMM

Here we present SBMM and prove Theorem 8. We begin with a short overview of the algebraic computations and node allocation in SBMM. We then proceed to presenting a communication scheme detailing how to perform the computations of SBMM in the Congested Clique model in $O(M_S\cdot b/n^2 + M_T\cdot a/n^2 + n/ab)$ rounds of communication.

Algorithm Description: Consider the partition of P into ab rectangles, such that $\forall(i,j)\in[a]\times[b]$, *sub-matrix* $P_{i,j} = P[(i-1)(n/a)+1 : i(n/a)][(j-1)(n/b)+1 : j(n/b)]$. Each sub-matrix $P_{i,j}$ is an $n/a \times n/b$ matrix, i.e., has n^2/ab entries. Notice that $P_{i,j} = S_i \cdot T_j$. We assign the computation of $P_{i,j}$ to a unique set of n/ab nodes $N_{i,j} = \{v_{i,j,k}|k\in[n/ab]\}$.

In the initial phase of algorithm SBMM, for every $(i,j)\in[a]\times[b]$, each non-zero element of S_i and T_j is sent to some node in $N_{i,j}$. Due to the sparsity-balanced property of S and T, all S_i's have roughly the same amount of non-zero elements, and likewise all T_j's. Therefore, each set of nodes $N_{i,j}$ receives roughly the same amount of non-zero elements from S and T.

Within each $N_{i,j}$, the computation of $P_{i,j}$ is carried out according to the following framework. For $\ell\in[n]$, denote each page of $P_{i,j}$ by $P_{i,j,\ell} = S_i[*][\ell]\cdot T_j[\ell][*]$. The computation

Algorithm 2: SBMM (S,T,a,b): Computing the product $P = ST$, for S and T that are sparsity-balanced w.r.t. (a, b). Code for node $v_{i,j,k}$.

1 ExchangeInfo (S, T, a, b)
2 Locally compute $P_{i,j,\ell}$ for every $\ell \in A_{i,j,k}$
3 Locally compute $P^k_{i,j} = \sum_{\ell \in A_{i,j,k}} P_{i,j,\ell}$
4 **foreach** $t \in [n/a]$ **do**
5 ⌊ **send** $P^k_{i,j}[t][*]$ to node of respective row
6 **foreach** $\ell \in [n]$ **do**
7 ⌊ $P[v][\ell] \leftarrow$ sum of n/ab respective elements received for this entry

of the n different $P_{i,j,\ell}$ sub-matrices is split among the nodes in $N_{i,j}$ as follows: The set $[n]$ is partitioned into $A_{i,j,1}, \ldots, A_{i,j,n/ab}$ such that for each $k \in [n/ab]$, node $v_{i,j,k} \in N_{i,j}$ is required to compute the entries of the matrices in the set $\{P_{i,j,\ell} | \ell \in A_{i,j,k}\}$. Then, node $v_{i,j,k} \in N_{i,j}$ locally sums its computed sub-matrices to produce $P^k_{i,j} = \sum_{\ell \in A_{i,j,k}} P_{i,j,\ell}$. Clearly, due to the associativity and commutativity of the addition operation in the semiring, it holds that $P_{i,j} = \sum_{\ell \in [n]} P_{i,j,\ell} = \sum_{k \in [n/ab]} P^k_{i,j}$. Therefore, once every node $v_{i,j,k}$ has $P^k_{i,j}$, the nodes can collectively compute P, and redistribute its entries in a straightforward manner such that each node obtains a distinct row of P.

Implementing SBMM: A pseudocode for Algorithm SBMM is given in Algorithm 2, which consists of three components: exchanging information between the nodes such that every node $v_{i,j,k}$ has the required information for computing $P_{i,j,\ell}$ for every $\ell \in A_{i,j,k}$, local computation of $P^k_{i,j}$ for each $(i, j, k) \in [a] \times [b] \times [n/ab]$ and, finally, the communication of the $P^k_{i,j}$ matrices and assembling of the rows of P.

The technical challenge is in Line 1, upon which we elaborate below. In Lines 2-3, only local computations are performed, resulting in each node $v_{i,j,k}$ holding $P^k_{i,j}$. In Line 4, each node sends each row of its sub-matrix $P^k_{i,j}$ to the appropriate node, so that in Line 6 each node can sum this information to produce its row in P. In the full version [8], we prove Lemma 9.

▶ **Lemma 9.** *Lines 2-6 of Algorithm 2 complete in $O(n/ab)$ rounds, producing a row of $P = ST$ for every node.*

The remainder of this section is dedicated to presenting and analyzing Line 1. During this part of the algorithm, for every $(i, j) \in [a] \times [b]$, each entry in S_i and T_j needs to be sent to a node in $N_{i,j}$. As per our motivation throughout the entire algorithm, we strive to achieve this goal in a way which ensures that all nodes send and receive roughly the same number of messages. This leads to the following three challenges which we need to overcome.

Sending Challenge: Initially, node v holds row v of S and row v of T. Every column v of S needs to be sent to b nodes - one node in each $N_{i,j}$ for an appropriate $i \in [a]$ and every $j \in [b]$. Similarly, every row of T needs to be sent to a nodes - one in each $N_{i,j}$ for an appropriate $j \in [b]$ and every $i \in [a]$. If we were to trivially choose node v to send all these messages, then node v would need to send $nz(S[*][v]) \cdot b + nz(T[v][*]) \cdot a$ messages. Since $nz(S[*][v])$ and $nz(T[v][*])$ may widely vary for different values of v, it may be the case that some nodes send a significant amount of messages while others are relatively silent.

Algorithm 3: ExchangeInfo (S,T,a,b): Sending each entry of S_i, T_j to a node in $N_{i,j}$, for every $(i,j) \in [a] \times [b]$.

1 Compute-Sending
2 Compute-Receiving
3 Resolve-Routing

Receiving Challenge: Since S and T are sparsity-balanced w.r.t. (a,b), for every $(i,j) \in [a] \times [b]$ it holds that the number of messages to be received by each set of nodes $N_{i,j}$ is at most $nz(S)/a + nz(T)/b + 2n$. This ensures that each node set $N_{i,j}$ receives roughly the same amount of messages as every other node set. The challenge remains to ensure that *within* any given node set $N_{i,j}$, every node receives roughly the same number of messages.

Routing Challenge: When overcoming the above mentioned challenges in a non-trivial manner, all nodes locally determine that they are senders and recipients of certain messages with the guarantee that each node sends and receives roughly the same number of messages. However, these partitions of sending and receiving messages are obtained independently and thus are not global knowledge; a sender of a message *does not necessarily know* who the recipient is. The routing challenge is thus to ensure that each node associates the correct recipient with every message that it sends.

2.2.1 ExchangeInfo (S, T, a, b)

We next present our implementation of ExchangeInfo (S,T,a,b) which solves the above challenges in an on-the-fly manner. To simplify the presentation, we split ExchangeInfo (S,T,a,b) into its three components, as given in the pseudocode of Algorithm 3.

Compute-Sending: In Compute-Sending, whose pseudocode is given in Algorithm 4, we overcome the sending challenge. The nodes communicate the distribution of non-zero elements across the columns of S and the rows of T and reorganize the entries held by each node such that all nodes hold roughly the same amount of non-zero elements of S and T.

Notably, in order to enable fast communication in Resolve-Routing, Algorithm 4 must guarantee no node holds entries of more than two columns of S and two rows of T.

▶ **Lemma 10.** *Algorithm 4 completes in $O(1)$ rounds, after which the entries of S and T are evenly redistributed across the nodes such that every node holds elements from at most 2 columns of S and 2 rows of T and such that every node v knows for every node u the indices of the two columns of S and two rows of T from which the elements which u holds are taken.*

Proof. In Lines 2, 4, 5 the nodes exchange entries of S such that each node holds a distinct column of S, and knows the number of non-zero entries in each column of S and in each row of T. This allows local computation of the average number of non-zeros in the following two lines, as well as locally computing the (same) partition into subsequences.

By Claim 2, in total across all n columns there are at most $2n$ subsequences of entries from S, and similarly there are at most $2n$ subsequences from T. Since $\forall u \in [n]$, all nodes know $nz(S[*][u])$ and $nz(T[u][*])$, then all nodes know how many subsequences are created for each u. Thus, all nodes can agree in Line 9 on the assignment of the subsequences, with each node assigned at most 2 subsequences of entries of S and 2 of entries of T. Crucially for what follows, all the nodes know the column ℓ in S or the row ℓ in T to which the

Algorithm 4: Compute-Sending: Code for node $v_{i,j,k}$.

1 **foreach** $u \in [n]$, $u \neq v$ **do**
2 **send** $S[u][v]$ to node u
3 **foreach** $u \in [n]$, $u \neq v$ **do**
4 **send** $nz(S[*][v])$ to node u
5 **send** $nz(T[v][*])$ to node u
6 $avg(S) \leftarrow (\sum_{u\in[n]} nz(S[*][u]))/n$
7 $avg(T) \leftarrow (\sum_{u\in[n]} nz(T[u][*]))/n$
8 Let $S^v_1, \ldots, S^v_{\lceil nz(S[*][v])/avg(S)\rceil}$ be a partition of the non-zero elements of $S[*][v]$ into sets of size at most $avg(S)+1$ and let $T^v_1, \ldots, T^v_{\lceil nz(T[v][*])/avg(T)\rceil}$ be a partition of the non-zero elements of $T[v][*]$ into sets of size at most $avg(T)+1$, both proven to exist in Claim 1. We refer to these sets as *subsequences*.
9 Assign two subsequences of S, denote by $B_S(v)$, and two subsequences of T, denote by $B_T(v)$ to each node v. For each subsequence B, denote by $v(B)$ the node to which B is assigned.
10 **foreach** $B \in \{S^v_1, \ldots, S^v_{\lceil nz(S[*][v])/avg(S)\rceil}, T^v_1, \ldots, T^v_{\lceil nz(T[v][*])/avg(T)\rceil}\}$ **do**
11 **send** B to node $v(B)$

subsequence B belongs. We denote this index $\ell(B)$. The entries of each subsequence B are then sent to its node $v(B)$ in the following loop.

For the round complexity, note that a node v sends a single message to every other node in each of Lines 2, 4, and 5. The rest of the computation until Line 9 is done locally. Therefore, these lines complete within 3 rounds.

In the last loop of Algorithm 4, node v potentially sends all subsequences with entries from column v of S and row v of T. Due to the facts that each subsequence is sent only once, no subsequences overlap, and all the subsequences which v send are parts of a single column of S and a single row of T, node v sends at most $2n$ messages during this loop. Additionally, since every node receives at most 4 subsequences and each subsequence consists of most n entries, each node receives at most $4n$ messages. Thus, by using Lenzen's routing scheme, this completes in $O(1)$ rounds as well. ◀

Compute-Receiving: The pseudocode for Compute-Receiving is given in Algorithm 5. This algorithm assigns the $P_{i,j,\ell}$ matrices to different nodes in $N_{i,j}$. Specifically, each node $v_{i,j,k}$ in $N_{i,j}$ is assigned ab such matrices, while verifying that all nodes in $N_{i,j}$ require roughly the same amount of non-zero entries from S and T in order to compute all their assigned $P_{i,j,\ell}$ matrices. Since each sub-matrix $P_{i,j,\ell}$ is defined as $P_{i,j,\ell} = S_i[*][\ell] \cdot T_j[\ell][*]$, we define the communication cost of computing $P_{i,j,\ell}$ to be $w(P_{i,j,\ell}) = nz(S_i[*][\ell]) + nz(T_j[\ell][*])$. By this definition, in order to obtain that each node in $N_{i,j}$ requires roughly the same amount of messages in order to compute all of its assigned $P_{i,j,\ell}$, we assign the $P_{i,j,\ell}$ matrices to the nodes of $N_{i,j}$ such that the total communication cost, as measured by w, of all matrices assigned to a given node is roughly the same for all nodes.

▶ **Lemma 11.** *Algorithm 5 completes in $O(1)$ rounds, after which each node $v_{i,j,k}$ is assigned a subset $A_{i,j,k} \subseteq [n]$, s.t. $\forall k \in [n/ab]$ it holds that $\sum_{\ell\in A_{i,j,k}} w(P_{i,j,\ell}) \leq \frac{n}{ab} \sum_{\ell\in[n]} w(P_{i,j,\ell}) + 2n$.*

Proof. The loop of Line 1 provides each node of $N_{i',j'}$ with the number of non-zero elements in each column of $S_{i'}$ and each row of $T_{j'}$. This allows the nodes to compute the required

Algorithm 5: Compute-Receiving: Code for node $v_{i,j,k}$.

```
1  foreach N_{i',j'}, i', j' ∈ [a] × [b] do
2      foreach u ∈ N_{i',j'} do
3          foreach B ∈ B_S(v) do
4              send nz(S_{i'} ∩ B) to node u
5          foreach B ∈ B_T(v) do
6              send nz(T_{j'} ∩ B) to node u
7  foreach ℓ ∈ [n] do
8      w(P_{i,j,ℓ}) ← nz(S_i[∗][ℓ]) + nz(T_j[ℓ][∗])
9  Let A'_{i,j,1}, ..., A'_{i,j,n/ab} be a partition of the sorted multiset {w(P_{i,j,ℓ}) | ℓ ∈ [n]} into
       n/ab multisets with a bound x = 2n on its elements, proven to exist in Claim 3.
10 Let A_{i,j,1}, ..., A_{i,j,n/ab} be a partition of [n] such that for every k ∈ [n/ab],
       A'_{i,j,k} = {w(P_{i,j,ℓ}) | ℓ ∈ A_{i,j,k}}. .
```

communication costs in Line 7. Claim 3 implies that after executing Line 10, each node $v_{i,j,k}$ is assigned a subset $A_{i,j,k} \subseteq [n]$, such that for every $k \in [n/ab]$ it holds that $\sum_{\ell \in A_{i,j,k}} w(P_{i,j,\ell}) \leq \frac{n}{ab} \sum_{\ell \in [n]} w(P_{i,j,\ell}) + 2n$.

Every node sends every other node exactly 4 messages throughout the loop in Line 1, while the remaining lines are executed locally for each node, without communication. As such, this completes in $O(1)$ rounds in total. ◀

Resolve-Routing: Roughly speaking, we solve this challenge by having the recipient of each possibly non-zero entry deduce which node is the sender of this entry, and inform the sender that it is its recipient, as follows. At the end of the execution of Compute-Sending in Algorithm 4, every node v has at most two subsequences in $B_S(v)$ and at most two subsequences in $B_T(v)$. Moreover, the subsequence assignment is known to all nodes due to performing the same local computation in Line 9. On the other hand, upon completion of Algorithm 5, node $v_{i,j,k}$ is assigned the task of computing $P_{i,j,\ell}$ for every $\ell \in A_{i,j,k}$. For this, it suffices for $v_{i,j,k}$ to know the non-zero entries of column ℓ of S_i and of row ℓ of T_j.

Hence, in Resolve-Routing, given in Algorithm 6, node $v_{i,j,k}$ sends every index $\ell \in A_{i,j,k}$ to the nodes that hold subsequences of column ℓ in S and row ℓ in T. Notice that $v_{i,j,k}$ does not know which indices inside these columns and rows are non-zero. However, the nodes which hold these subsequences have this information, and respond with the non-zero entries of the respective columns and rows that are part of S_i or T_j. The proof of the following Lemma 12 appears in the full version [8].

▶ **Lemma 12.** *Algorithm 6 completes in $O(nz(S) \cdot b/n^2 + nz(T) \cdot a/n^2 + 1)$ rounds, after which each node $v_{i,j,k}$ has $S[(i-1)(n/a)+1 : i(n/a)][\ell]$ and $T[\ell][(j-1)(n/b)+1 : j(n/b)]$, for every $\ell \in A_{i,j,k}$.*

Proof of Theorem 8. Lemma 12 implies that each node $v_{i,j,k}$ has the required entries of S and T in order to compute $P_{i,j,\ell}$ for every $\ell \in A_{i,j,k}$. Lemma 9 then gives that Algorithm SBMM correctly produces a row of $P = ST$ for each node.

Lemmas 10 and 11 show that Compute-Sending and Compute-Receiving complete in $O(1)$ rounds. Lemma 12 gives the claimed round complexity of $O(nz(S) \cdot b/n^2 + nz(T) \cdot a/n^2 + 1)$

Algorithm 6: Resolve-Routing: Code for node $v_{i,j,k}$.

```
1  foreach ℓ ∈ A_{i,j,k} do
2      foreach node u for which there exists B ∈ B_S(u) such that ℓ(B) = ℓ do
3          send ℓ to node u
4      foreach node u for which there exists B ∈ B_T(u) such that ℓ(B) = ℓ do
5          send ℓ to node u
6  foreach message ℓ received from node v_{i',j',k'} in Line 3 do
7      foreach B ∈ B_S(v) do
8          send S[(i'−1)(n/a)+1 : i'(n/a)][ℓ] ∩ B to node v_{i',j',k'}
9  foreach message ℓ received from node v_{i',j',k'} in Line 5 do
10     foreach B ∈ B_T(v) do
11         send T[ℓ][(j'−1)(n/b)+1 : j'(n/b)] ∩ B to node v_{i',j',k'}
```

for Resolve-Routing, giving the same total number of rounds for ExchangeInfo. By Lemma 9, the remainder of Algorithm SBMM completes in $O(n/ab)$ rounds, completing the proof. ◀

3 Discussion

This work significantly improves upon the round complexity of multiplying two matrices in the distributed Congested Clique model, for input matrices which are sparse. As mentioned, we are unaware of a similar algorithmic technique being utilized in the literature of parallel computing, which suggests that our approach may be of interest in a more general setting. The central ensuing open question left for future reserach is whether the round complexity of sparse matrix multiplication in the Congested Clique can be further improved.

Finally, an intriguing question is the complexity of various problems in the more general k-machine model [18, 23], where the size of the computation clique is $k << n$. The way of partitioning the data to the nodes is of importance. One may assume that the input to each node consists of n/k unique consecutive rows of S and T, and its output should be the corresponding n/k rows of the product $P = S \cdot T$. Applying our algorithm in this setting gives a round complexity of $O(\min_{n\text{-split pairs } (a,b)} n^2/k^2 + nz(S) \cdot b/k^2 + nz(T) \cdot a/k^2 + n^2/kab + 1)$ rounds, which is $O(n^{2/3} \cdot nz(S)^{1/3} nz(T)^{1/3}/k^{5/3} + 1)$ rounds with the assignment $a = n^{2/3}k^{1/3} \cdot nz(S)^{1/3}/nz(T)^{2/3}$ and $b = n^{2/3}k^{1/3} \cdot nz(T)^{1/3}/nz(S)^{2/3}$. To see why, consider each node as simulating the behavior of n/k *virtual* nodes of the Congested Clique model that belong to the same $N_{i,j}$ set. The round complexity of all steps of the algorithm grows by a multiplicative factor of n^2/k^2, apart from the steps in Algorithm 2 which grow only by a multiplicative factor of n/k, since part of the simulated communication consists of messages sent between virtual nodes that are simulated by the same actual node, and as such do not require actual communication. We ask whether this complexity can be improved for $k << n$.

References

1 Rasmus Resen Amossen and Rasmus Pagh. Faster join-projects and sparse matrix multiplications. In *The 12th International Conference on Database Theory (ICDT)*, pages 121–126, 2009.

2 Ariful Azad, Grey Ballard, Aydin Buluç, James Demmel, Laura Grigori, Oded Schwartz, Sivan Toledo, and Samuel Williams. Exploiting Multiple Levels of Parallelism in Sparse Matrix-Matrix Multiplication. *SIAM J. Scientific Computing*, 38(6), 2016.

3 Grey Ballard, Aydin Buluç, James Demmel, Laura Grigori, Benjamin Lipshitz, Oded Schwartz, and Sivan Toledo. Communication optimal parallel multiplication of sparse random matrices. In *Proceedings of the 25th ACM Symposium on Parallelism in Algorithms and Architectures, (SPAA)*, pages 222–231, 2013.

4 Grey Ballard, Alex Druinsky, Nicholas Knight, and Oded Schwartz. Hypergraph Partitioning for Sparse Matrix-Matrix Multiplication. *TOPC*, 3(3):18:1–18:34, 2016.

5 Aydin Buluç and John R. Gilbert. The Combinatorial BLAS: design, implementation, and applications. *IJHPCA*, 25(4):496–509, 2011.

6 Aydin Buluç and John R. Gilbert. Parallel Sparse Matrix-Matrix Multiplication and Indexing: Implementation and Experiments. *SIAM J. Scientific Computing*, 34(4), 2012.

7 Keren Censor-Hillel, Petteri Kaski, Janne H. Korhonen, Christoph Lenzen, Ami Paz, and Jukka Suomela. Algebraic Methods in the Congested Clique. In *Proceedings of the ACM Symposium on Principles of Distributed Computing (PODC)*, pages 143–152, 2015.

8 Keren Censor-Hillel, Dean Leitersdorf, and Elia Turner. Sparse Matrix Multiplication in the Congested Clique model, 2018. `arXiv:arXiv:1802.04789`.

9 Don Coppersmith and Shmuel Winograd. Matrix Multiplication via Arithmetic Progressions. *J. Symb. Comput.*, 9(3):251–280, 1990.

10 Danny Dolev, Christoph Lenzen, and Shir Peled. "Tri, Tri Again": Finding Triangles and Small Subgraphs in a Distributed Setting - (Extended Abstract). In *Proceedings of the 26th International Symposium on Distributed Computing (DISC)*, pages 195–209, 2012.

11 Andrew Drucker, Fabian Kuhn, and Rotem Oshman. On the power of the congested clique model. In *Proceedings of the ACM Symposium on Principles of Distributed Computing (PODC)*, pages 367–376, 2014.

12 François Le Gall. Faster Algorithms for Rectangular Matrix Multiplication. In *Proceedings of the 53rd Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 514–523, 2012.

13 François Le Gall. Powers of tensors and fast matrix multiplication. In *International Symposium on Symbolic and Algebraic Computation (ISSAC)*, pages 296–303, 2014.

14 François Le Gall. Further Algebraic Algorithms in the Congested Clique Model and Applications to Graph-Theoretic Problems. In *Proceedings of the 30th International Symposium on Distributed Computing (DISC)*, pages 57–70, 2016.

15 Francois Le Gall and Florent Urrutia. Improved Rectangular Matrix Multiplication using Powers of the Coppersmith-Winograd Tensor. In *Proceedings of the Twenty-Ninth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 1029–1046, 2018.

16 Taisuke Izumi and François Le Gall. Triangle Finding and Listing in CONGEST Networks. In *Proceedings of the ACM Symposium on Principles of Distributed Computing (PODC)*, pages 381–389, 2017.

17 Haim Kaplan, Micha Sharir, and Elad Verbin. Colored intersection searching via sparse rectangular matrix multiplication. In *Proceedings of the 22nd ACM Symposium on Computational Geometry (SocG)*, pages 52–60, 2006.

18 Hartmut Klauck, Danupon Nanongkai, Gopal Pandurangan, and Peter Robinson. Distributed Computation of Large-scale Graph Problems. In *Proceedings of the Twenty-Sixth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 391–410, 2015.

19 Penporn Koanantakool, Ariful Azad, Aydin Buluç, Dmitriy Morozov, Sang-Yun Oh, Leonid Oliker, and Katherine A. Yelick. Communication-Avoiding Parallel Sparse-Dense Matrix-Matrix Multiplication. In *Proceedings of the IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 842–853, 2016.

20 Alfio Lazzaro, Joost VandeVondele, Jürg Hutter, and Ole Schütt. Increasing the Efficiency of Sparse Matrix-Matrix Multiplication with a 2.5D Algorithm and One-Sided MPI. *CoRR*, abs/1705.10218, 2017.

21 Christoph Lenzen. Optimal deterministic routing and sorting on the congested clique. In *The ACM Symposium on Principles of Distributed Computing (PODC)*, pages 42–50, 2013.

22 Danupon Nanongkai. Distributed approximation algorithms for weighted shortest paths. In *The 46th ACM Symposium on Theory of Computing (STOC)*, pages 565–573, 2014.

23 Gopal Pandurangan, Peter Robinson, and Michele Scquizzato. On the Distributed Complexity of Large-Scale Graph Computations. In *Proceedings of the 30th on Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 405–414, 2018.

24 Edgar Solomonik, Maciej Besta, Flavio Vella, and Torsten Hoefler. Scaling betweenness centrality using communication-efficient sparse matrix multiplication. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 47:1–47:14, 2017.

25 Volker Strassen. Gaussian elimination is not optimal. *Numerische Mathematik*, 13(4):354–356, 1969.

26 Virginia Vassilevska Williams. Multiplying matrices faster than coppersmith-winograd. In *The 44th ACM Symposium on Theory of Computing (STOC)*, pages 887–898, 2012.

27 Raphael Yuster and Uri Zwick. Fast sparse matrix multiplication. *ACM Trans. Algorithms*, 1(1):2–13, 2005.