Sparse Matrix Multiplication Kernels

Matteo Dusefante

Advisor: Riko Jacob
Submitted: August 2018

IT UNIVERSITY OF COPENHAGEN

Abstract

The fast progress of information technology in the present years led to a substantial growth in terms of the volume of data to be analyzed and the size of the problems needed to be solved. Since the Eighties, the formulation of out-of-memory models of computation was motivated by the intractability of the problems by the prevailing architectures. This trend is expected to grow even further. Therefore, algorithm designers have to provide solutions specifically tailored for big data processing.

The design of algorithms often relies on *kernels*, basic yet essential primitives that may be invoked several times by an algorithm during its execution. It is therefore fundamental to rely on theoretically efficient, highly optimized and high performance kernels.

In the field of linear algebra, the most common primitives concern the manipulation of data vectors, such as matrix-matrix multiplication or matrix-vector multiplication. Matrix multiplication has countless applications, from graph algorithms, machine learning, to computer graphics and image processing. Its prominence is validated by the almost half century old research field that aims at providing optimal algorithms for matrix multiplication and trying to answer a long-standing conjecture.

This dissertation aims at providing efficient kernels for sparse matrix multiplication. The main contributions are the following:

(i) we present new *Monte Carlo* data structures for *sparse output-sensitive matrix multiplication* based on repeated predecessor queries that are used to find specific random linear combination of vectors in sparse matrices,

(ii) we introduce new *Atlantic City* algorithms for *sparse Boolean matrix multiplication* derived from a novel framework that exploits size estimators for sparse matrix products,

(iii) we design new *Monte Carlo* algorithms for *fast sparse matrix multiplication* that combine several existing frameworks to provide improved bounds for sparse matrix multiplication,

(iv) we present new deterministic data structures for *permutation matrices in the Parallel External Memory model* that translate into efficient and experimentally evaluated algorithms for permuting in concurrent out-of-memory models.

The result is a collection of randomized and deterministic algorithms for sparse matrix computations that are meant to provide improvements over the state of the art in time or memory efficiency.

Resumé

Informationsteknologiens hurtige udvikling medfører at stadig større mængder data skal analyseres og større problemer skal løses. Siden 1980'erne har motivationen for maskinmodeller med ekstern hukommelse været baseret på intraktibiliteten af store problemer i tidens fremherskende maskinarkitekturer. Denne udvikling forventes at fortsætte. Det er nødvendigt at algoritmedesignere udvikler løsninger som er skræddersyet til behandlingen af Big Data.

Design af algoritmer bygger ofte på *kerner*. En kerne er en simpel, men essentiel grundalgoritme, som kan kaldes flere gange i løbet af kørslen af en algoritme. Det er derfor vigtigt at algoritmedesignere har adgang til kerner som har teoretiske effektivitetsgarantier og samtidig fungerer godt i praksis.

Indenfor lineær algebra bruges de mest almindelige kerner til manipulation af datavektorer såsom matrix-matrix og matrix-vektor multiplikation. Matrixmultiplikation har utallige anvendelser som strækker sig fra grafalgoritmer og maskinlæring til computergrafik og billedbehandling. Problemets fremtrædende betydning er understøttet af op mod et halvt århundrede af stadig aktiv forskning med mål om at vise optimale algoritmer til matrixmultiplikation.

Målet med denne afhandling er at vise effektive kerner til multiplikation af tynde matricer. Hovedbidragene er som følger:

(i) Vi præsenterer nye *Monte Carlo* datastrukturer til *resultatfølsom tynd matrixmultiplikation* baseret på gentagne prædecessorforespørgsler som anvendes til at finde bestemte tilfældige linearkombinationer af vektorer i tynde matricer,

(ii) Vi introducerer nye *Atlantic City* algoritmer til *multiplikation af tynde Boolske matricer* som udnytter størrelsesestimatorer for tynde matrixprodukter,

(iii) Vi designer nye *Monte Carlo* algoritmer til *hurtig multiplikation af tynde matricer* som kombinerer flere eksisterende tilgange for at vise forbedrede øvre grænser,

(iv) Vi præsenterer nye deterministiske datastrukturer til *permutationsmatricer i den Parallelle Eksterne Hukommelsesmodel* som medfører effektive og eksperimentelt understøttede algoritmer til at udføre permutationer i sideløbende eksterne hukommelsesmodeller.

Resultatet er en samling af randomiserede og deterministiske algoritmer til beregninger på tynde matricer som er mere tid- eller pladseffektive end hvad hidtil har været kendt.

Acknowledgments

First of all I would like to show my gratitude to my supervisor, Professor Riko Jacob, for guiding me through the years of my PhD. I thank him for the time he found to listen and discuss my ideas, for introducing me to a new field of research and for supporting me in the quest of becoming a better and independent researcher.

My gratitude goes also to Professor Micheal T. Goodrich for hosting me at University of California at Irvine in the summer of 2018. Having had the chance to visit the Center for Algorithms and Theory of Computation at UCI was a great honor.

My office mates and colleagues deserve my deepest appreciation: Daniel, Johan, Martin, Thomas, Tobias. My PhD would definitely not have been the same without such amazing colleagues. A special acknowledgment goes to Johan for proofreading part of this dissertation and to Tobias for the translation of the abstract.

The Algorithm group at ITU was a very pleasant and motivating workplace. During my PhD I was privileged to meet outstanding researchers who were capable to make from research seminars to lunch conversations inspiring moments.

A special thanks goes to the "Italians" at ITU: Francesco, Hugo, Marco, Paolo, Rosario. You were able to make my journey an experience worth remembering both inside and outside ITU. A special mention goes to Paolo for the countless "coffee breaks" and enlightening conversations. Grazie.

I would like to personally thank the PhD students at the Center for Algorithms and Theory of Computation of the University of California at Irvine for the pleasant stay.

On a person level, I would like to thank my long time friends Gioel and Mattia. Thank you for all the good memories. A special acknowledgment goes to Mattia to whom I owe my decision of pursuing a PhD abroad and many unforgettable moments.

Last but not least, I would like to thank my parents, Dino and Maria. I owe to you a debt of gratitude for your constant support and neverending guidance throughout my life. Finally, I am deeply grateful to Ilaria for her endless amount of encouragement and support during the years of my PhD.

Contents

Chapter 1

Introduction

Matrix multiplication is a fundamental operation in computer science and mathematics with numerous applications in graph algorithms and combinatorial optimization. Matrix multiplication kernels can be exploited to obtain improved algorithms for finding cycles in graphs, small subgraphs and cliques, shortest paths, for designing improved dynamic reachability algorithms and for solving matching problems.

Given two $n \times n$ matrices A and C, the matrix product AC can be trivially computed with $\mathcal{O}(n^3)$ arithmetic operations. Strassen [Str69], in 1969, provided a recursive algorithm able to multiply two matrices in $\mathcal{O}(n^\omega)$ with $\omega = \log_2 7 = 2.807$ by cleverly exploiting cancellations. Beside this, he contributed to lay the foundations of a new, prolific research path [Pan78, BCRL79, Sch81, Pan81, Rom82, CW82, Str86, CW87, Sto10, Wil12]. The last known result is due to Le Gall [LG14] who holds the current record of $\omega < 2.3728639$. In his paper, Le Gall analyzed the powers of a specific trilinear form introduced by Coppersmith and Winograd combined with convex optimization methods, obtaining improved upper bounds on the asymptotic complexity of matrix multiplication.

The vast literature shows that the problem of matrix multiplication over a ring admits truly subcubic algebraic algorithms. That is, it is possible to multiply $n \times n$ matrices using $\mathcal{O}(n^{3-\delta})$ additions and multiplications, for $\delta > 0$. Nevertheless, over algebraic structures that do not guarantee inverse elements, Strassen-like techniques are not allowed which translate to asymptotically worse bounds. For instance, in the Boolean semiring, the best known upper bound for multiplying $n \times n$ matrices is due to Yu [Yu15] whose combinatorial algorithm uses $\tilde{\mathcal{O}}(n^3 / \log^4 n)$ operations. What is more, it is conjectured that there exist no truly sub-

cubic combinatorial algorithm for Boolean matrix multiplication. Using matrix multiplication over algebraic structures without inverse elements can lead to interesting results. For example, the all pairs bottleneck paths problem on weighted graphs, where it is required to find a maximum capacity path from all pairs of nodes in the graph, can be solved using matrix multiplication over the $(\min, \max)$ semiring [WW10].

In several application, the input matrices are sparse, i.e. the number of nonzero elements is upper bounded by $o(n^2)$ and in several instances it is upper bounded by $\mathcal{O}(n)$. In this regard, sparse matrix products are a central kernel in many algorithms, ranging from machine learning, data mining to graph analysis. Sparse matrix-matrix products have been used for multi-source breadth-first search, Markov clustering, high-dimensional and topological similarity search. In numerical applications, sparse matrix computations appear in the Algebraic Multigrid method for solving sparse system of linear equations, linear systems, eigenvalues and least squares problems. A non exhaustive list of specific applications can be found in [BG12, NMAB18].

As mentioned, matrix multiplication has applications in graph theory. Specifically, consider a directed graph $G(V, E)$ with n vertices and h edges. The adjacency matrix A_G associated with the graph $G(V, E)$ is an $n \times n$ matrix with h nonzero entries. Real-world graphs commonly exhibit sparse structures, as confirmed by the Stanford Large Network Dataset Collection [LK14], see Figure 1.1. For instance, the web graph representing web pages from `berkely.edu` and `stanford.edu` domains collected in 2002 have average degree $\bar{d} = 2|E|/|V| \leq 23$, [LLDM09], where directed edges represent hyperlinks between web pages. Similarly, Facebook and Twitter social network graphs have an average degree $\bar{d} \leq 43$, [LM12]. Graphs with sparser structures can be found in road networks, e.g. the California and the Texas road network are low-dimensional networks with average degree $\bar{d} \leq 3$, [LLDM09].

This dissertation contributes with new kernels for matrix computations. The algorithms and data structures are specifically designed for matrices that exhibit a sparse structure, in the input or in the output. We give particular emphasis to big data processing by designing and analyzing kernels on several computational models. In relation to this, we assume that the size of the input is too large to be stored in main memory and we design our algorithms with a specific focus on out-of-memory and concurrent models.

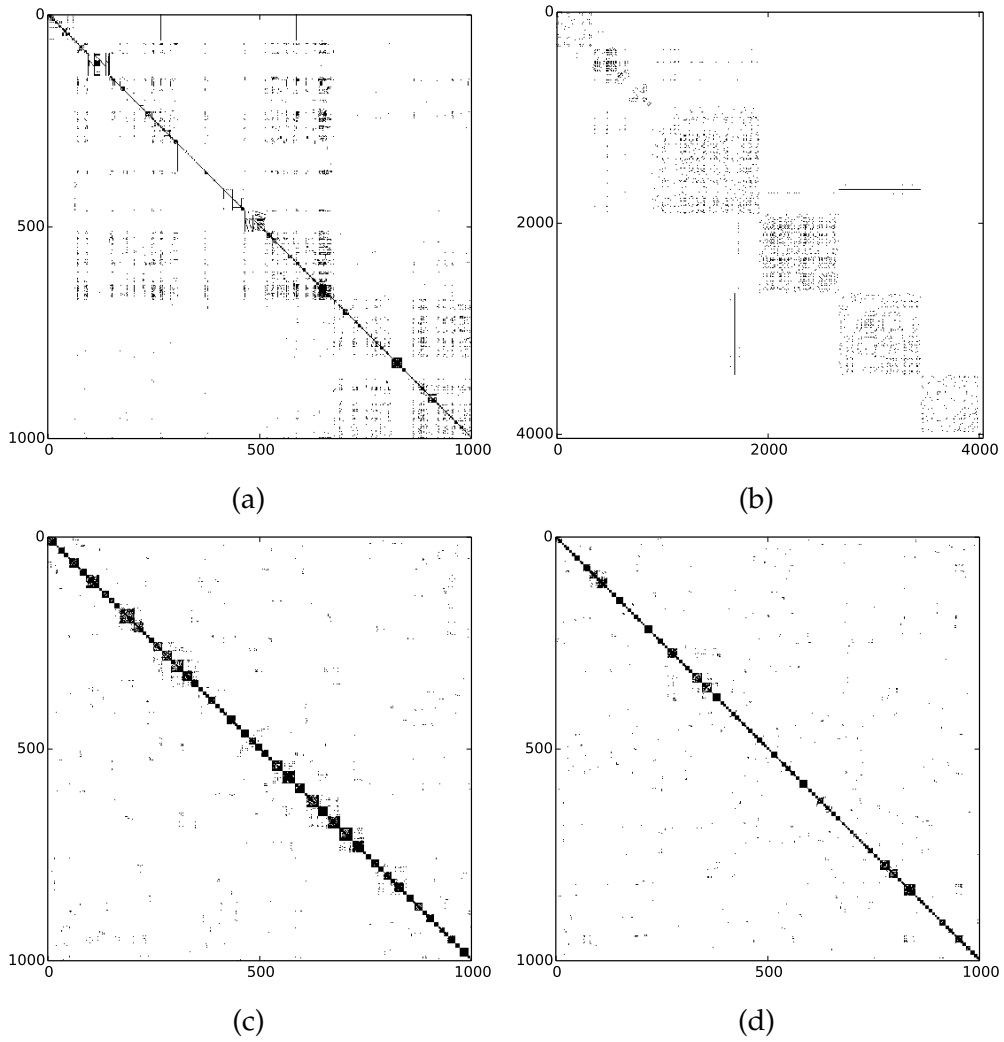


Figure 1.1: Matrix representation of (a) the Berkeley and Stanford domains, (b) Facebook combined egonets, (c) California road network and (d) Texas road network. Matrices (a), (c) and (d) have been collapsed to matrices of size $10^3 \times 10^3$. The datasets are from the Stanford Large Network Dataset Collection [LK14].

1.1 Synopsis

The present dissertation is structured in six chapters. Chapter 1 forms the introduction of the thesis. Chapter 2, Chapter 3, Chapter 4 and Chapter 5 are based on research papers listed in detail in the sequel. Each research chapter is structured to give a short presentation at the

beginning of each chapter followed by a detailed description of the contribution and a thorough comparison with the related work. For each research chapter we identify open questions and future working directions which are summarized in Chapter 6, together with concluding remarks derived from this dissertation.

Chapter 1 The chapter introduces the problem of matrix multiplication. We briefly survey the history of the problem starting from the seminal paper by Strassen [Str69]. We proceed by providing motivation for studying matrix multiplication problems which leads to two fundamental research areas: linear algebra and graph theory. The chapter continues by providing preliminaries for the upcoming research chapters. We analyze more thoroughly the problem of matrix multiplication by introducing algebraic structures and the current hardware design which leads to the review of the models of computation taken into consideration in this dissertation. We conclude the chapter by introducing the probabilistic tools for the analysis of our randomized algorithms and by giving an overview of the lower bounds and conjectures revolving around matrix multiplication.

Chapter 2 The chapter introduces new randomized algorithms for sparse matrix multiplication in the Word RAM model, in the Cache Oblivious model and in the Parallel External Memory model. Specifically, we present Monte Carlo data structures for output-sensitive sparse matrix multiplication that extend the algorithms from [WY14] to the sparse input case. In the Word RAM model, the time bounds of our algorithms match with the state of the art algorithms for sparse matrix multiplication. However, we are able to compute sparse matrix products using linear space, improving the space complexity compared to the state of the art. The chapter is based on the paper *Cache Oblivious Sparse Matrix Multiplication* by Matteo Dusefante and Riko Jacob, appeared on the proceedings of the 2018 Latin American Symposium on Theoretical Informatics [DJ18].

Chapter 3 In this chapter we study the problem of Boolean Matrix multiplication in the Word RAM model and in the Cache Oblivious model and we introduce Atlantic City algorithms for Boolean matrix multiplication. The algorithms in this chapter solve Boolean matrix multiplication by embedding the Boolean semiring into a larger semiring and

by exploiting size estimation algorithm in a novel way. Motivated by the impracticality of fast matrix multiplication à la Coppersmith and Winograd, if we disallow the use of this class of algorithms, then the algorithm presented in this chapter improves over the current state of the art by a polylog factor. The chapter is based on the unpublished manuscript *Atlantic City Boolean Matrix Multiplication* by Matteo Dusefante [Dus18a].

Chapter 4 In this chapter we investigate the problem of Sparse Matrix multiplication in the Word RAM model and in the External Memory model, when fast matrix multiplication is allowed as a subroutine. The result is new partitioning techniques and we show how combining the combinatorial framework of Yuster and Zwick [YZ05] and Amossen and Pagh [AP09] with randomized compression techniques by Jacob and Stöckel [JS15] yields improved bounds for sparse matrix multiplication. The chapter is based on the unpublished manuscript *Compressed Sparse Matrix Multiplication* by Matteo Dusefante [Dus18b].

Chapter 5 This chapter investigates empirically the problem of permuting in the Parallel External Memory model and we present a collection of algorithms, with related C++ implementation, that permute records in the Parallel External Memory model. This chapter serves the purpose of giving motivation for designing algorithms in external memory models. Influenced by the empirical studies of Vuduc [VD03], we study how cache-friendly algorithms operate on modern mutlicore architectures and we achieve significant speed-ups for the permuting phase of our algorithms. This chapter is based on the unpublished manuscript *An Empirical Evaluation of Permuting in Parallel External Memory* by Matteo Dusefante and Riko Jacob [DJ17].

Chapter 6 This chapter aims at summarizing the contributions of this dissertation and to present new open problems and future research directions arose in each chapter. For each research chapter, we highlight the contributions and we guide the reader through each open problem via a detailed discussion.

1.2 Matrix Multiplication in Linear Algebra

The natural applications of matrix multiplication reside in linear algebra. A useful way to understand its algebraic significance is by investigating how matrix multiplication is defined. In fact, the most natural way of defining multiplication between matrices is to consider the elementwise product between entries. The latter, known as Hadamard product, although being used for lossy compression algorithms, does not capture the true significance of matrix multiplication: the composition of linear transformations. It should be mentioned that there are several ways of defining matrix products, e.g. the Hadamard product, the Kronecker product. In this thesis we refer to matrix multiplication as composition of linear transformations.

A transformation is a map between two vector spaces. A linear transformation is a map that preserves the operations of addition and scalar multiplication. Given a linear transformation $T\colon \mathbb{R}^m \to \mathbb{R}^n$, the linearity implies the following properties: $T(\mathbf{u}+\mathbf{v}) = T(\mathbf{u}) + T(\mathbf{v})$ and $T(\alpha\mathbf{u}) = \alpha T(\mathbf{u})$ for $\alpha \in \mathbb{R}$ and $\mathbf{u}, \mathbf{v} \in \mathbb{R}^m$.

Consider a linear transformation $S\colon \mathbb{R}^n \to \mathbb{R}^p$. It maps a n-dimensional vector space to a p-dimensional vector space. The linear transformation S can be represented as a matrix $C \in \mathbb{R}^{p\times n}$ where it holds $S(\mathbf{u}) = C\mathbf{u}$ with $\mathbf{u} \in \mathbb{R}^p$. Consider another linear transformation $T\colon \mathbb{R}^p \to \mathbb{R}^m$. As above, it holds $T(\mathbf{v}) = A\mathbf{v}$ with $\mathbf{v} \in \mathbb{R}^p$ and $A \in \mathbb{R}^{m\times p}$ is the matrix representation of T.

Given the linear transformations S and T, suppose we want to compose them thereby generating the transformation $T \circ S\colon \mathbb{R}^n \to \mathbb{R}^m$ defined as $(T \circ S)(\mathbf{u}) = T(S(\mathbf{u}))$. Note that the the transformation $T \circ S$ is linear since $T(S(\mathbf{u}+\mathbf{v})) = T(S(\mathbf{u}) + S(\mathbf{v})) = T(S(\mathbf{u})) + T(S(\mathbf{v}))$ and $T(S(\alpha\mathbf{u})) = T(\alpha S(\mathbf{u})) = \alpha T(S(\mathbf{u}))$ for the linearity of T and S. The question now becomes, *what is the matrix D associated with the transformation $T \circ S$?* By construction, we know that the transformation $T \circ S$ maps a n-dimensional vector space to a m-dimensional vector space. Hence, the matrix $D \in \mathbb{R}^{m\times n}$. Consider the standard basis for the vector space $\mathbb{R}^n$, i.e. the $n \times n$ identity matrix I_n, where $(I_n)_{*,j}$ is the j-th column vector. Recall that $A \in \mathbb{R}^{m\times p}$ and $C \in \mathbb{R}^{p\times n}$. Define D as n, m-dimensional column vectors as follows,

$$D = \begin{pmatrix} A\left(C(I_n)_{*,1}\right) & A\left(C(I_n)_{*,2}\right) & \cdots & A\left(C(I_n)_{*,n}\right) \end{pmatrix},$$

which is equal to

$$D = \begin{pmatrix} A\left(C_{*,1}\right) & A\left(C_{*,2}\right) & \cdots & A\left(C_{*,n}\right) \end{pmatrix} = AC. \tag{1.1}$$

Formula 1.1 defines matrix multiplication as a composition of linear transformations. It is clear that, the applications of matrix multiplication in linear algebra reside on scenarios where it is necessary to compose transformations, e.g. roto-translation, scaling or shearing. The latter are fundamental operations in the fields of computer graphics and image processing.

1.3 Matrix Multiplication and Graph Theory

The relation between matrices and graphs is straightforward: every graph $G(V, E)$ can be represented through an adjacency matrix A_G, where $(A_G)_{i,j} = 1$ if and only if $(i, j) \in E$ with $i, j \in V$. Furthermore, let $|V| = n$ and $|E| = h$. This affinity makes possible to solve graph related problems using matrix multiplication. The most well known connections between graphs and matrix multiplication revolve around computing the transitive closure of a graph and solving the all pairs shortest path problem.

The transitive closure $G^\star(V, E^\star)$ of a graph $G(V, E)$ is the graph in which $(u, v) \in E^\star$ if and only if there is a path from u to v in G. If A_G is the adjacency matrix of a graph, then $(A_G^k)_{i,j} = 1$ if and only if there is a path of length k from i to j. Furman [Fur70] and Munro [Mun71] showed independently that $A_{G^\star} = I_n \vee A_G \vee A_G^2 \vee \cdots \vee A_G^n$ (where I_n is the $n \times n$ identity matrix) can be computed in $\mathcal{O}(\log n)$ iterations using the method of repeated squaring. The intuition behind the repeated squaring method lies in the observation that the shortest path of at most m edges is the same as the shortest path of at most $n - 1$ edges for all $m > n - 1$. Hence, it is sufficient to compute $(A_G)^{2^{k+1}} = (A_G)^{2^k} \vee (A_G)^{2^k}$ for $k \leq \log n$. This leads to an algorithm for computing the transitive closure in time $\mathcal{O}(n^\omega \log n)$.

The distance version of the all pairs shortest path problem was solved by Seidel [Sei95] who presented an algorithm for undirected, unweighted graphs with n vertices that runs in $\mathcal{O}(n^\omega \log n)$ time together with a randomized algorithm for finding a shortest path between each pair of vertices with the same time bound. Shoshan and Zwick [SZ99] presented a pseudo-polynomial time algorithm for the all pairs shortest path problem for undirected, weighted graphs of n vertices that runs in $\tilde{\mathcal{O}}(Wn^\omega)$, where the edge weights are integers and W is the largest edge weight in the graph. The all pairs shortest paths problem for weighted

and directed graphs was solved by Zwick [Zwi02] who provided an algorithm using $\mathcal{O}(n^{2+\mu})$ time, where $\omega(1,\mu,1) = 1+2\mu$ is the exponent of fast matrix multiplication between matrices of size $n \times n^{\mu}$ and $n^{\mu} \times n$.

The problem of triangle finding and matrix multiplication are intrinsically connected. A triangle in a graph $G(V,E)$ with n vertices is a three node subgraph $\{(u,v),(v,w),(w,u)\}$ such that $u,v,w \in V$ and $\{(u,v),(v,w),(w,u)\} \in E$. That is, the nodes u, v and w are connected to form a triangle. In order to identify a triangle in $G(V,E)$, consider the $n \times n$ adjacency matrix A_G associated with G and compute A_G^2. If $(A_G^2)_{i,j} \neq 0$ then there exists $k \in [n]$ such that $A_{i,k} \neq 0$ and $A_{k,j} \neq 0$. If $A_{i,j} \neq 0$ then there is a triangle in G, namely $\{(i,j),(j,k),(k,i)\} \in E$. It follows that, an algorithm for finding triangles in graphs has complexity at most the one required to square a matrix, i.e. $\mathcal{O}(n^{\omega})$.

The algorithm just presented for triangle finding is algebraic in nature and it achieves subcubic asymptotic complexity by mapping the connectivity matrix into a matrix with entries from a ring. Nevertheless, when considering algebraic structures without subtraction, Strassen-like algorithms are not allowed, as the additive inverse is not guaranteed. In contrast to algebraic algorithms, Boolean matrix multiplication admits combinatorial algorithms. The most famous combinatorial algorithm is the *Four Russians* algorithm by Arlazarov, Dinic, Kronrod, and Faradzhev [ADFK70], which stores precomputed lookup tables of polynomial size, used to answer queries of $\mathcal{O}(\log n)$ bits in constant time. The result is an algorithm to multiply Boolean matrices using $\mathcal{O}(n^3/\log^2 n)$ time, with words of size $\Theta(\log n)$. As mentioned, the last known result belongs to Yu [Yu15] whose algorithm achieves $\tilde{\mathcal{O}}(n^3/\log^4 n)$ time. Furthermore, Williams and Williams [WW10] proved equivalences between Boolean matrix multiplication and detecting triangles in a graph. That is, either both problems have truly subcubic combinatorial algorithms, or none of them do.

1.4 Algebraic structures

The problem of multiply matrices of $m \times p$ and $p \times n$, as in Figure 1.2, can be seen as mn inner products of size p. An alternative approach to matrix multiplication requires the computation of p outer products of size mn, see Figure 1.3. Each of these p outer products generates an intermediate $m \times n$ matrix that contributes to the final output matrix. Regardless of the technique used, matrix multiplication involves

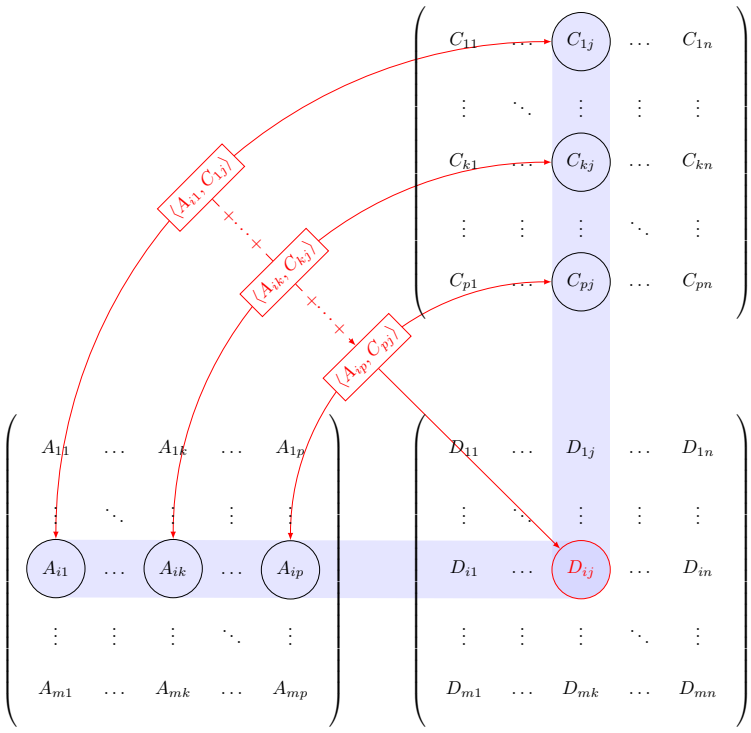


Figure 1.2: Falk's scheme for visualizing the product between a $m \times p$ matrix A with a $p \times n$ matrix C viewed as mn inner products $\langle A_{i,*}, C_{*,j} \rangle$ of size p. Image template courtesy of `texample.net/tikz/examples/matrix-multiplication`.

the product and the sum of entries from row and column vectors. The entry type, i.e. the algebraic structure an entry belongs to, may simplify or complicate the problem. For instance, the inner product of Boolean vectors is nonzero if there exists a nonzero entry in the same coordinate in both vectors. It follows that, Boolean matrix multiplication condenses to detecting at least a witness for each of the mn inner products.

Definition 1.1. *Let $A \in \{0,1\}^{n \times n}$, $C \in \{0,1\}^{n \times n}$ and let $AC \in \{0,1\}^{n \times n}$ be the product matrix. A* witness *for the entry $(AC)_{i,j} = 1$ is an index $\kappa \in [n]$ such that $A_{i,\kappa} = 1$ and $C_{\kappa,j} = 1$.*

Despite its apparent simplicity, we mentioned that there are no truly subcubic algorithms for Boolean matrix multiplication. As surveyed in Chapter 1, matrix multiplication is commonly defined among matrices with elements from a ring. In the following, we review the main algebraic structures where matrix multiplication has been studied starting

with semirings. A semiring $(\mathbf{S}, +, \cdot)$ is a set $\mathbf{S}$ together with two binary operations $+$ and $\cdot$ satisfying the following axioms:

- Additive associativity: for all $a, b, c \in \mathbf{S}$, $(a + b) + c = a + (b + c)$,
- Additive commutativity: for all $a, b \in \mathbf{S}$, $a + b = b + a$,
- Multiplicative associativity: for all $a, b, c \in \mathbf{S}$, $(a \cdot b) \cdot c = a \cdot (b \cdot c)$,
- Left and right distributivity: for all $a, b, c \in \mathbf{S}$, $a \cdot (b + c) = (a \cdot b) + (a \cdot c)$ and $(b + c) \cdot a = (b \cdot a) + (c \cdot a)$.

The absence of an additive inverse in the semiring causes an inner product to be nonzero if the partial inner product (or intermediate result) is nonzero. Examples of semirings include the set of natural numbers under addition and multiplication, the non-negative rational numbers and the non-negative real numbers. Several studies have been conducted on matrix multiplication in semiring models, e.g. [BBF+10, PS14]. Other more peculiar semirings include the $(\max, \min)$-semiring used for the all pairs bottleneck paths problem and the $(\min, +)$-semiring used for the all pairs shortest paths problem.

A ring $(\mathbf{R}, +, \cdot)$ is a semiring satisfying the ring axioms:

- Additive identity: there exists an element $0 \in \mathbf{R}$ such that for all $a \in \mathbf{R}$, $0 + a = a + 0 = a$,
- Additive inverse: for every $a \in \mathbf{R}$ there exists $-a \in \mathbf{R}$ such that $a + (-a) = (-a) + a = 0$.

A ring satisfying the multiplicative associativity is an associative ring. As an example, the set of all algebraic integers forms a ring.

A field $(\mathbf{F}, +, \cdot)$ is a ring satisfying the additional following properties.

- Multiplicative commutativity: for all $a, b \in \mathbf{F}$, $a \cdot b = b \cdot a$ (also referred as commutative ring),
- Multiplicative identity: there exists an element $1 \in \mathbf{F}$ such that for all $a \neq 0 \in \mathbf{F}$, $1 \cdot a = a \cdot 1 = a$ (also referred as ring with identity),
- Multiplicative inverse: for each $a \neq 0 \in \mathbf{F}$, there exists an element $a^{-1} \in \mathbf{F}$ such that for all $a \neq 0 \in \mathbf{F}$, $a \cdot a^{-1} = a^{-1} \cdot a = 1$, where 1 is the identity element.

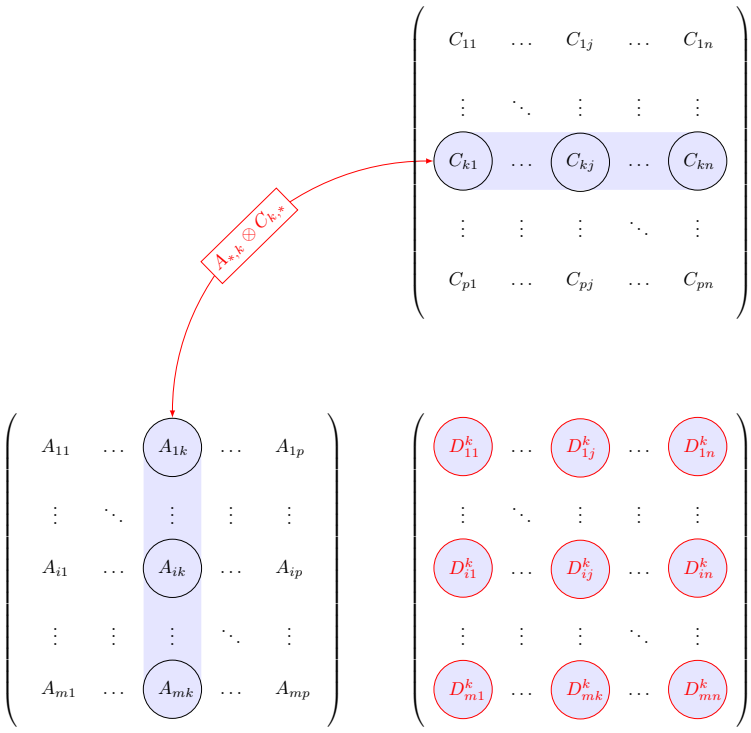


Figure 1.3: Falk's scheme for visualizing the product between a $m \times p$ matrix A with a $p \times n$ matrix C viewed as p outer products $A_{*,k} \otimes C_{k,*}$ of size mn. Each of the p outer products generates a $m \times n$ matrix $D^k = A_{*,k} \otimes C_{k,*}$. The output matrix is computed via the element-wise sum $D = \sum_{k=1}^{p} D^k$. Image template courtesy of `texample.net/tikz/examples/matrix-multiplication`.

Examples of fields are the set of rational numbers, real numbers and the complex numbers. Conversely to semirings, rings and fields do not guarantee nonzero inner products, given nonzero intermediate results. This phenomenon is known as cancellation of terms. A cancellation occurs when $(AC)_{i,j} = \langle A_{i,*}, C_{*,j} \rangle = 0$ while elementary products do not evaluate to zero, i.e. $A_{i,\kappa} \cdot C_{\kappa,j} \neq 0$, for some $\kappa \in [n]$. Equivalently, when the following property holds

$$-A_{i,\ell} \cdot C_{\ell,j} = \sum_{k=1}^{\ell-1} A_{i,k} \cdot C_{k,j} + \sum_{k=\ell+1}^{n} A_{i,k} \cdot C_{k,j},$$

for some $\ell \in [n]$. In general, in semirings, either $a = -A_{i,\ell} \cdot C_{k,\ell} \notin \mathbf{S}$ or $a = \sum_{k=1}^{\ell-1} A_{i,k} \cdot C_{k,j} + \sum_{k=\ell+1}^{n} A_{i,k} \cdot C_{k,j} \notin \mathbf{S}$ since the inverse of an element

$a \in \mathbf{S}$ is not guaranteed to be in $\mathbf{S}$. Hence, cancellations cannot occur. Nevertheless, the following may hold: $\mathtt{nnz}(A_{i,*}) > 0$, $\mathtt{nnz}(C_{*,j}) > 0$ while $(AC)_{i,j} = \langle A_{i,*}, C_{*,j} \rangle = 0$. From a set notation and by considering Boolean matrices, this corresponds to nonempty sets with empty intersection, i.e. disjoint sets. For algebraic structures with modular arithmetics, cancellations may occur whenever $0 \equiv (AC)_{i,j} \pmod{q}$ while $(AC)_{i,j} \neq 0$.

The matrix multiplication kernels presented in this dissertation are designed for matrices with entries from fields. That is, our focus is on algebraic structures that allow cancellation of terms. In general, the choice of fields (rather than more generic rings) is required since multiplicative inverses appear in our probability analysis. For our Boolean matrix multiplication algorithms we embed the Boolean semiring into a sufficiently large semiring to allow the size estimation algorithm to output reliable estimates for the number of nonzero entries in the output matrix.

1.5 Models of Computation

The design and analysis of algorithms requires an abstract model of computation with predefined atomic operations. The Turing machine is one of the first theoretical models of computation designed for the purpose of mathematical calculation. Despite its Turing completeness, a Turing Machine is unsuitable for designing algorithms for modern real-world machines since it does not support random-access memory.

The Random Access Machine, on the other hand, is an computational model similar to a multiple-register machine with indirect addressing closer in design to moder architectures. It can be considered as the standard de-facto model for the design and analysis of algorithms. As Turing machines, the Random Access machine does not capture the concept of locality of memory. In general, locality of memory refers to either temporal locality, i.e. a process access the same data a multiple number of times, or spacial locality, i.e. a process is more likely to access data in a neighborhood of a location of an early memory access. The temporal and spacial locality are exploited through the use of caches and block transfers respectively on current hardwares.

In order to fill the locality-gap of the Random Access Machine it is necessary to design a computational model that support temporal and spacial locality. A motivation for memory-related models, applied to matrix multiplication algorithms, is given by empirical evidences.

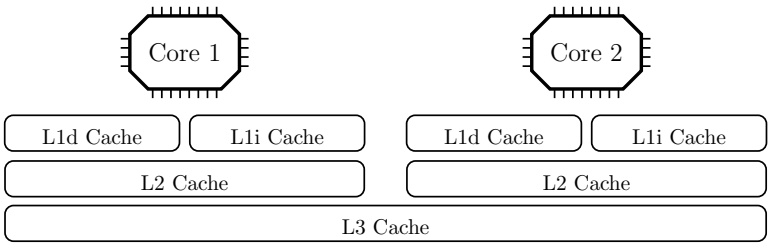


Figure 1.4: Example of a single-socket, dual core architecture. Each core has private L1 data and L1 instruction cache and private L2 cache. The L3 cache is shared between cores and is located above the main memory in the memory hierarchy.

Vuduc [VD03] observed that the CPU utilization is less than 10% when computing elementary products for inner products. This relies on irregular memory access patterns generated by sparse matrices [Gre12].

The structure of a modern architecture is depicted in Figure 1.4. The figure shows a simplification of a single-socket dual core architecture with a three level memory hierarchy. A setup as depicted in Figure 1.4 has to be expected on modern architectures as, among others, Intel® [Lev09]. The purpose of caches is indeed to exploit spacial locality. Every level of the cache hierarchy collects data that has been previously accessed by the processor. If the current requested data is present in one cache layer then we have a so called *cache hit*. On the other hand, if the data is not present in one cache layer then we have a so called *cache miss* and the request for a specific memory address is propagated down the hierarchy to the first layer contacting the address. Meanwhile, the processor remains idle and on hold for the data to be loaded in the caches. It is clear that, to optimize performances, algorithms designers have to exploit data locality in order to reduce cache misses and CPU idleness. In fact, cache hits account for fewer CPU cycles compared to cache misses, especially in the lower levels of the memory hierarchy, where they are responsible for the most expensive operations, causing the CPU to remain idle for up to 40 to 100 cycles for an L3 cache on Intel® Xeon® processors [Lev09]. The motivation for multi-layered memory hierarchies is given by a trade off between cache sizes and performances. The performance of caches degrades descending the hierarchy whereas the size increases. Fast caches are usually of limited size, ranging from 32 KB for L1 to 256 KB for L2. On the performance front, Intel's Haswell

architecture has clean-read latencies that double from L1 cache to L2 cache, namely from 5 ns to 10/15 ns, see Figure 5.6b in Chapter 5.

1.5.1 Word RAM model

The Random Access Machine is the standard de-facto for the design and analysis of algorithms. It supports all the basic data manipulation operations and it allows indirect addressing. Nevertheless, the progress in hardware architectures causes the RAM model to be outdated in certain aspects. The recent advances in CPU hardware makes possible to take advantage of bit-level parallelism through packing various elements in one word and operating on words simultaneously.

The Word RAM model, introduced by Fredman and Willard [FW90], aims at overcoming this issue and specifies an abstract model similar to a Random Access Machine, but with the additional property of performing bitwise operations on a word of w bits, where $w = \Omega(\log n)$ and n is the problem size. Note that it is natural to assume also $w = \mathcal{O}(\log n)$ otherwise it would be possible to pack super-polynomially many elements in a word. Similar to the RAM model, we are interested in bounding the number of word operations performed during the computation, such as logical and algebraic operations, bit shift and comparison operations. It is generally assumed that a memory access takes constant time and the space usage is given by the number of words used during the computation.

For the sake of completeness, we acknowledge that there exists other RAM models, such as the Real RAM model, by Shamos [Sha78], used mostly in computational geometry problems, where computation is carried on exact real numbers. The Word RAM model is widely accepted as the standard for analysis of algorithms as it closely resembles the modern architectures which operate on 64 bits registers and the atomic operations in the Word RAM model follow an imperative, C-like paradigm that can be considered as an abstraction of the assembly language.

1.5.2 The External Memory model

The pioneers in external memory analysis were Hong and Kung [JWK81] who introduced, in 1981, the red-blue pebble game to analyze the I/O complexity of algorithms. This framework was used to derive lower bounds to the n-point Fast Fourier Transform and the $n \times n$ matrix multiplication algorithm using a memory of size $\mathcal{O}(M)$.

In 1988, Aggarwal and Vitter [AV88] introduced the External Memory model. This model enhances Hong and Kung's framework by introducing the concept of spacial locality via block-organized memory. Also known as I/O model, the External Memory model is a two level memory hierarchy with an unbounded external memory and a limited internal memory that can host at most M elements. The computation can take place exclusively in internal memory and, within the memory hierarchy, elements are exchanged using blocks of size B. In this model, we are interested in analyzing the number of I/Os, i.e. memory exchanges, between internal and external memory. In this setting, we denote with M the number of records that can fit into internal memory and B the number of records that can be transferred in a single block.

The implicit constraint between memory and block size is given by the inequality $M \geq B$, since internal memory must host at least a block for the purpose of I/O operations. A common assumption, legitimized by modern architectures, is to consider the size of internal memories expressed as a univariate polynomial on B. Known as *tall cache assumption*, this constraint corresponds to requiring the number of blocks M/B to be larger than the size of each block B. Using asymptotic notation, this becomes $M = \Omega(B^2)$ which can be adjusted to obtain weaker constraints of the form $M = \Omega(B^{1+\varepsilon})$, for some constant $\varepsilon > 0$. Assuming a tall cache assumption may simplify certain problems. In this regard, consider the problem of transposing an $m \times n$ matrix into an $n \times m$ matrix. This problem is equivalent to a layout transformation, namely from row major to column mayor layout (or vice versa). Assuming $M = \Omega(B^2)$, simply load submatrices of size $B \times B$ in internal memory, order the entries accordingly and output the submatrices transposed. Such an algorithm would yield an optimal I/O complexity of $\mathcal{O}(mn/B)$ I/Os. Conversely, without a tall cache assumption a sorting-based algorithm has to be preferred.

1.5.3 The Cache Oblivious Model

The External Memory model is commonly referred as the cache aware model since the parameters M and B are known to the algorithm designer. This allows the design of algorithms that exploit blocking techniques. As a matter of fact, one of the first lower bounds for matrix multiplication on a model of limited memory was provided by Hong and Kung [JWK81] using blocking arguments. The intuition was to

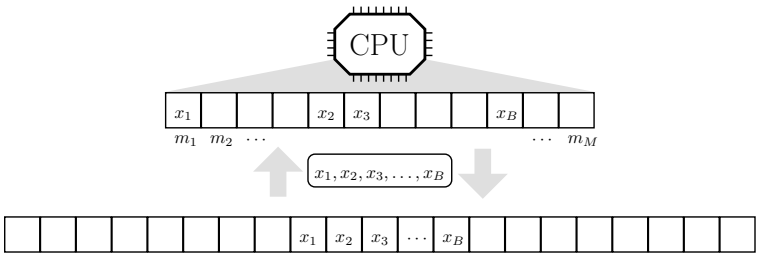


Figure 1.5: The External Memory model with an input/output operation where the B elements $x_1, x_2, x_3, \ldots, x_B$ are loaded in internal memory.

divide the matrix into blocks of size $\sqrt{M} \times \sqrt{M}$ and perform matrix multiplication thereby obtaining a lower bound of $\Omega(n^3/\sqrt{M})$.

The advantages of knowing the parameters M and B reflect on faster algorithms for a specific layer of the memory hierarchy. Conversely, cache aware algorithms do not scale efficiently on multi-level memory hierarchies. In order to optimize programs such that they perform well on all layers of the memory hierarchy, it is necessary to consider a model oblivious to block and memory size. The Cache Oblivious model by Frigo et al. [FLPR99] is a two level memory hierarchy with an internal memory of size M and an unbounded external memory. Similar to the External Memory model, the I/O communication between the levels of the memory is performed through blocks of size B. Nevertheless, in this model, the parameters M and B are oblivious to the algorithm designer and they can only be used to analyze the I/O complexity. The obliviousness causes an algorithm that is optimal in one unknown level of the memory hierarchy to be optimal on all levels simultaneously. Therefore, the Cache Oblivious model makes possible to model a multi-level memory hierarchy via a two-level abstract model.

The Cache oblivious model implicitly assumes an optimal and automatic cache replacement strategy, the presence of only two levels in the memory hierarchy and full cache associativity, i.e. data can be stored in any cache block. The motivation for designing a model with such rigid assumptions is to make the design of algorithms easier. Although these constraints may be considered unrealistic, Frigo et al. showed that, to a certain extent, these assumptions require no loss of generality.

Besides the apparent limitations of oblivious models, Brodal and Fagerberg [BF03] proved an inherent trade off for cache oblivious algorithms between the strength of the tall cache assumption and the

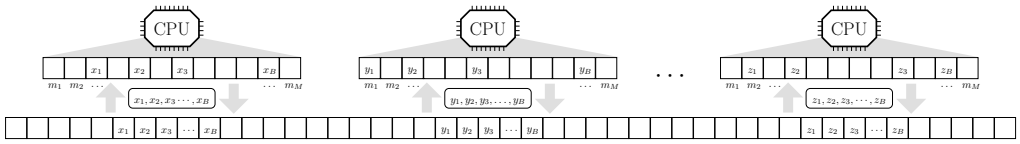


Figure 1.6: The Parallel External Memory model with a parallel input/output operation where each processor loads B elements in internal memory.

overhead when $M \gg B$. In addition, they proved that I/O optimal cache-oblivious comparison-based sorting is not possible without a tall cache assumption and that the problem of permuting is not possible cache-obliviously, not even under the tall cache assumption, see Chapter 5 for further considerations on permuting in the I/O model.

1.5.4 The Parallel External Memory model

The constant progress on hardware design and optimization implies the necessity to design models of computation that closely reflects real hardware settings. The advent of parallel architectures motivated the design of the Parallel Random Access Machine, a parallel-computing counterpart to the Random Access Machine, in order to allow algorithm designers to model the performance of parallel programs.

Similarly, the The Parallel External Memory (PEM) model was introduced by Arge et al. [AGNS08] and it is a cache aware version of the model proposed by Bender et al. [BFGK05]. It is a computational model with P processors and a two-level memory hierarchy which consists of an external memory, shared by all the processors, and P private memories, i.e. caches, exclusive to each processor of limited size M. To perform any operation on the data, a processor must have the data in its cache. The data is transferred between the main memory and the cache in blocks of size B. Multiple processors can access distinct blocks of the external memory concurrently. This means that the external memory is treated as shared memory, and within one parallel I/O, each processor can perform an input or an output. When it comes to accessing the same block of the external memory by multiple processors three different approaches arise: Concurrent Read Concurrent Write (CRCW), Concurrent Read Exclusive Write (CREW) and Exclusive Read Exclusive Write (EREW). Arge et al., as well as we do in this dissertation, consider the Concurrent Read Exclusive Write (CREW) PEM.

1.6 Probability Theory

Several algorithms presented in this thesis are randomized. Randomized algorithms exploit a probabilistic process during their execution. Since randomness is involved, the algorithms exhibit a certain degree of uncertainty which may cause the algorithms to fail by producing an incorrect result. Known in literature as *Monte Carlo*, this class of algorithms use deterministic runtime and provide correct results with a bounded probability. In this regard, it is fundamental to guarantee that the algorithms output correct solutions with a *reasonable* probability. As a consequence, we require our algorithms to succeed with probability polynomially approaching one as the problem size approaches infinity. We refer to this as *with high probability* (whp), which can be expressed as $1 - \mathcal{O}(1/n^c)$ for some constant $c > 0$. Similarly, algorithms may operate using resources, such as number of operations or memory, that are expressed as random variables. This class of algorithms, known as *Las Vegas*, always output a correct result whereas running time or memory consumption are random variables. In this context, we are interested in bounding the resources used in expectation by the algorithms.

In this section, we give an overview of the techniques used in the following chapters for the probabilistic analysis of algorithms. We start with the definition of random variables.

Definition 1.2 (Mitzenmacher and Upfal [MU05, Definition 2.1])**.** *A random variable, X on a sample space Q is a real-valued function on Q; that is, $X\colon Q \to \mathbb{R}$. A discrete random variable X takes on only a finite or countably infinite number of values.*

One of the most common probability distribution is the *discrete uniform distribution* where all the events on the sample space Q are equally likely to be observed. This distribution requires the sample space to be finite since random variables cannot be uniformly distributed among infinite and countable sets. For instance, consider a discrete random variable X which assumes values in a countable infinite set Q. We claim that there is no uniform distribution on Q. That is, there exists no probability function such that $\mathbf{Pr}[X = q] = p$, for all $q \in Q$. By the second probability axiom, i.e. the unit measure axiom, it holds

$$\sum_{q \in Q} p = \sum_{q \in Q} \mathbf{Pr}[X = q] = \mathbf{Pr}[X \in Q] = 1 \tag{1.2}$$

By the first probability axiom, i.e. the probability of an event is a non-negative real number, $\mathbf{Pr}[X = q] = p \geq 0$. Therefore, if $p = 0$ then $\sum_{q \in Q} p = 0$ contradicting (1.2). Similarly, if $p > 0$, then $\sum_{q \in Q} p = \infty$ contradicting (1.2). As a result, whenever we sample uniformly from a set Q we require Q to be finite.

A common procedure in the analysis of randomized algorithms is to estimate the probability of the union of some events. Boole's inequality, also known as union bound, gives an upper bound on the probability of the union of a countable set of events $X_i = q_i$, with $i \in \mathbb{N}$,

$$\mathbf{Pr}\left[\bigcup_i (X_i = q_i)\right] \leq \sum_i \mathbf{Pr}[X_i = q_i].$$

In probability theory, it is often fundamental to study the deviation of a random variable from its expectation $\mathbf{E}[X]$, also known as *tail distribution*. This property, particularly significant to the analysis of Las Vegas algorithms, gave rise to several tail inequalities. One of the simplest tail bounds is the following inequality due to Markov.

Theorem 1.3 (Markov's Inequality [MU05, Theorem 3.1])**.** *Let X be a random variable that assumes only nonnegative values. Then, for all $\delta > 0$,*

$$\mathbf{Pr}[X \geq \delta] \leq \mathbf{E}[X]/\delta.$$

Markov's inequality, in contrast to stronger tail bounds such as Chebyshev or Chernoff-Hoeffding, does not require any type of independence as it is derived solely from the linearity of expectation.

Stronger tail distributions are given by Chernoff bounds which provide exponentially decreasing probabilities on the sum of independent random variables. Hoeffding's inequality provides an upper bound on the probability that the sum of bounded independent random variables deviates from its expected value by more than a certain amount.

Theorem 1.4 (Chernoff-Hoeffding bounds [DP09, Theorem 1.1])**.** *Let $X = \sum_{i \in [k]} X_i$ where the variables X_i, with $i \in [k]$, are independently distributed in $[0, 1]$. Then, for all $\delta > 0$*

$$\mathbf{Pr}[X > \mathbf{E}[X] + \delta] \leq e^{-2\delta^2/k}.$$

If $\delta > 2e\,\mathbf{E}[X]$ then

$$\mathbf{Pr}[X > \delta] \leq 2^{-\delta}.$$

Chernoff-Hoeffding bounds as presented in Theorem 1.4 require independent random variables. The random variables X_i and X_j, with $i \neq j$, are independent if and only if

$$\mathbf{Pr}[(X_i = q_i) \cap (X_j = q_j)] = \mathbf{Pr}[X_i = q_i] \cdot \mathbf{Pr}[X_j = q_j],$$

for all values $q_i, q_j \in Q$. However, Dubhashi and Panconesi [DP09] showed that, for negatively associated random variables, Chernoff-Hoeffding bounds still hold.

Definition 1.5 (Negative association [DP09, Definition 3.1])**.** *The random variables X_i, with $i \in [k]$, are negatively associated if, for all disjoint subsets $I, J \subseteq [k]$ and all nondecreasing functions f and g,*

$$\mathbf{E}[f(X_i)g(X_j)] \leq \mathbf{E}[f(X_i)]\,\mathbf{E}[g(X_j)],$$

with $i \in I$ and $j \in J$.

To prove that a family of random variables is negatively associated [DP09] used two basic properties, namely closure under products and monotone aggregation, that exploit solely monotonicity, symmetry and independence.

Definition 1.6 (Closure under products [DP09])**.** *If $X_1, \ldots, X_n$ and $Y_1, \ldots, Y_m$ are two independent families of random variables that are separately negatively associated, then the family $X_1, \ldots, X_n, Y_1, \ldots, Y_m$ is also negatively associated.*

Definition 1.7 (Monotone aggregation [DP09])**.** *If X_i, $i \in [n]$, are negatively associated and $\mathcal{A}$ is a family of disjoint subsets of $[n]$, then the random variables*

$$f_A(X_i, i \in A), \quad A \in \mathcal{A}$$

are also negatively associated, where the f_A's are arbitrary non decreasing (or non-increasing) functions.

As promised, we present the result from [DP09] which allows to use Chernoff-Hoeffding bounds on negatively associated random variables.

Theorem 1.8 ([DP09, Theorem 3.1])**.** *The Chernoff-Hoeffding bounds can be applied as is to $X = \sum_{i \in [k]} X_i$ if the random variables $X_1, \ldots, X_k$ are negatively associated.*

The canonical example of negatively associated random variables is given by bivariate 0-1 random variables $X_{i,j}$ which take value 1 if the j-th ball is placed into the i-th bin. In order to prove Chernoff-Hoeffding bounds on the sum $X_i = \sum_j X_{i,j}$, one only needs to prove negative association among disjoint subsets of the random variables.

1.7 Lower Bounds and Conjectures

The trivial lower bound for multiplying two $n \times n$ dense matrices is $\Omega(n^2)$. This stems from the size of the input/output matrices, which, in the worts case, will have n^2 entries. It is conjectured that the exponent of matrix multiplication is $\omega = 2 + o(1)$ which, given $n^{2+o(1)} = \text{polylog}(n^2)$, yields an $\tilde{\mathcal{O}}(n^2)$ algorithm. Since Coppersmith and Winograd [CW87], the approach exploited to derived improved bounds was to analyze higher tensor powers of a specific trilinear form. For instance, Le Gall [LG14] analyzed the thirty-second power of the Coppersmith and Winograd's tensor achieving $\omega < 2.3728639$. Similarly, one could think of analyzing higher powers of the Coppersmith and Winograd's tensor hoping to achieve $\omega = 2 + o(1)$. Ambainis, Filmus and Le Gall [AFLG15] proved that analyzing higher tensor powers cannot lead to algorithms running in $\mathcal{O}(n^{2.3725})$ time. In addition, they identify variants of this approach which cannot lead to algorithms with running time $\mathcal{O}(n^{2.3078})$. The results from [AFLG15] imply explicitly that taking higher powers of the Coppersmith and Winograd's tensor cannot lead to an $\mathcal{O}(n^{2+\varepsilon})$ algorithm for matrix multiplication.

Besides defining a specific trilinear form used until recently to derived improved bounds for matrix multiplication, Coppersmith and Winograd showed that the existence of an Abelian group and a subset of it, satisfying certain conditions, imply that their techniques can yield an $\mathcal{O}(n^{2+o(1)})$ time algorithm. Alon, Shpilka and Umans [ASU13] showed that if the sunflower conjecture of Erdős and Rado [ER60] is true then no such group and subset exist. Similarly, Cohn and Umans [CU03], by introducing a new approach for matrix multiplication based on group representation theory, conjectured the existence of combinatorial structures that would yield fast matrix multiplication algorithms. Alon, Shpilka and Umans formulate a variant of the sunflower conjecture and prove that, if true, it contradicts Cohn and Umans' hypothesis.

Despite the subcubic advances of matrix multiplication on a ring, in the Boolean semiring, there is no truly subcubic algorithm know so far.

The current state of the art is given by Yu's combinatorial algorithm. It is believed that combinatorial techniques cannot yield truly subcubic algorithms. Abboud and Williams [AW14] conjectured that, in the Word RAM model with words of $\mathcal{O}(\log n)$ bits, any combinatorial algorithm requires $n^{3-o(1)}$ time in expectation to compute the Boolean product of two $n \times n$ matrices.

Bläser [Blä99] showed that $5n^2/2 - 3n$ multiplications are required for computing the rank of $n \times n$ matrix multiplication. The same result hold for the rank of multiplication in noncommutative division algebras (where all the nonzero elements are invertible) over an arbitrary field. Bläser [Blä01] extended the same result for the multiplicative complexity of $n \times n$ matrix multiplication over arbitrary fields. Landsberg [Lan14] proved that the rank of matrix multiplication is at least $3n^2 - 4n^{3/2} - n$. This result was later improved by Massarenti and Raviolo [MR13] to $3n^2 - 2\sqrt{2}n^{3/2} - 3n$. Raz [Raz02] proved that, for any $c = c(n) \geq 1$, the size of any arithmetic circuit for the product of two matrices, over the real or complex numbers, is $\Omega(n^2 \log_{2c} n)$ unless products with field elements of absolute value larger than c are performed by the circuit.

The lower bounds surveyed so far concern dense matrix products. When the input matrices are sparse and they generate sparse products, the bounds one wants to achieve are of the form $\tilde{\mathcal{O}}(h + k)$, where $h = \mathtt{nnz}(A) + \mathtt{nnz}(C)$ and $k = \mathtt{nnz}(AC)$. This is consistent with the matrix multiplication conjectured since, for dense matrices, it holds $\tilde{\mathcal{O}}(n^2)$.

Consider the problem of determining a pair of vectors $\mathbf{u}$ and $\mathbf{v}$ such that $\langle \mathbf{u}, \mathbf{v} \rangle = 0$, i.e. orthogonal, from two sets A and C each containing n vectors of $d = \omega(\log n)$ dimensions. This problem, known as the *Orthogonal Vectors Problem*, is conjectured not to have a truly subquadratic algorithm, i.e. running in time $n^{2-\epsilon}$. In fact, if such an algorithm exists, then k-CNF-SAT would be solvable in $2^{(1-\epsilon/2)n}$ time, refuting the Strong Exponential Time Hypothesis (SETH) [IPZ01], see [WY14, Lemma A.1]. This problem has been thoroughly studied in communication complexity and it is widely used for proving conditional lower bounds. It is also intrinsically connected with sparse matrix multiplication. In this context, Williams and Yu [WY14] designed algorithms for the communication complexity of sparse matrix multiplication where, using at most $\tilde{\mathcal{O}}(kn)$ randomized communication, they were able to compute the product of matrices with elements over an arbitrary finite field. Moreover, they conjectured that $\Omega(n)$ bits of communication were necessary in the worst case for exchanging the n-bit vectors, for every nonzero entry in

the output. This result was later improved by Van Gucht et al. [VG+15] who proved that $o(kn)$ randomized communication is in fact possible for computing the product of matrices from arbitrary fields.

1.8 Notation

Given matrices $A \in \mathbf{A}^{n \times n}$ and $C \in \mathbf{A}^{n \times n}$ from a generic algebraic structure $\mathbf{A}$, we define h as the number of nonzero entries in the input, i.e. $h = \mathtt{nnz}(A) + \mathtt{nnz}(C)$, k as the number of nonzero entries in the output, i.e. $k = \mathtt{nnz}(AC)$, where $\mathtt{nnz}(A)$ denotes the number of nonzero entries in a matrix A. Let $A_{i,j}$ be the value of the entry in the matrix A with coordinates (i, j). We denote with $A_{*,j}$ and $A_{i,*}$ the j-th column of A and the i-th row of A respectively.

Sparse matrices are stored using a coordinate format, i.e. each entry $A_{i,j} = a$ is stored as a triple $\langle a, i, j \rangle$. Note that we can easily detect, by scanning the input matrices, null vectors. Hence, without loss of generality, we consider only the case where $h \geq 2n$ and the rows of A and the columns of C are not n-dimensional null vectors. Observe that, in contrast cancellations can lead to situations where $k \leq n$. We use the notation $[n]$ to denote the set $\{1, 2 \ldots, n-1, n\}$ and $[m, n]$ to denote the set $\{m, m+1, \ldots, n-1, n\}$.

The definition of Boolean matrix multiplication follows by the standard definition of logical operators,

$$(AC)_{i,j} = \langle A_{i,*}, C_{*,j} \rangle = \bigvee_{\ell=1}^{n} A_{i,\ell} \wedge C_{\ell,j}.$$

The inner product (also known as dot product or scalar product, see Figure 1.2) of a row vector $A_{i,*}$ and a column vector $C_{*,j}$ is defined as

$$(AC)_{i,j} = \langle A_{i,*}, C_{*,j} \rangle = \sum_{\ell=1}^{n} A_{i,\ell} \cdot C_{\ell,j}$$

The outer product (also known as tensor product, see Figure 1.3), of a column vector $A_{*,\ell}$ and a row vector $C_{\ell,*}$ is defined as

$$(AC)^{\ell} = A_{*,\ell} \otimes C_{\ell,*} \text{ where } (AC)^{\ell}_{i,j} = A_{i,\ell} \cdot C_{\ell,j} \text{ and } (AC)_{i,j} = \sum_{\ell=1}^{n} (AC)^{\ell}_{i,j}$$

We denote with *column major layout* the lexicographic order where the entries of A are sorted by column first, and by row index within a column. Analogously, we denote with *row major layout* the order where the entries of A are sorted row-wise first, and column-wise within a row. Observe that a row major layout can be obtained from column major layout by transposing A, denoted with A^T, and vice versa. Furthermore, we use $f(n) = \tilde{\mathcal{O}}(g(n))$ as shorthand for $f(n) = \mathcal{O}(g(n)\log^c g(n))$ for some constant c. The following equivalence holds $\tilde{\mathcal{O}}(g(n)) = \mathcal{O}(g(n)^{1+o(1)})$.

The memory hierarchy we refer to is modeled by the I/O model by Aggarwal and Vitter [AV88], the Ideal Cache Oblivious model by Frigo et al. [FLPR99] and the Parallel External Memory model by Arge et al. [AGNS08]. We denote with M the number of elements that can fit into internal memory, B the number of elements that can be transferred in a single block and P as the number of processors. The parameters M, and B are referred as the *memory size* and *block size* respectively. It holds $1 \leq B \leq M \ll h, n$. We assume that a word is big enough to hold a matrix element as well as its coordinates positions, i.e. a block holds B matrix elements in the coordinate format, and the main memory can store at most M words. Note that, packing a constant number of elements in a word yields only a constant improvement and therefore, implies no loss of generality.

We use the notation $\text{sort}(h)$ as a shorthand for indicating the number of I/Os required to sort h elements. Cache obliviously, it holds $\text{sort}(h) = \mathcal{O}((h/B)\log_M h)$, see [FLPR99]. In the External Memory model it holds $\text{sort}(h) = (h/B)\log_{M/B}(h/B)$ while in the Parallel External Memory model we have $\text{sort}(h) = (h/PB)\log_{M/B}(h/PB)$. Furthermore, we assume that the input is stored contiguously in main memory and, for our parallel algorithms, we assume that the input is partitioned into P segments that are assigned to each processor. Hence, processors are responsible only for the address space spanned by its segment.

Chapter 2

Cache Oblivious Sparse Matrix Multiplication

In this chapter we study the problem of sparse matrix multiplication in the Word RAM model, in the Cache Oblivious model and in the Parallel External Memory model. We present output-sensitive algorithms that exploit randomization in order to compute the product of two sparse matrices with elements over an arbitrary field.

Let $A \in \mathbf{F}^{n \times n}$ and $C \in \mathbf{F}^{n \times n}$ be matrices with h nonzero entries in total from an arbitrary field $\mathbf{F}$. In the Word RAM model, we are able to compute all the k nonzero entries of the product matrix $AC \in \mathbf{F}^{n \times n}$ using $\tilde{\mathcal{O}}(h + kn)$ time and $\mathcal{O}(h)$ space.

In the Cache Oblivious model, we are able to compute cache obliviously all the k nonzero entries of the product matrix $AC \in \mathbf{F}^{n \times n}$ using $\tilde{\mathcal{O}}(h/B + kn/B)$ I/Os and $\mathcal{O}(h)$ space. In the Parallel External Memory model, we are able to compute all the k nonzero entries of the product matrix $AC \in \mathbf{F}^{n \times n}$ using $\tilde{\mathcal{O}}(h/PB + kn/PB)$ time and $\mathcal{O}(h)$ space, which makes the analysis in the Cache Oblivious model a special case of Parallel External Memory for $P = 1$.

The guarantees are given in terms of the size of the field and by bounding the size of $\mathbf{F}$ as $|\mathbf{F}| > kn \log(n^2/k)$ we guarantee an error probability of at most $1/n$ for computing the matrix product.

2.1 Introduction

Williams and Yu [WY14] studied the problem of finding pairs of orthogonal vectors in discrete structures. Inspired by the work of Freivalds [Fre77], they provided a simple probabilistic procedure to test whether there is a pair of non-orthogonal vectors in discrete sub-structure. Their

analysis is centered around dense input matrices. In this chapter we extend their techniques to the sparse input case.

2.2 Contributions

We study the problem of sparse matrix multiplication in the Word RAM model, in the Cache Oblivious model and in the Parallel External Memory model over an arbitrary field $(\mathbf{F}, +, \cdot)$, where $(+, \cdot)$ are atomic operations over the elements of $\mathbf{F}$ in the computational models.

We present a randomized algorithm for multiplying matrices $A \in \mathbf{F}^{n\times n}$, $C \in \mathbf{F}^{n\times n}$ that, after $\mathcal{O}(h)$ time for preprocessing using deterministic $\mathcal{O}(h)$ space, computes, using $\mathcal{O}(nk\log(n^2/k))$ time all the k nonzero entries of the product matrix $AC \in \mathbf{F}^{n\times n}$, with high probability. The algorithm is summarized in the following theorem.

Theorem 2.1 (Word RAM)**.** *Let* $\mathbf{F}$ *be an arbitrary field, let* $A \in \mathbf{F}^{n\times n}$, $C \in \mathbf{F}^{n\times n}$ *and assume* A *and* C *have* h *nonzero entries. After* $\mathcal{O}(h)$ *time for preprocessing and using deterministic* $\mathcal{O}(h)$ *space, it is possible to compute all the* k *nonzero entries of* $AC \in \mathbf{F}^{n\times n}$ *with high probability, using* $\mathcal{O}(kn\log(n^2/k))$ *time.*

Subsequently, we present a cache oblivious algorithm for multiplying matrices $A \in \mathbf{F}^{n\times n}$ and $C \in \mathbf{F}^{n\times n}$. After $\mathcal{O}((h/B)\log_{M/B} h/B)$ I/Os for preprocessing, using deterministic $\mathcal{O}(h)$ space, we are able to compute, using $\mathcal{O}((n/B)k\log(n^2/k))$ I/Os, all the k nonzero entries of the product matrix $AC \in \mathbf{F}^{n\times n}$, with high probability, under a tall cache assumption, i.e. $M \geq B^{1+\varepsilon}$, for some $\varepsilon > 0$.

Theorem 2.2 (Cache Oblivious)**.** *Let* $\mathbf{F}$ *be an arbitrary field, let* $A \in \mathbf{F}^{n\times n}$, $C \in \mathbf{F}^{n\times n}$ *and assume* A *and* C *have* h *nonzero entries. Let* $M \geq B^{1+\varepsilon}$ *for some* $\varepsilon > 0$*. After* $\mathcal{O}((h/B)\log_{M/B}(h/B))$ *I/Os for preprocessing and using deterministic* $\mathcal{O}(h)$ *space, it is possible to compute all the* k *nonzero entries of* $AC \in \mathbf{F}^{n\times n}$ *with high probability, using* $\mathcal{O}((kn/B)\log(n^2/k))$ *I/Os.*

Similarly, from Theorem 2.2, in the Parallel External Memory model, we derive an algorithm for multiplying matrices $A \in \mathbf{F}^{n\times n}$, $C \in \mathbf{F}^{n\times n}$. After $\mathcal{O}((h/PB)\log_d(h/B))$ I/Os for preprocessing, with $d = \max\{2, \min\{M/B, h/PB\}\}$, using deterministic $\mathcal{O}(h)$ space, it computes, using $\mathcal{O}((n/PB + \log P)k\log(n^2/k))$ I/Os, all the k nonzero entries of the product matrix $AC \in \mathbf{F}^{n\times n}$, with high probability, as phrased in the following corollary.

Corollary 2.3 (Parallel External Memory)**.** *Let* $\mathbf{F}$ *be an arbitrary field, let* $A \in \mathbf{F}^{n\times n}$, $C \in \mathbf{F}^{n\times n}$, *assume* A *and* C *have* h *nonzero entries and let* $P \leq n/B$. *After* $\mathcal{O}((h/PB)\log_d(h/B))$ *I/Os for preprocessing, with* $d = \max\{2, \min\{M/B, h/PB\}\}$, *and using deterministic* $\mathcal{O}(h)$ *space, it is possible to compute all the* k *nonzero entries of* $AC \in \mathbf{F}^{n\times n}$ *with high probability, using* $\mathcal{O}((n/PB + \log P)k\log(n^2/k))$ *I/Os.*

Note that, for the Cache Oblivious model and the Parallel External Memory model, preprocessing is dominated by $\mathcal{O}(\mathrm{sort}(h))$ I/Os which stems from layout transposition. Although faster algorithms for transposing sparse matrices exist, for the ease of exposition, we consider $\mathcal{O}(\mathrm{sort}(h))$ I/Os as an upper bound for preprocessing which weakens the bounds only in the parameters of the logarithmic factors. For our parallel algorithm, we consider a cache aware model since concurrency is nontrivial in external memory models whenever the block size B is unknown [BFGK05].

We give rigorous guarantees on the probability of detecting all the nonzero entries of the output matrix by studying how the process of generating random elements from the field affects the process of locating entries. The guarantees are given in terms of the size of the field. If the algorithms generate random variables from an arbitrary field $\mathbf{F}$ then we are able to detect a nonzero entry in the matrix with probability at least $1 - 2/|\mathbf{F}| + 1/|\mathbf{F}|^2$. On the other hand, given an arbitrary field $\mathbf{F}$, if the random variables are generated from $\mathbf{F}^* = \mathbf{F} \setminus \{0\}$ then we detect a nonzero entry with probability at least $1 - 1/|\mathbf{F}^*|$. By bounding the size of $\mathbf{F}$ as $|\mathbf{F}| \geq ckn\log(n^2/k)$, for some constant c, we guarantee an error probability of at most $1/n$. Conversely, if $|\mathbf{F}| < ckn\log(n^2/k)$ we can improve the error probability by repeating the random process an arbitrary number of time, say $\log n$ times, thus sacrificing a $\log n$ factor in space and time with the effect of decreasing the error probability by a factor of n.

2.3 Comparison with the Related Work

Yuster and Zwick [YZ05] presented an algorithm that multiplies two $n \times n$ matrices over a ring using $\tilde{\mathcal{O}}(h^{0.7}n^{1.2} + n^{2+o(1)})$ arithmetic operations. Iwen and Spencer [IS09] proved that if each column of AC contains at most $n^{0.29462}$ nonzero entries, then A and C can be multiplied with $\mathcal{O}(n^{2+\varepsilon})$ operations. Our algorithms improve over Yuster and Zwick

[YZ05] as well as Iwen and Spencer [IS09] when $k < n$ and $h \ll n^2$. In addition, we do not require a balanced assumption of the output matrix, e.g. the number of nonzero entries per column, as in [IS09].

Amossen and Pagh [AP09] provided a sparse, output-sensitive matrix multiplication that incurs in $\tilde{\mathcal{O}}(h^{2/3}k^{2/3} + h^{0.862}k^{0.408})$ operations and $\tilde{\mathcal{O}}(h\sqrt{k}/(BM^{1/8}))$ I/Os. Lingas [Lin09] presented an output-sensitive, randomized algorithm for computing the product of two $n \times n$ Boolean matrices using $\tilde{\mathcal{O}}(n^2k^{\omega/2-1})$ operations. Compared to Amossen and Pagh and Lingas, we exploit cancellations of terms, we do not restrict our analysis to Boolean matrices and we do not make use of fast matrix multiplication, considered by many as impractical [Rob05, VG$^+$15]. In addition, Amossen and Pagh's I/O algorithm is not cache oblivious, i.e. it requires knowledge of the memory size.

Pagh [Pag12] presented a randomized algorithm that computes an unbiased estimator of AC in time $\tilde{\mathcal{O}}(h + nk)$, with guarantees given in terms of the Frobenius norm. Pagh's compressed algorithm achieves the same time bounds as our algorithms. However, we improve over space complexity whenever $k < h/\log n$, i.e when cancellations account for a $1/\log n$ factor compared to the number of input entries. Besides this, Pagh's result is algorithmically more involved, since it makes use of hash functions and Fast Fourier Transform.

Williams and Yu [WY14] provided an output-sensitive, randomized algorithm for matrix multiplication with elements over any field, that, after $\mathcal{O}(n^2)$ preprocessing, computes each nonzero entry of the matrix product in $\tilde{\mathcal{O}}(n)$ time. We extend their techniques to the sparse input case and we improve whenever $h \ll n^2$, i.e. when the input matrices are sparse, both in time and space complexity.

Jacob and Stöckel [JS15] presented a Monte Carlo algorithm that uses $\tilde{\mathcal{O}}(n^2(k/n)^{\omega-2} + h)$ operations and, with high probability, outputs the nonzero elements of the matrix product. In addition, they state an I/O complexity of $\tilde{\mathcal{O}}(n^2(k/n)^{\omega-2}/(M^{\omega/2-1}B) + n^2/B)$. Their analysis is focused specifically on dense matrices and we improve over their results, both in time and I/O complexity, whenever k is asymptotically smaller than n in the general case while we achieve the same bounds when $k = n$. In addition, we do not require knowledge of the memory size as opposed to [JS15].

Van Gucht et al. [VG$^+$15] presented a randomized algorithm for multiplying two Boolean matrices in $\tilde{\mathcal{O}}(k + h\sqrt{k} + h)$ time. In contrast

to their results, our algorithms are not restricted to the Boolean case and we are able to compute the product of matrices from an arbitrary field.

Matrix multiplication has been widely studied in external memory as well. In a restricted setting, i.e. the semiring model, Hong and Kung [JWK81] provided a lower bound of $\Omega(n^3/\sqrt{M})$ I/Os for multiplying two $n \times n$ matrices using n^3 operations and a memory of size M.

Frigo et al. [FLPR99] provided a cache oblivious algorithm for multiplying two $n \times n$ matrices cache obliviously using $\mathcal{O}(n^3)$ operations and $\mathcal{O}(n^2/B + n^3/(B\sqrt{M}))$ I/Os. In the External Memory model, Pagh and Stöckel [PS14] provided a randomized, I/O optimal algorithm for matrix multiplication that incurs in $\tilde{\mathcal{O}}((h/B)\min\{\sqrt{k}/\sqrt{M}, h/M\})$ I/Os. However, their algorithm does not allow cancellation of terms and it requires knowledge of the memory size in order to partition the input matrices. In relation to this, we require no knowledge on the size of M and our algorithm run cache obliviously. To the knowledge of the authors, there are no previously known cache oblivious algorithms for sparse matrix multiplication that exploit cancellations.

2.4 Algorithms

Williams and Yu designed a very simple, output-sensitive algorithm for matrix multiplication, when the output matrices are sparse. Their idea was to design a membership query algorithm to test whether a submatrix in the output contains nonzero entries. This subroutine is then used to guide a two dimensional binary search among rows and columns of AC in search of the k nonzero entries. Their result can be rephrased as follows.

Theorem 2.4 (Williams and Yu [WY14])**.** *Let* $\mathbf{F}$ *be an arbitrary finite field, let* $A \in \mathbf{F}^{n\times n}$ *and let* $C \in \mathbf{F}^{n\times n}$*. After* $\mathcal{O}(n^2)$ *preprocessing time, it is possible to compute all the* k *nonzero entries of* $AC \in \mathbf{F}^{n\times n}$ *with high probability, with only* $\mathcal{O}(kn \log n)$ *time.*

We recall more in detail the proof idea from [WY14]. The intuition behind [WY14] is that nonzero entries in a submatrix of AC with indices in $[i_1, i_2] \times [j_1, j_2]$ and $i_1, i_2, j_1, j_2 \in [n]$, can be detected by extracting sketches

$$a = \sum_{k=i_1}^{i_2} u_k A_{k,*} \qquad c = \sum_{k=j_1}^{j_2} v_k C_{*,k}$$

and testing whether $\langle a, c\rangle = 0$, where u_k and v_k are chosen uniformly at random from $\mathbf{F}$. In general, we can generate $u = (u_1, \ldots, u_n) \in \mathbf{F}^n$ and $v = (v_1, \ldots, v_n) \in \mathbf{F}^n$ uniformly at random, compute all the prefix sums

$$S_i = \sum_{k=1}^{i} u_k A_{k,*} \qquad T_j = \sum_{k=1}^{j} v_k C_{*,k}$$

for $i, j \in [n]$, with $S_0 = T_0 = 0$. For any interval $[i_1, i_2], [j_1, j_2]$ we can detect a nonzero in the $[i_1, i_2] \times [j_1, j_2]$ submatrix of AC by computing $a = S_{i_2} - S_{i_1-1}$, $c = T_{j_2} - T_{j_1-1}$ and subsequently $\langle a, c\rangle$.

When the input matrices are sparse, the prefix sums densify the matrices thus having to compute and store n^2 elements. In addition, sparse matrices make nontrivial to compute linear combinations since row/-column vectors are not explicitly stored.

We refine the analysis of [WY14] as follows. In order to detect the k positions (i, j) such that $(AC)_{i,j} \neq 0$, using binary search among the n^2 feasible locations, we need at most $k \log n^2$ comparisons. We note that the algorithms do not yield false positives when querying submatrices. That is, given an all-zero submatrix with related sketches a and c, it holds $\langle a, c\rangle = 0$ always. This leads to the following lemma.

Lemma 2.5. *Let $\mathbf{F}$ be an arbitrary field and let $A = \{a_1, \ldots, a_n\}$ and $C = \{c_1, \ldots, c_n\}$ be two sets of d-dimensional vectors such that $a_i, c_j \in \mathbf{F}^d$, for all $i, j \in [n]$. In addition, let $u_1, \ldots, u_n, v_1, \ldots, v_n$ be $2n$ random variables chosen uniformly at random from $\mathbf{F}$ and let a, c be vectors defined as*

$$a = \sum_{k=1}^{n} u_k a_k, \qquad c = \sum_{k=1}^{n} v_k c_k.$$

If $\langle a_i, c_j\rangle = 0$, for all $i, j \in [n]$, then $\langle a, c\rangle = 0$.

Proof. The proof follows from the linearity of the inner product,

$$\langle a, c\rangle = \Big\langle \sum_{i=1}^{n} u_i a_i, \sum_{j=1}^{n} v_j c_j \Big\rangle = \sum_{i=1}^{m}\sum_{j=1}^{n} \langle u_i a_i, v_j c_j\rangle = \sum_{i=1}^{m}\sum_{j=1}^{n} u_i v_j \langle a_i, c_j\rangle = 0,$$

where the last equality holds since vectors are pairwise orthogonal by hypothesis. □

As a consequence, at most k queries produce a positive answer, i.e. $\langle a, c\rangle \neq 0$. Via a level-by-level top-down analysis, we note that at most

$\min\{2^i, 2k\}$ nodes are explored at each recursive level, with $i \in [\log n^2]$. Hence, we deduce the following.

$$\sum_{i=1}^{\log n^2} \min\{2^i, 2k\} = \sum_{i=1}^{\log k} 2^i + \sum_{i=\log k}^{\log n^2} k \leq 2k + 2k \log(n^2/k). \tag{2.1}$$

Accordingly, we recursively split AC into two evenly divided submatrices, which resembles the splitting phase of a *k-d* tree. We query each submatrix and after at most $\log(n^2/k)$ queries we isolate each nonzero entry. In the following theorem we show how to compute linear combinations of sparse matrices.

Algorithms overview The intuition for our algorithm is as follows: we preprocess the matrix where we compute sparse vector prefix sums while preserving the sparseness of the input matrices. In order to compute the matrix product, we binary search among the rows and the columns of AC following a partitioning scheme similar to a *k-d* tree. Each submatrix is queried using sketches $a = \sum_{k=1}^{n} u_k a_k$ and $c = \sum_{k=1}^{n} v_k c_k$. For the Word RAM model, Theorem 2.1, we show how to compute sketches efficiently using *fractional cascading* [CG86] and linear time. For the Cache Oblivious model, Theorem 2.2, we show that is possible to compute sketches using $\mathcal{O}(n/B)$ I/Os with an I/O efficient *range coalesced* data structure [DGH15]. The latter is then extended to the Parallel External Memory model, Theorem 2.3.

Theorem 2.1 (Word RAM)**.** *Let* $\mathbf{F}$ *be an arbitrary field, let* $A \in \mathbf{F}^{n\times n}$, $C \in \mathbf{F}^{n\times n}$ *and assume* A *and* C *have* h *nonzero entries. After* $\mathcal{O}(h)$ *time for preprocessing and using deterministic* $\mathcal{O}(h)$ *space, it is possible to compute all the* k *nonzero entries of* $AC \in \mathbf{F}^{n\times n}$ *with high probability, using* $\mathcal{O}(kn \log(n^2/k))$ *time.*

Proof. We assume that the input matrices A and C are stored in column major and row major layout respectively. If not, we can transpose A and C using $\mathcal{O}(h)$ time and $\mathcal{O}(h)$ additional space.

Preprocessing: We generate vectors $u = (u_1, \ldots, u_n) \in \mathbf{F}^n$ and $v = (v_1, \ldots, v_n) \in \mathbf{F}^n$ uniformly at random and we initialize the data structures $\mathcal{A}$ and $\mathcal{C}$ as follows: for each $A_{i,j} \neq 0$ and $C_{i,j} \neq 0$ then

$$\mathcal{A}_{i,j} = \sum_{k=1}^{i} u_k A_{k,j} \quad \mathcal{C}_{i,j} = \sum_{k=1}^{j} v_k C_{i,k} \quad A_{k,j}, C_{i,k} \neq 0,\ i, j \in [n]. \tag{2.2}$$

Intuitively, $\mathcal{A}_{i,j}$ (resp. $\mathcal{C}_{i,j}$) denotes the prefix sum of the nonzero entries of the column vector $A_{*,j}$ up to row i (row vector $C_{i,*}$ up to column j). After this phase, $\mathcal{A}$ and $\mathcal{C}$ maintain the same sparse structure, as well as the same layout, of the original input matrices. Initializing $\mathcal{A}$ and $\mathcal{C}$ requires $\mathcal{O}(h)$ time and $\mathcal{O}(h)$ space.[1]

Starting from column $j = n - 1$, every column vector $\mathcal{A}_{*,j}$ is augmented with every element in even position from the sparse column vector $\mathcal{A}_{*,j+1}$. After the augmentation, the vector $\mathcal{A}_{*,j}$ contains entries native to $\mathcal{A}_{*,j}$ and entries inherited from $\mathcal{A}_{*,j+1}$. For each inherited entry, we add pointers to its native-predecessor and its native-successor. If $\mathcal{A}_{1,j}$ is undefined, every column vector stores a *dummy* entry in first position with value 0. For each entry in $\mathcal{A}_{*,j}$, we add a *bridge* to the entry with the same row index in $\mathcal{A}_{*,j+1}$ or, if it is undefined, we add a bridge to the predecessor. Dummy entries ensure that every element in $\mathcal{A}_{*,j}$ has at least a bridge towards $\mathcal{A}_{*,j+1}$. The augmentation, together with bridging, requires a linear scan of the column vectors. The space required by the augmented vectors is $T(j) = \texttt{nnz}(A_{*,j}) + T(j+1)/2 + 1$, with $T(n) = \texttt{nnz}(A_{*,n})$ and $j \in [n-1]$, which is a geometric series bounded by $2h$. The data structure $\mathcal{A}$ is further augmented with a dense vector $\mathcal{A}_{*,0}$ where every $\mathcal{A}_{i,0}$ has a bridge to either the entry with the same row index or its predecessor in $\mathcal{A}_{*,1}$. The total space required is $2h + n \leq 3h$. Analogous considerations hold for data structure $\mathcal{C}$.

Computing AC**:** We recursively divide AC into two evenly divided submatrices (which resembles the splitting of a *k-d* tree) and query each submatrix in order to detect nonzero entries. Each query is answered via an inner product $\langle a, c \rangle$ where sketches a and c are constructed using fractional cascading. Given a generic submatrix of AC with indices in $[i_1, i_2] \times [j_1, j_2]$ we compute sketches of matrix A with rows in $[i_1, i_2]$ and of matrix C with columns in $[j_1, j_2]$ respectively.

We start by indexing $\mathcal{A}_{i_1,0}$ which redirects to an entry $\mathcal{A}_{i,1}$. We probe the data structure for the native-predecessor, call it $\mathcal{A}_{i_p,1}$, and the native-successor, call it $\mathcal{A}_{i_s,1}$, of $\mathcal{A}_{i,1}$. Recall $i_2 \geq i_1$ and $i \leq i_1$.

1. If $\mathcal{A}_{i,1}$ is native then:

    a) if $i_s < i_1$ then we emit $\mathcal{A}_{i_s,1}$,

[1] Initializing $\mathcal{A}$ and $\mathcal{C}$ corresponds to computing prefix sums of each row and column vector of A and C respectively, which requires a linear scan of the input matrices.

b) if $i = i_1$ then we emit $\mathcal{A}_{i_p,1}$,

c) otherwise we emit $\mathcal{A}_{i,1}$.

2. If $\mathcal{A}_{i,1}$ is inherited then:

a) if $i_s < i_1$ then we emit $\mathcal{A}_{i_s,1}$,

b) otherwise we emit $\mathcal{A}_{i_p,1}$.

Note that, if the predecessor or the successor of $\mathcal{A}_{i,j}$ is not defined in the j-th column vector we simply output 0 or $\mathcal{A}_{i,j}$ respectively. Accordingly, we correct the following lookup by redirecting the search from either the successor $\mathcal{A}_{i_s,1}$, if $i_s < i_1$, or to $\mathcal{A}_{i,1}$, otherwise, and following its bridge to $\mathcal{A}_{i,2}$. We iterate the process up to the n-th column and we produce a n-dimensional vector a_{i_1}. The process for i_2 is analogous. Note that, for i_2, the case (1b) is omitted and inequalities become non-strict as we want to capture the elements with row index i_2. After cascading through the n columns we have vectors a_{i_1} and a_{i_2}.

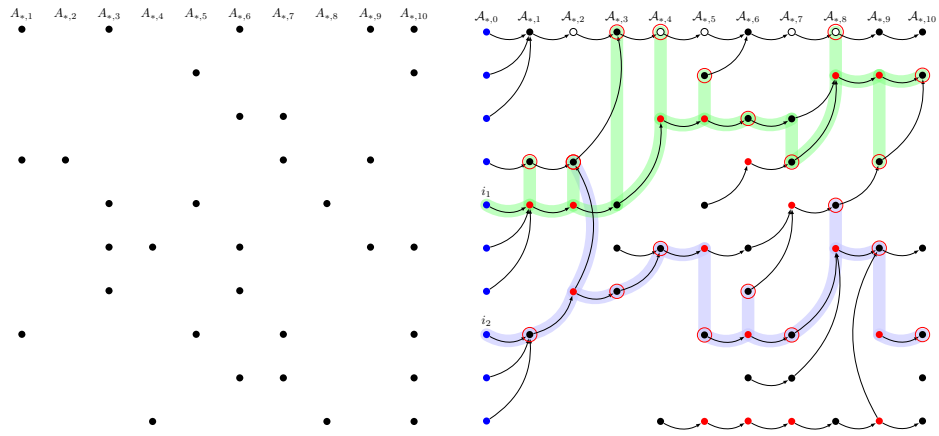
The sketch of the submatrix A with row indices in $[i_1, i_2]$ stems from $a = a_{i_2} - a_{i_1}$, i.e. the element-wise difference. We repeat the same process for $\mathcal{C}$ thus computing c and we query the submatrix of AC by performing the inner product $\langle a, c \rangle$. The construction of sketches a and c requires to probe the data structure a constant number of times per column and per row respectively. Hence, $\mathcal{O}(n)$ time is required per query. By Formula (2.1) at most $k \log(n^2/k)$ queries are required to isolate the k nonzero entries of AC. The claim follows. □

The algorithm from Theorem 2.1 computes k locations (i, j) to as many nonzero entries in $AC \in \mathbf{F}^{n \times n}$. In order to compute $(AC)_{i,j}$ we can retrieve, using Formula (2.2), the entry value as follows

$$(AC)_{i,j} = \langle A_{i,*}, C_{*,j} \rangle = \sum_{\ell=1}^{n} \left[(\mathcal{A}_{i,\ell} - \mathcal{A}_{i-1,\ell})(\mathcal{C}_{\ell,j} - \mathcal{C}_{\ell,j-1}) \right] / u_i v_j$$

while querying unit length matrices.

Figure (2.1a) shows an example of a sparse matrix where $A_{*,j}$ identifies the j-th column. Figure (2.1b) shows the same matrix after preprocessing where the red elements of $\mathcal{A}_{*,j}$ are the entries inherited from column $\mathcal{A}_{*,j+1}$ while the black elements are the original entries of the matrix. The 0-th column, i.e. the vector containing the blue entries, is the dense vector used to index the rows of the matrix. White entries in



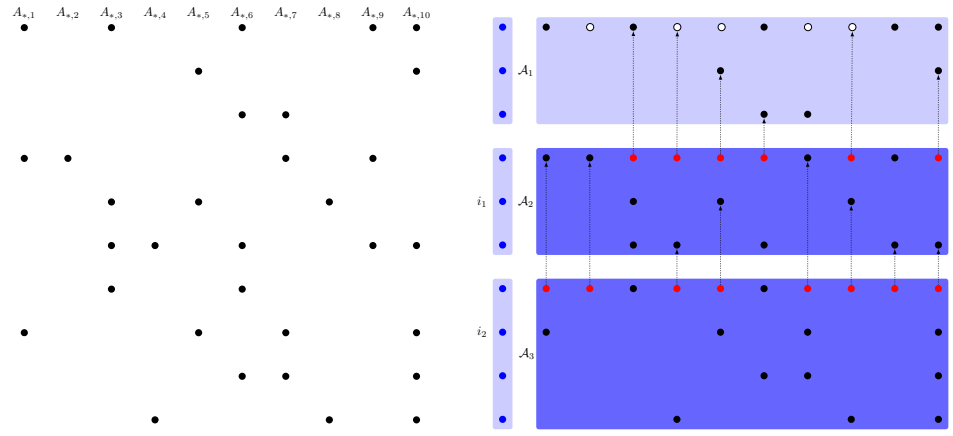
(a) (b)

Figure 2.1: Example of a sparse matrix (a) and the process of generating a sketch a with the data structure of Theorem 2.1, (b).

$\mathcal{A}_{1,*}$ denote dummy entries with value 0. The highlighted paths depict the process of computing a sketch of the submatrix of A with row indexes in $[i_1, i_2]$. Note that, for the ease of exposition, we omit pointers to predecessors and successors between consecutive elements of the same column, as they are implicitly derived from the ordering of the column vectors.

Example 2.6. *Starting from the sparse matrix of Figure 2.1a, we show how the data structure $\mathcal{A}$ is built and how to compute sketches. Given $j = n = 10$, $\mathcal{A}_{*,10} = A_{*,10} = \{A_{1,10}, A_{2,10}, A_{6,10}, A_{8,10}, A_{9,10}, A_{10,10}\}$. For $j = 9$, the augmented vector is as follows $\mathcal{A}_{*,9} = \{A_{1,9}, A_{4,9}, A_{6,9}\} \cup \{A_{2,10}, A_{8,10}, A_{10,10}\}$. That is, $A_{*,9}$ together with the entries from $\mathcal{A}_{*,10}$ in even position. Note that, if $\mathcal{A}_{i,j}$ has to be inherited but $A_{i,j-1} \neq 0$ then we trivially consider $A_{i,j-1}$. This is the case of, e.g., $\mathcal{A}_{10,9}$ and $A_{10,8}$. After the augmentation, predecessors and successors are computed for the inherited entries and bridges are added between column vectors $\mathcal{A}_{*,9}$ and $\mathcal{A}_{*,10}$. The process is iterated down to vector $\mathcal{A}_{*,1}$. Finally, bridges are added between the dense vector $\mathcal{A}_{*,0}$ and $\mathcal{A}_{*,1}$. Storing $\mathcal{A}$ in column major layout allow us to keep the data structure compact and to derive implicitly predecessors/successors for consecutive elements, even between different columns.*

Given indexes, e.g. $i_1 = 5$ and $i_2 = 8$, we proceed to show how to compute a sketch of the submatrix. For i_1, we index $\mathcal{A}_{5,0}$ which redirects to $\mathcal{A}_{5,1} \notin A_{*,1}$. That is, $\mathcal{A}_{5,1}$ is inherited. The predecessor and the successor of $\mathcal{A}_{5,1}$ have row



(a) (b)

Figure 2.2: Example of a sparse matrix (a) and the process of generating a sketch a with a coalesced data structure of Theorem 2.2, (b).

index $i_p = 4$ and $i_s = 8$ respectively. Since $i_s > i_1$ we return $\mathcal{A}_{4,1}$, case 2b from Theorem 2.1. The same considerations apply for $\mathcal{A}_{*,2}$. In $\mathcal{A}_{*,3}$ the lookups redirect to $\mathcal{A}_{5,3} \in A_{*,3}$, i.e. a native entry. Since the row index is equal to i_1 we emit its predecessor, i.e. $\mathcal{A}_{1,3}$, case 1b. In $\mathcal{A}_{*,4}$, the inherited entry $\mathcal{A}_{3,4}$ has a successor $\mathcal{A}_{6,4}$ with row index within $[i_1, i_2]$ but has no predecessor. As a result, we output a dummy entry with value 0. The lookups for the other columns are similar. As a remark, note the path correction between $\mathcal{A}_{3,7}$ and $\mathcal{A}_{4,7}$, case 1a. The construction of the vector a_{i_2} is similar and the sketch follows straightforwardly.

2.4.1 External Memory and Parallel External Memory

Fractional cascading relies on random memory accesses for cascading through $\mathcal{A}_{*,j}$, with $j > 1$. In the worst case, $\mathcal{O}(n)$ blocks must be loaded in memory. Instead, we use a data structure which is close in spirit to the *range coalescing* data structure by Demaine et al. [DGH15].

Theorem 2.2 (Cache Oblivious). *Let $\mathbf{F}$ be an arbitrary field, let $A \in \mathbf{F}^{n \times n}$, $C \in \mathbf{F}^{n \times n}$ and assume A and C have h nonzero entries. Let $M \geq B^{1+\varepsilon}$ for some $\varepsilon > 0$. After $\mathcal{O}((h/B)\log_{M/B}(h/B))$ I/Os for preprocessing and using deterministic $\mathcal{O}(h)$ space, it is possible to compute all the k nonzero entries of $AC \in \mathbf{F}^{n \times n}$ with high probability, using $\mathcal{O}((kn/B)\log(n^2/k))$ I/Os.*

Proof. We describe the procedure for preprocessing matrix A and generating the sketch a. We transpose the input matrix A in column major layout using $\mathcal{O}((h/B)\log_{M/B}(h/B))$ I/Os with a cache oblivious sorting algorithm [BF02] (this requires the tall cache assumption $M \geq B^{1+\varepsilon}$) and we compute column-wise prefix sums using $\mathcal{O}(h/B)$ I/Os.

Given the matrix A, we generate a sparse 0-1 representation A' of A, where $A'_{i,j} = 1$ if and only if $A_{i,j} \neq 0$, $A'_{i,j} = 0$ otherwise, using $\mathcal{O}(h/B)$ I/Os. We compute a counting vector $H = A'\mathbf{1}$, where $\mathbf{1} \in 1^n$ and $H_i = \sum_i \mathtt{nnz}(A_{i,*})$, using a cache oblivious Sparse Matrix Vector Multiplication algorithm [BBF+10] and $\mathcal{O}((h/B)\log_{M/B}(n/M))$ I/Os.

After a prefix sum over H we are able to emit h/n index positions $r_l \in [n]$ such that $\sum_{i=r_l}^{r_{l+1}} \mathtt{nnz}(A_{i,*}) \leq 3n$. As a consequence, we build h/n buckets $\mathcal{A}_l$ of size $\mathcal{O}(n)$ where the elements of $\mathcal{A}_l$ are the entries $A_{i,j}$ such that $i \in [r_l, r_{l+1})$. Starting from $\mathcal{A}_2$, we incrementally augment the bucket $\mathcal{A}_l$ with elements from $\mathcal{A}_{l-1}$ such that, after the augmentation, for every column index j, there is an entry with value equal to the prefix sum up to bucket l. As in Theorem 2.1, we augment the data structure with a column vector $\mathcal{A}_{*,0}$ of size n, where $\mathcal{A}_{i,0}$ indices the l-th bucket if and only if $i \in [r_l, r_{l+1})$, with $l \in [h/n]$.

A query on the data structure $\mathcal{A}$ probes $\mathcal{A}_{i_1,0}$ using a single I/O and it incurs in $\mathcal{O}(n/B)$ I/Os for scanning the bucket, thus generating the sketch a. Analogously, we generate the sketch c and we compute the inner product $\langle a, c\rangle$ by scanning the vectors using $\mathcal{O}(n/B)$ I/Os. □

Corollary 2.3 (Parallel External Memory)**.** *Let* $\mathbf{F}$ *be an arbitrary field, let* $A \in \mathbf{F}^{n\times n}$, $C \in \mathbf{F}^{n\times n}$, *assume* A *and* C *have* h *nonzero entries and let* $P \leq n/B$. *After* $\mathcal{O}((h/PB)\log_d(h/B))$ *I/Os for preprocessing, with* $d = \max\{2, \min\{M/B, h/PB\}\}$, *and using deterministic* $\mathcal{O}(h)$ *space, it is possible to compute all the* k *nonzero entries of* $AC \in \mathbf{F}^{n\times n}$ *with high probability, using* $\mathcal{O}((n/PB + \log P)k\log(n^2/k))$ *I/Os.*

Proof. We describe the procedure for preprocessing matrix A and generating the sketch a. We transpose the input matrix A in Column Major Layout using $\mathcal{O}((h/PB)\log_{M/B}(h/B))$ I/Os with a parallel sorting algorithm [Gre12] and we compute column-wise prefix sums using $\mathcal{O}(h/B + \log P)$ I/Os, where the $\log P$ term stems from a synchronization phase among processors [AGNS08, Ble90].

We generate a sparse 0-1 representation A' of A, where $A'_{i,j} = 1$ if and only if $A_{i,j} \neq 0$, $A'_{i,j} = 0$ otherwise, using $\mathcal{O}(h/PB)$ I/Os. We compute

a counting vector $H = A'\mathbf{1}$, where $\mathbf{1} \in 1^n$ and $H_i = \sum_i \mathtt{nnz}(A_{i,*})$, using a Sparse Matrix Vector Multiplication algorithm in Parallel External Memory [Gre12] and $\mathcal{O}((h/PB)\log_d \min\{n^2/h, n/B\} + \log(h/B))$ I/Os.

After a prefix sum over H, using $\mathcal{O}(n/PB + \log P)$ I/Os, we emit h/n index positions $r_l \in [n]$ such that $\sum_{i=r_l}^{r_{l+1}} \mathtt{nnz}(A_{i,*}) \in [n, 3n]$. Accordingly, we build h/n buckets $\mathcal{A}_l$ of size $\mathcal{O}(n)$ where the elements of $\mathcal{A}_l$ are the entries $A_{i,j}$ such that $i \in [r_l, r_{l+1})$. Starting from $\mathcal{A}_2$, we incrementally augment the bucket $\mathcal{A}_l$ with elements from $\mathcal{A}_{l-1}$ such that, after the augmentation, for every column index j, there is an entry with value equal to the prefix sum up to bucket l. As in Theorem 2.1, we augment the data structure with a column vector $\mathcal{A}_{*,0}$ of size n, where $\mathcal{A}_{i,0}$ indexes the l-th bin if and only if $i \in [r_l, r_{l+1})$, with $l \in [h/n]$.

A query on the data structure $\mathcal{A}$ probes $\mathcal{A}_{i_1,0}$ using a single I/O and it incurs in $\mathcal{O}(n/PB)$ I/Os for scanning the bucket, thus generating sketch a. Analogously, we generate the sketch c and we compute the inner product $\langle a, c\rangle$ by scanning the vectors using $\mathcal{O}(n/PB + \log P)$ I/Os. □

2.5 Probabilistic Analysis

We proceed to give guarantees on the probability of detecting non-zero entries in the output matrix and we study how altering the process of random generation alters the probability of detection.

The guarantees are given in terms of the field size rather on the size of the matrix as, e.g., in [Pag12]. Throughout the analysis we gave no restriction on the field $\mathbf{F}$. Nevertheless, when $\mathbf{F}$ is infinite and countable, we require to sample from a finite subset of $\mathbf{F}$. This constraint is justified since random variables cannot be uniformly distributed among infinite and countable sets, see Section 1.6 for further considerations.

Fields, in contrast with other algebraic structures, guarantee the existence of the multiplicative inverse for elements of $\mathbf{F}$, a property we use for proving the following lemmas, see also Section 1.4.

Lemma 2.7. *Let $A \in \mathbf{F}^{n\times n}$, $C \in \mathbf{F}^{n\times n}$ and let $AC \in \mathbf{F}^{n\times n}$ have at most k nonzero entries. Consider a submatrix of AC with indices $[i_1, i_2] \times [j_1, j_2]$ and assume to query the submatrix with sketches a, c as in Theorem 2.1.*

(i) The matrix has a nonzero entry if and only if $\langle a, c\rangle \neq 0$ with probability at least $1 - 2/|\mathbf{F}| + 1/|\mathbf{F}|^2$.

(ii) The submatrix is all zero if and only if $\langle a, c\rangle = 0$ *with probability at least* $1 - 2k\log(n^2/k)/|\mathbf{F}| + k\log(n^2/k)/|\mathbf{F}|^2$.

Lemma 2.7. (i) If $\langle a, c\rangle \neq 0$, then there exist $i, j \in [n]$ such that $u_i, v_j \neq 0$ and $\langle a_i, c_j\rangle \neq 0$, hence, $(AC)_{i,j} \neq 0$. If there is a nonzero entry then $\langle a, c\rangle \neq 0$ with probability at least $1 - 2/|\mathbf{F}| + 1/|\mathbf{F}|^2$. This is equivalent of saying that if there is a nonzero entry then $\langle a, c\rangle = 0$ with probability at most $2/|\mathbf{F}| - 1/|\mathbf{F}|^2$.

Without loss of generality, let $i_1 = i_2 = i$ and $j_1 = j_2 = j$. Considering a bigger submatrix with exactly one nonzero entry leaves the probability unchanged, while considering more nonzero entries will only increase the probability of $\langle a, c\rangle \neq 0$. Therefore, we consider the case where we want to isolate, with high probability, the location of a single nonzero entry in a submatrix of unit size. It follows that, in order to query the submatrix we have to perform the following inner product $\langle a, c\rangle = \langle u_i a_i, v_j c_j\rangle$, where u, v are chosen uniformly at random from $\mathbf{F}$. Since $\langle a_i, c_j\rangle \neq 0$ by hypothesis, we have that $\mathbf{Pr}(\langle a, c\rangle = 0) \geq 2/|\mathbf{F}| - 1/|\mathbf{F}|^2$.

(ii) If the submatrix of AC with indices $[i_1, i_2] \times [j_1, j_2]$ is all zero then $\langle a, c\rangle = 0$ with probability at least $1 - 2k\log(n^2/k)/|\mathbf{F}| + k\log(n^2/k)/|\mathbf{F}|^2$. By Lemma 2.5 this is true. If $\langle a, c\rangle = 0$ then the submatrix of AC with indices $[i_1, i_2] \times [j_1, j_2]$ is all zero with probability at least $1 - 2k\log(n^2/k)/|\mathbf{F}| + k\log(n^2/k)/|\mathbf{F}|^2$. That is, if $\langle a, c\rangle = 0$ then the submatrix has a nonzero entry with probability at most $2k\log(n^2/k)/|\mathbf{F}| - k\log(n^2/k))/|\mathbf{F}|^2$.

Without loss of generality, let $i_1 = i_2 = i$ and $j_1 = j_2 = j$. We have that $\langle a, c\rangle = \langle ua_i, vc_j\rangle = 0$, where u, v are chosen uniformly at random from $\mathbf{F}$. Therefore, $\mathbf{Pr}(\langle a_i, c_j\rangle \neq 0) \geq 2/|\mathbf{F}| - 1/|\mathbf{F}|^2$. The latter is a lower bound on the probability to not detect a nonzero entry in the output matrix. A union bound over the $k\log(n^2/k)$ queries needed to isolate the k nonzero entries, gives us the probability to incur in at least one false negative. By considering its complement, the claim follows. □

$\mathbf{Pr}(\langle a, c\rangle = \langle u_i a_i, v_j c_j\rangle = 0)$, with $\langle a_i, c_j\rangle \neq 0$, is given by the probability of choosing either u_i or v_j zero uniformly at random from $\mathbf{F}$. By altering the algorithm, such that random entries are now generated from $\mathbf{F}^* = \mathbf{F} \setminus \{0\}$, we derive the following lemma.

Lemma 2.8. *Let* $A \in \mathbf{F}^{n\times n}$, $C \in \mathbf{F}^{n\times n}$ *and let* $AC \in \mathbf{F}^{n\times n}$ *have at most* k *nonzero entries. Let* $\mathbf{F}^* = \mathbf{F} \setminus \{0\}$, *consider the submatrix of* AC *with indices*

$[i_1, i_2] \times [j_1, j_2]$ and assume to query the submatrix with sketches a, c as in Theorem 2.1 where the entries of the vectors u and v are chosen uniformly at random from $\mathbf{F}^*$.

(i) The submatrix has a nonzero entry if and only if $\langle a, c\rangle \neq 0$ with probability at least $1 - 1/|\mathbf{F}^*|$.

(ii) The submatrix is all zero if and only if $\langle a, c\rangle = 0$ with probability at least $1 - k\log((n^2/k) - 1)/|\mathbf{F}^*|$.

Lemma 2.8. (i) If $\langle a, c\rangle \neq 0$, then there exist $i, j \in [m]$ such that $u_i, v_j \neq 0$ and $\langle a_i, c_j\rangle \neq 0$, hence, $(AC)_{i,j} \neq 0$. If there is a nonzero entry then $\langle a, c\rangle \neq 0$ with probability at least $1 - 1/|\mathbf{F}^*|$. This is equivalent of saying that if there is a nonzero entry then $\langle a, c\rangle = 0$ with probability at most $1/|\mathbf{F}^*|$.

If $i_1 = i_2 = i$, $j_1 = j_2 = j$ and $\langle a_i, c_j\rangle \neq 0$ then $\langle a, c\rangle \neq 0$ since scaling vectors with random elements from $\mathbf{F}^*$ preserves non orthogonality. If $i_1 < i_2$ and $j_1 < j_2$ and the submatrix contains exactly one nonzero entry, the same reasoning applies. If the submatrix has $\ell > 1$ nonzero entries, then, without loss of generality, there exist $a_1, \ldots, a_\ell, c_1, \ldots, c_\ell$ such that $\langle u_1 a_1, v_1 c_1\rangle + \cdots + \langle u_\ell a_\ell, v_\ell c_\ell\rangle = 0$ and $\langle a_i, c_j\rangle \neq 0$, for all $i, j \in [\ell]$. That is, ℓ inner products that generate as many nonzero entries and produce a false negative when the submatrix is queried. By the linearity of the inner product, we have that

$$\langle u_1 a_1, v_1 c_1\rangle + \cdots + \langle u_\ell a_\ell, v_\ell c_\ell\rangle = u_1 v_1 \langle a_1, c_1\rangle + \cdots + u_\ell v_\ell \langle a_\ell, c_\ell\rangle.$$

Hence, the sum cancels whenever

$$u_i = -\frac{u_1 v_1 \langle a_1, c_1\rangle + \cdots + u_\ell v_\ell \langle a_\ell, c_\ell\rangle}{v_i \langle a_i, c_i\rangle}$$

for a generic $i \in [\ell]$. Note that, such a u_i is in $\mathbf{F}$ since fields guarantee the existence of additive and multiplicative inverses. The probability to choose u_i such that it cancels the other inner products is the same as choosing an element from $\mathbf{F}^*$ uniformly at random, i.e. $1/|\mathbf{F}^*|$.

(ii) If the submatrix of AC with indices $[i_1, i_2] \times [j_1, j_2]$ is all zero then $\langle a, c\rangle = 0$ with probability at least $1 - k\log((n^2/k) - 1)/|\mathbf{F}^*|$. By Lemma 2.5 this is true. If $\langle a, c\rangle = 0$ then the submatrix of AC with indices $[i_1, i_2] \times [j_1, j_2]$ is all zero with probability at least

$1 - k\log((n^2/k) - 1)/|\mathbf{F}^*|$. That is, if $\langle a, c\rangle = 0$ then the submatrix of AC has a nonzero entry with probability at most $k\log((n^2/k) - 1)/|\mathbf{F}^*|$.

If $\langle a, c\rangle = 0$ and $i_1 = i_2 = i$, $j_1 = j_2 = j$ then $\langle a_i, c_j\rangle = 0$. The same reasoning applies for $i_1 < i_2$, $j_1 < j_2$ and exactly one nonzero entry in the submatrix. Let $\langle a, c\rangle = 0$ and suppose there exist $a_1, \ldots, a_\ell, c_1, \ldots, c_\ell$ such that $\langle u_1 a_1, v_1 c_1\rangle + \cdots + \langle u_\ell a_\ell, v_\ell c_\ell\rangle = 0$ and $\langle a_i, c_j\rangle \neq 0$, for all $i, j \in [\ell]$. Hence, as in (i), the sum cancels with probability $1/|\mathbf{F}^*|$. The latter is a lower bound on the probability to not detect a nonzero entry in the output matrix. A union bound over the $k\log((n^2/k) - 1)$ queries needed to isolate the k nonzero entries, gives us the probability to incur in at least one false negative. We do not consider the last layer, i.e. $\log(n^2/k)$, as it does not involve any stochastic process. By considering its complement, the claim follows. □

Lemma 2.7 and Lemma 2.8 refine the analysis of [WY14] and gives guarantees on the probability that, multiplying two sketches a and c related with submatrices of A, with row indexes in $[i_1, i_2]$, and C, with column indexes in $[j_1, j_2]$, respectively, gives the correct answer, i.e. $\langle a, c\rangle = 0$ if there is no entry in the submatrix, $\langle a, c\rangle \neq 0$ otherwise. In addition, Lemma 2.7 gives guarantees even for small fields, e.g. $\mathbf{F}_2$.

On the other hand, Lemma 2.8 does not apply for $\mathbf{F}_2$ as removing the zero from the field takes randomness out of the process. Nevertheless, removing the zero allow us to deterministically detect entries whenever a submatrix with exactly one nonzero element is considered.

2.6 Conclusions

We present randomized algorithms for multiplying two $n \times n$ matrices with elements over an arbitrary field $\mathbf{F}$. Our analysis covers the Word RAM model, the Cache Oblivious model and the Parallel External Memory model. The algorithms exploit a preprocessing phase in order to compute sketches used to query the output matrix and to find the k nonzero entries of the matrix product.

The algorithms presented require no specific knowledge of the input/output matrices, such as number of nonzero entries per column or balanced distribution of the entries, as, e.g., in [IS09, JS15]. As a matter of fact, we ask whether knowledge on the structure of the input/output matrices may be exploited in order to improve our algorithms.

Algorithm 2.3 computes the matrix product cache obliviously. That is, no knowledge of M and B is required during the computation. Conversely, Pagh and Stöckel [PS14] provided an optimal cache aware algorithm for multiplying sparse matrices that uses blocking arguments and matrix size estimations. Motivated by [PS14], we ask whether such bounds hold even for cache oblivious models. Similarly, in parallel cache-aware models, we formulate the problem whether the same techniques from [PS14] are efficient even in parallel models. That is, we ask whether the following bound $\tilde{\mathcal{O}}((h/PB)\min\{\sqrt{k}/\sqrt{M}, h/M\})$, with matching lower bound, holds in the Parallel External Memory model.

The parallel algorithm from Theorem 2.3 has been designed only in a cache aware model. Given the concurrency problems that arise from concurrent cache oblivious models [BFGK05]. An interesting future direction is to try to extend the bounds from Theorem 2.3 to a concurrent cache oblivious model.

Chapter 3

Atlantic City Boolean Matrix Multiplication

In this chapter, we study the problem of Boolean Matrix multiplication in the Word RAM model and in the Cache Oblivious model. We present novel randomized techniques for computing the product of two Boolean matrices $A \in \{0,1\}^{n \times n}$ and $C \in \{0,1\}^{n \times n}$ with h nonzero entries.

The first algorithm computes all the k nonzero entries of $AC \in \{0,1\}^{n \times n}$ using expected $\tilde{\mathcal{O}}(h + h\sqrt{k} + k)$ time and expected $\tilde{\mathcal{O}}(h/B + h\sqrt{k}/B + k/B)$ I/Os cache obliviously, where the notation $\tilde{\mathcal{O}}(\cdot)$ suppresses polylogarithmic factors, matching the time bound of Van Gucht et al. [VG+15]. Similarly, we propose a second, more refined, randomized algorithm that computes the k nonzero entries of the Boolean product AC using $\mathcal{O}(h + h\sqrt{k} + k)$ expected operations and $\mathcal{O}(\text{sort}(h) + h\sqrt{k}/B + k/B)$ expected I/Os cache obliviously, thus improving over our previous result and over Van Gucht et al. by a $\log^2 n$ factor.

The guarantees for detecting nonzeros are given in terms of the size of the matrix and state an error probability of at least $1 - \mathcal{O}(1/n^{1/4})$ for detecting single entries. For a specif class of output matrices, we guarantee a probability of at least $1 - 1/2^{\sqrt{k}}$ to exceed our time and I/O bounds while computing nonzero in submatrices of AC. In the general case, our bounds are guaranteed with error probability at most $1/\sqrt{k}$.

3.1 Introduction

Randomized algorithms are usually categorized in two classes: Monte Carlo and Las Vegas. The former give correct results with a bounded probability using deterministic runtime. The latter produce always the

correct answer, with a random variable as runtime whose expectation is bounded. In between these categories, there lies a third class: Atlantic City algorithms. These algorithm give correct results with a bounded probability and their runtime is a random variable whose expectation is also bounded. Mollin [Mol02] provides the following definition: "an Atlantic City algorithm is a probabilistic polynomial-time algorithm that answers correctly at least seventy-five percent of the time. In other versions of the algorithm, the value $3/4$ may be replaced by any value greater than $1/2$". As Mollin reports, the term Atlantic City was first introduced by J. Finn in one of his unpublished manuscripts.

The algorithms presented in this chapter compute the product of matrices where single entries are computed with error probability polynomially decreasing on the size of the matrix. Conversely, the expected time and I/O bounds on output submatrices are guaranteed with probability at least polynomially decreasing on the size of the product matrix.

3.2 Contributions

We study the problem of Boolean matrix multiplication in the Word RAM model and in the Cache Oblivious model. We present a randomized algorithm for multiplying matrices $A \in \{0,1\}^{n\times n}$ and $C \in \{0,1\}^{n\times n}$, with $h = \mathtt{nnz}(A) + \mathtt{nnz}(C)$ nonzero entries which can be summarized in the following theorem.

Theorem 3.1. *Let $A, C \in \{0,1\}^{n\times n}$ be Boolean matrices and assume A and C have h nonzero entries. It is possible to compute all the k nonzero entries of $AC \in \{0,1\}^{n\times n}$ using expected $\tilde{\mathcal{O}}(h + h\sqrt{k} + k)$ time and expected $\tilde{\mathcal{O}}(h/B + h\sqrt{k}/B + k/B)$ I/Os cache obliviously.*

The intuition behind Theorem 3.1 is to use size estimation algorithms to compute the positions (i, j) of nonzero entries in highly sparse submatrices of AC. In this framework, it is clear that, the more efficient the estimation algorithm is, the more efficient the overall solution will be. As a matter of fact, the polylogarithmic factors in Theorem 3.1 stem from estimating the number of nonzero entries in the matrix product. By considering an improved algorithm for size estimation, we derive our second result presented in the following theorem.

Theorem 3.2. *Let $A, C \in \{0,1\}^{n\times n}$ be Boolean matrices and assume A and C have h nonzero entries. It is possible to compute all the k nonzero entries of $AC \in \{0,1\}^{n\times n}$ using expected $\mathcal{O}(h + h\sqrt{k} + k)$ time and expected $\mathcal{O}(\text{sort}(h) + h\sqrt{k}/B + k/B)$ I/Os cache obliviously.*

Theorem 3.1 and Theorem 3.2 share the same technique. However, the primitive for the estimation step is now replaced with an extension of Amossen et al. [ACP10] size estimator, where a $1 \pm \varepsilon$ approximation of k is computed using expected linear time in terms of the number of nonzeros in the input matrices.

The guarantees of detecting nonzero entries are given in terms of the size n of the input matrix. The algorithm from Theorem 3.1 detects single nonzero entries with probability at least $1 - 1/n$. Conversely, the algorithm from Theorem 3.2 detects single nonzero entries with probability at least $1 - \mathcal{O}(1/n^{1/4})$. Our time and I/O bounds are guaranteed in terms of the size k of the matrix product AC. In the specific case of entries on the same row/column not sharing the same submatrix, by a Chernoff-Hoeffding bound on negatively associated random variables, we prove that the probability of having highly dense submatrices is exponentially decreasing in k, namely $1 - 2^{-\sqrt{k}}$ for $k > 4e^2$. In the general case, we prove polynomially decreasing bounds in k, namely $1 - 1/\sqrt{k}$.

3.3 Comparison with the Related Work

O'Neil and O'Neil [OO73] presented a probabilistic algorithm to compute the Boolean product of two $n \times n$ Boolean matrices using $\mathcal{O}(n^2)$ expected elementary operations. Their solution is based on an inductive argument where they exploit precomputed positions of the nonzeros entries. Nevertheless, their algorithm runs in worst case cn^3 operations while they achieve $\mathcal{O}(n^2)$ for random matrices.

The *Four Russians* algorithm by Arlazarov, Dinic, Kronrod, and Faradzhev [ADFK70] is the most famous combinatorial algorithm for Boolean matrix multiplication. The Four Russians algorithm exploits partitioning techniques and look up tables to compute the product of Boolean matrices in the Word RAM model using $\mathcal{O}(n^3/\log^2 n)$ time, with words of size $\Theta(\log n)$. Bansal and Williams [BW09] presented an improved combinatorial algorithm for Boolean Matrix multiplication that uses $O(n^3(\log\log n)^2/\log^{9/4} n)$ time. Their algorithm combined the Four Russians technique together with exploiting small sub-

structures in the graph to give the aforementioned time algorithm for triangle detection. Chan [Cha15] later improved over Bansal and Williams and presented a divide and conquer algorithm that incurs in $\mathcal{O}(n^3(\log\log n)^3/\log^3 n)$ operations. The last known result, in term of combinatorial algorithms, is due to Yu [Yu15] who presented a new algorithm for triangle finding and Boolean matrix multiplication. Yu's algorithm generalizes the divide and conquer technique of Chan and runs in $\tilde{\mathcal{O}}(n^3/\log^4 n)$ time. Our algorithms are not combinatorial in nature. By exploiting randomization, we improve over combinatorial algorithms and we achieve both input and output sensitivity.

Lingas [Lin09] presented an output-sensitive, randomized algorithm that computes the product of two $n \times n$ Boolean matrices using $\tilde{\mathcal{O}}(n^2k^{\omega/2-1})$ operations. Compared to Lingas, our algorithms are both input and output sensitive and we improve over Lingas' algorithm whenever $hk^{0.314} < n^2$, according to the actual exponent ω of matrix multiplication [LG14]. Amossen and Pagh [AP09] presented an algorithm for computing Boolean matrix products that incurs in $\tilde{\mathcal{O}}(h^{2/3}k^{2/3} + h^{0.862}k^{0.408})$ operations.[1] Compared to our first result, considering $h = k = \mathcal{O}(n)$, our bound is worse than Amossen and Pagh by a factor of $n^{0.167}$. However, as primitives for our algorithm, we make use of size estimators, compared to fast matrix multiplication à la Coppersmith and Winograd of [AP09]. Our second algorithm closes the gap with Amossen and Pagh for sparse input matrices, given $\tilde{\mathcal{O}}(g(n)) = \mathcal{O}(g(n)^{1+o(1)})$, unless n is large. It holds $n^y \leq \log n$ if and only if $y \leq \log\log n/\log n$ and $n^{0.168} \leq \log n$ for $n \leq 10^8$, assuming [AP09] suppresses only a $\log n$ factor in the analysis. For $\log n \log n$ factors, it holds $n^{0.168} \leq \log n \log n$ for $n \leq 10^{22}$. Assuming the latter and ignoring the constants for fast matrix multiplication, we expect to see asymptotic improvements of [AP09] over our solution only for $n > 10^{22}$.

Amossen and Pagh, as well as Lingas, exploit fast matrix multiplication. It is folklore, e.g. see [Rob05, VG$^+$15], that fast matrix multiplication algorithms are empirically impractical. It follows that, although designing algorithms that rely on fast matrix multiplication may lead to asymptotically better bounds, the result may be of no practical use. Conversely, our algorithms exploit simple primitives and they are designed for providing practical advantages for computing matrix products.

Van Gucht et al. [VG$^+$15] presented a randomized algorithm for multiplying two Boolean matrices which runs in $\tilde{\mathcal{O}}(h + h\sqrt{k} + k)$ time.

[1] The bound from [AP09] becomes $\tilde{\mathcal{O}}(h^{2/3}k^{2/3} + h^{0.859}k^{0.406})$ with ω from [LG14].

Our algorithm is very close in spirit to their solution. As a matter of fact, we treat dense subinstances of the problem as in [VG+15], i.e. by just computing matrix vectors products. Conversely, Van Gucht et al. solve sparse sub-instances using algorithmically more involved techniques, including dyadic intervals, Count Sketch matrices and efficient recovery techniques. Conversely, we use simple matrix products derived from estimation algorithms. In terms of complexity analysis, our first algorithm matches the time bounds of Van Gucht et al. while our second, more efficient algorithm, improves over [VG+15] by a $\log^2 n$ factor. In term of I/O bounds, our second algorithm improves over our first solution by a $\log^2 n$ factor in the $h\sqrt{k}$ term.

3.4 Algorithms

The algorithms presented in this chapter are based on size estimation techniques. The idea is to estimate the number of nonzero entries in highly sparse submatrices. In the following, we survey the state-of-the-art estimation algorithms used as subroutines by our algorithms. In addition, we prove that estimation algorithms are false positive free. This property is fundamental since false positives increase our time and I/O complexity. Conversely, we prove that false negatives (also known as silent failure) occur with probability $1/n$.

Pagh and Stöckel [PS14] showed the existence of an algorithm that, using quasilinear time and I/Os, i.e. $\mathcal{O}(\epsilon^{-3}h\,\mathrm{polylog}(n))$, computes a $1 \pm \varepsilon$ approximation $\tilde{k}_1, \ldots, \tilde{k}_n$ of the number of nonzero entries for the columns of $AC \in \mathbf{F}^{n\times n}$ with high probability.

Lemma 3.3 (Pagh and Stockel [PS14, Lemma 1])**.** *Let* $\mathbf{F}$ *be an arbitrary field and let* $A \in \mathbf{F}^{n\times n}$ *and* $C \in \mathbf{F}^{n\times n}$*, with* $h = \mathtt{nnz}(A) + \mathtt{nnz}(C)$*. Furthermore, let* $\delta, \varepsilon \in (0,1]$*. We can compute estimates* $\tilde{k}_1, \ldots, \tilde{k}_n$ *using* $\mathcal{O}(\varepsilon^{-3}h\log(n/\delta)\log n)$ *time and* $\tilde{\mathcal{O}}(\varepsilon^{-3}h/B)$ *I/Os such that with probability at least* $1-\delta$ *it holds that* $(1-\varepsilon)\,\mathtt{nnz}((AC)_{*,j}) \le \tilde{k}_j \le (1+\varepsilon)\,\mathtt{nnz}((AC)_{*,j})$*, for all* $j \in [n]$*.*

Intuitively, such result holds for the rows of AC, since $C^TA^T = (AC)^T$. Therefore, assume $\tilde{k}_1^T, \ldots, \tilde{k}_n^T$ are estimates for $\mathtt{nnz}(AC)_{1,*}, \ldots, \mathtt{nnz}(AC)_{n,*}$. Equivalently, the same guarantees hold, i.e. $(1-\varepsilon)\,\mathtt{nnz}((AC)_{i,*}) \le \tilde{k}_i^T \le (1+\varepsilon)\,\mathtt{nnz}((AC)_{i,*})$, for all $i \in [n]$.

Lemma 3.3 works by creating a F_0-distinguishability sketch $F \in \{0,1\}^{d\times n}$ with $d = \mathcal{O}(\varepsilon^{-2}\log n/\delta)$ and ε constant. The estimates are

then computed via $(FA)C$, i.e. a $2d$ dense-vector sparse-matrix multiplications of dimension $d \times n \times n$. In order to compute estimates with high probability, in Lemma 3.3, we set $\delta = 1/n$. This increases the time complexity by a factor of two.

The following lemma ensures that the estimation algorithm does not yield false positives, i.e. $\texttt{nnz}((AC)_{*,j}) = 0$ while $\tilde{k}_j \neq 0$. Since false positives are deterministically detected, we assume to have $d = 1$ in the following lemma.

Lemma 3.4. *Let $A, C \in \{0,1\}^{n \times n}$ be matrices with pairwise orthogonal vectors, i.e. $AC = 0^{n \times n}$. Assume to compute estimates $\tilde{k}_1, \ldots, \tilde{k}_n$ using the F_0-distinguishability sketch $F \in \{0,1\}^{1 \times n}$ from Lemma 3.3. It holds $\tilde{k}_j = 0$ for all $j \in [n]$.*

Proof. Let $j \in [n]$ be an arbitrary index. By construction, we have that

$$\begin{aligned}\tilde{k}_j &= (F_1 A_{1,1} + \cdots + F_n A_{n,1})C_{1,j} + \cdots + (F_1 A_{1,n} + \cdots + F_n A_{n,n})C_{n,j} \\ &= F_1(A_{1,1}C_{1,j} + \cdots + A_{1,n}C_{n,j}) + \cdots + F_n(A_{n,1}C_{1,j} + \cdots + A_{n,n}C_{n,j}) \\ &= F_1(A_{1,*}C_{*,j}) + \cdots + F_n(A_{n,*}C_{*,j}) = 0\end{aligned}$$

where the last equality follows since $\langle A_{i,*}, C_{*,j} \rangle = 0$ for all $i, j \in [n]$ by hypothesis. □

Conversely, false negatives may occur, i.e. $\texttt{nnz}((AC)_{*,j}) > 0$ while $\tilde{k}_j = 0$. Nevertheless, in the following, we prove that, with an appropriate choice of d, the size estimator is false negative free with high probability.

Lemma 3.5. *Let $A, C \in \{0,1\}^{n \times n}$ be matrices such that there exist at least a nonzero entry in AC. Assume to compute estimates $\tilde{k}_1, \ldots, \tilde{k}_n$ using the F_0-distinguishability sketch $F \in \{0,1\}^{d \times n}$ from Lemma 3.3, with $d = \log n$. Then, there exists $j \in [n]$ such that $\tilde{k}_j \neq 0$ with probability at least $1 - 1/n$.*

Proof. Without loss of generality, assume $(AC)_{i,j} \neq 0$ and ℓ is the witness for $\langle A_{i,*}, C_{*,j} \rangle \neq 0$, i.e. $A_{i,\ell} \cdot C_{\ell,j} \neq 0$. To lower bound $\mathbf{Pr}[\tilde{k}_j \neq 0]$, we study its complement, i.e. $\mathbf{Pr}[\tilde{k}_j \neq 0] = 1 - \mathbf{Pr}[\tilde{k}_j = 0]$. Consider the definition of $\tilde{k}_j$ from the proof of Lemma 3.4.

$$\begin{aligned}\mathbf{Pr}[\tilde{k}_j = 0] &= \mathbf{Pr}[F_1(A_{1,*}C_{*,j}) + \cdots + F_i(A_{i,*}C_{*,j}) + \cdots + F_n(A_{n,*}C_{*,j}) = 0] \\ &\leq \mathbf{Pr}[F_i(A_{i,1}C_{1,j} + \cdots + A_{i,\ell}C_{\ell,j} + \cdots + A_{i,n}C_{n,j}) = 0] \\ &\leq \mathbf{Pr}[F_i(A_{i,\ell}C_{\ell,j}) = 0],\end{aligned}$$

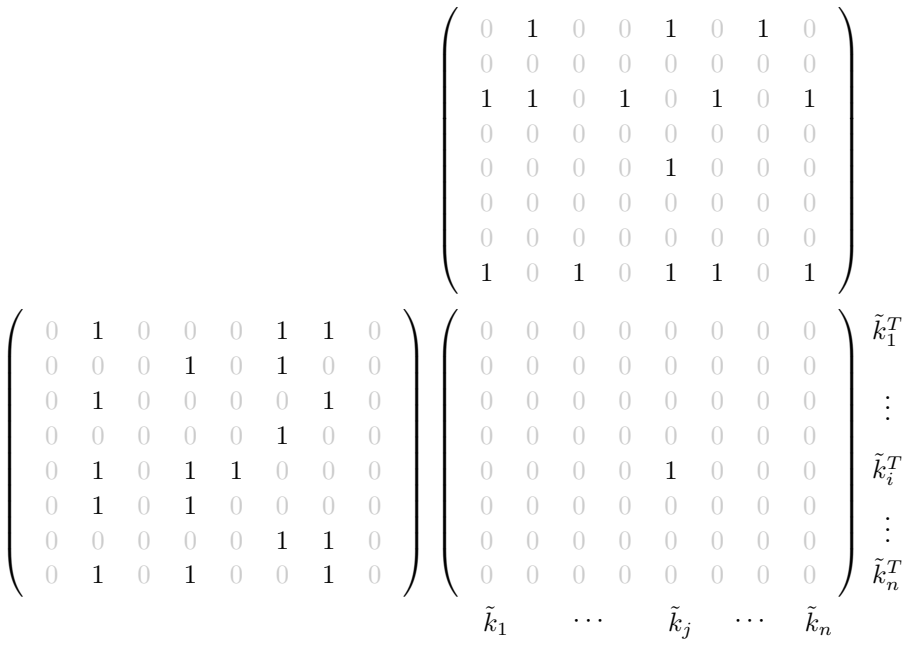


Figure 3.1: Boolean matrices $A, C \in \{0,1\}^{n \times n}$ where the product matrix AC has a single nonzero entry. By Lemma 3.4 and Lemma 3.5, the estimates $\tilde{k}_1, \ldots, \tilde{k}_n$ and $\tilde{k}_1^T, \ldots, \tilde{k}_n^T$ are zero except for $\tilde{k}_j \neq 0$ and $\tilde{k}_i^T \neq 0$, with high probability. Therefore, $(AC)_{i,j} \neq 0$.

where the first inequality holds since, in the worst case, $(AC)_{i,j}$ is the unique nonzero in $(AC)_{*,j}$ while the second inequality holds since, in the worst case, ℓ is the unique witness for $\langle A_{i,*}, C_{*,j} \rangle \neq 0$. The probability is given by choosing F_i as 0 uniformly at random from $\{0,1\}$. With a F_0-distinguishability sketch $F \in \{0,1\}^{d \times n}$ we can increase such probability up to $1 - 1/2^d$. Taking $d = \log n$, it holds $\mathbf{Pr}[\tilde{k}_j \neq 0] = 1 - 1/n$, whenever $(AC)_{i,j} \neq 0$. □

Algorithm overview The intuition for our algorithm is as follows. Consider the matrices $A, C \in \{0,1\}^{n \times n}$ and assume that the product $AC \in \{0,1\}^{n \times n}$ has a single nonzero entry, i.e. $\mathtt{nnz}(AC) = 1$. In order to compute AC it is sufficient to estimate the number of nonzero entries in the columns and in the rows of AC. That is, by applying the algorithm from Lemma 3.3, we have access to estimates $\tilde{k}_1, \ldots, \tilde{k}_n$ and $\tilde{k}_1^T, \ldots, \tilde{k}_n^T$ for the number of nonzeros in the columns and in the rows of AC. It follows that, there exists a unique pair $(i,j) \in n \times n$ such that $\tilde{k}_j \neq 0$

and $\tilde{k}_i^T \neq 0$ with high probability. The uniqueness holds since AC has a single nonzero entry and, by Lemma 3.4, an F_0 sketch on random linear combinations of orthogonal vectors do not yield false positives. Additionally, by Lemma 3.5, (i, j) is reported with probability $1 - 1/n$. Hence, we set $(AC)_{i,j} = 1$, see Figure 3.1. Nevertheless, the constraint introduced so far, i.e. $\texttt{nnz}(AC) = 1$, is rigid and makes the algorithm impractical. However, by exploiting random permutations and matrix splittings we can extend this technique to a more general case.

Lemma 3.6. *Let $AC \in \{0,1\}^{n\times n}$ be the product of two Boolean matrices $A, C \in \{0,1\}^{n\times n}$. Let $k = \texttt{nnz}(AC)$ and assume that every row and every column of AC has at most $\sqrt{k}$ nonzero entries. Assume AC is the matrix resulting from permuting uniformly at random the rows of A and the columns of C with permutation functions π_r and π_c. Divide AC into k submatrices of size $n/\sqrt{k} \times n/\sqrt{k}$. The expected number of nonzero entries in each submatrix is one.*

Proof. Suppose to enumerate the k nonzero entries as $z = 1, \ldots, k$. Consider an arbitrary submatrix $(AC)^\ell$ of size $n/\sqrt{k} \times n/\sqrt{k}$. Define an indicator random variable

$$X_{\ell,z} = \begin{cases} 1 & \text{if the } z\text{-th nonzero entry is in } (AC)^\ell \\ 0 & \text{otherwise} \end{cases} \tag{3.1}$$

We have $X_\ell = \sum_{z=1}^{k} X_{\ell,z} = \texttt{nnz}((AC)^\ell)$. The probability that a nonzero entry is mapped into a submatrix $(AC)^\ell$ is given by the probability that among n possible outcomes the random permutation function for the rows π_r maps a nonzero to the $n/\sqrt{k}$ rows of $(AC)^\ell$ and the random permutation function for the columns π_c maps a nonzero to the $n/\sqrt{k}$ columns of $(AC)^\ell$. That is, if $z = (AC)_{i,j}$ then $\pi_r(i) = i'$ and $\pi_c(j) = j'$, where i' and j' are row and column indices within the submatrix $(AC)^\ell$. It follows,

$$\mathbf{Pr}[\pi_r(i) = i' \wedge \pi_c(j) = j'] = \frac{n/\sqrt{k}}{n} \cdot \frac{n/\sqrt{k}}{n} = \frac{1}{k}. \tag{3.2}$$

The expected number of nonzero entries in the submatrix $(AC)^\ell$ is given by

$$\mathbf{E}[X_\ell] = \mathbf{E}\Big[\sum_{z=1}^{k} X_{\ell,z}\Big] = \sum_{z=1}^{k} \mathbf{E}[X_{\ell,z}] = \sum_{z=1}^{k} \mathbf{Pr}[\pi_r(i) = i' \wedge \pi_c(j) = j'] = 1,$$

which follows from the linearity of expectation. □

Algorithm BMM(A, C)
Input $A, C \in \{0,1\}^{n\times n}$
Output $AC \in \{0,1\}^{n\times n}$

(1) Generate uniformly distributed random permutations π_r and π_c and permute the rows and columns of AC.

(2) Compute estimates for the number of nonzero entries in the rows of AC and for the number of nonzero entries in the columns of AC.

(3) Define $D_c = \{j \in [n] \colon \tilde{k}_j > \sqrt{k}\}$ and $D_r = \{i \in [n] \colon \tilde{k}_i^T > \sqrt{k}\}$. That is, D_c and D_r refers to dense columns and rows of AC respectively, i.e. with more than $\sqrt{k}$ nonzeros. Similarly, define $S_c = [n] \setminus D_c$ and $S_r = [n] \setminus D_r$, i.e. sparse columns and rows of AC.

(4) For each index in D_c and D_r compute directly the matrix vector product.

(5) Filter AC by excluding all the column vectors $j \in D_c$ and all the row vectors $i \in D_r$ from the estimation algorithm.

(6) Divide AC into k submatrices $(AC)^\ell$ of size $n/\sqrt{k} \times n/\sqrt{k}$, with $\ell \in [k]$.

(7) For each submatrix $(AC)^\ell$, compute estimates for the number of nonzero entries in the columns of $(AC)^\ell$, $\tilde{k}_1, \ldots, \tilde{k}_\nu$, and for the number of nonzero entries in the rows of $(AC)^\ell$, $\tilde{k}_1^T, \ldots, \tilde{k}_\nu^T$, with $\nu = n/\sqrt{k}$.

(8) For each $\tilde{k}_j \neq 0$ and $\tilde{k}_i^T \neq 0$, set $(AC)^\ell_{i,j} = 1$, with i, j indices of the sumbatrix $(AC)^\ell$.

Algorithm 3.1: Boolean Matrix Multiplication

Lemma 3.6 allows to use our main technique for an arbitrary number of nonzero entries in the output matrix. Nevertheless, we still require the condition that every row and column of AC has at most $\sqrt{k}$ nonzero entries. With a sparsification trick we achieve full generality.

Theorem 3.1. *Let $A, C \in \{0,1\}^{n\times n}$ be Boolean matrices and assume A and C have h nonzero entries. It is possible to compute all the k nonzero entries of*

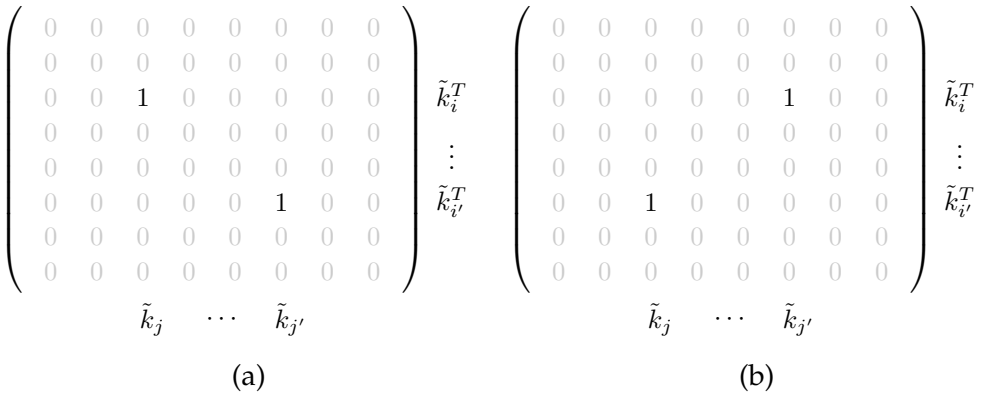


Figure 3.2: Ambiguous estimators of a Boolean matrix $AC \in \{0,1\}^{n\times n}$ where $\tilde{k}_j \neq 0$ and $\tilde{k}_{j'} \neq 0$ and $\tilde{k}_i^T \neq 0$ and $\tilde{k}_{i'}^T \neq 0$. To discern whether (a) $(AC)_{i,j} \neq 0$ and $(AC)_{i',j'} \neq 0$ or (b) $(AC)_{i,j'} \neq 0$ and $(AC)_{i',j} \neq 0$, we generate two estimation subproblems where the rows $A_{i'*}$ and $A_{i',*}$ are mutually ignored by the estimation algorithm. In the worst case, k nonzero entries in a submatrix generate k estimation subproblems.

$AC \in \{0,1\}^{n\times n}$ using expected $\tilde{\mathcal{O}}(h + h\sqrt{k} + k)$ time and expected $\tilde{\mathcal{O}}(h/B + h\sqrt{k}/B + k/B)$ I/Os cache obliviously.

Proof. Steps **(3)**,**(5)** and **(6)** of Algorithm 3.1 require at most a linear scan of the input matrices. Hence, we focus on the remaining steps. Random permutations can be computed in $\mathcal{O}(h)$ time and $\tilde{\mathcal{O}}(h/B)$ I/Os [BBF+10]. Estimates in **(2)** require $\tilde{\mathcal{O}}(h)$ time and $\tilde{\mathcal{O}}(h/B)$ I/Os, by Lemma 3.3.

Consider now the set D_c. It holds $|D_c| \leq \sqrt{k}$. If, by contradiction, $|D_c| > \sqrt{k}$ then there would be more than k entries in AC, since D_c collects the indices of column vectors with more than $\sqrt{k}$ entries. Hence, the step **(4)** requires to compute at most $\sqrt{k}$ matrix vector products. It follows $\tilde{\mathcal{O}}(h\sqrt{k})$ and $\tilde{\mathcal{O}}(h\sqrt{k}/B)$ I/Os. After the filtering step **(5)**, every column and every row of AC has at most $\sqrt{k}$ nonzero entries since all the dense vectors with more than $\sqrt{k}$ nonzeros have already been computed. Therefore, we can apply the estimation technique from Lemma 3.6.

Step **(7)** requires to compute k estimates for k distinct submatrices. In the event that a submatrix has more than one nonzero entry, it is impossible to recover the entries. For instance, consider the case where the algorithm from Lemma 3.3 returns estimates $\tilde{k}_j \neq 0$ and $\tilde{k}_{j'} \neq 0$ and $\tilde{k}_i^T \neq 0$ and $\tilde{k}_{i'}^T \neq 0$. It is impossible to discern whether $(AC)_{i,j} \neq 0$ and

$(AC)_{i',j'} \neq 0$ or $(AC)_{i,j'} \neq 0$ and $(AC)_{i',j} \neq 0$, see Figure 3.2. Hence, we need to run the estimation algorithm for each nonzero in the submatrix $(AC)^{\ell}$. Since we are interested in the expected running time, we repeat the estimation algorithm only $\mathbf{E}[X_\ell]$ times.

Define $\phi(A, C, k)$ as the complexity of step **(7)** with parameters A, C and k. In the following, we use the notation A^I to denotenews $n/\sqrt{k}$ consecutive rows of A and C^J for $n/\sqrt{k}$ consecutive columns of C. The submatrix of AC induced by A^I and C^J is $(AC)^{\ell=I\cap J}$ with related random variable $X_{\ell=I\cap J}$.

$$\begin{aligned}
\mathbf{E}[\phi(A,C,k)] &\leq \sum_{I=1}^{\sqrt{k}} \sum_{J=1}^{\sqrt{k}} \mathbf{E}[X_{\ell=I\cap J}] \Big(\sum_{i\in A^I} \tilde{\mathcal{O}}(\mathtt{nnz}(A_{i,*})) + \sum_{j\in C^J} \tilde{\mathcal{O}}(\mathtt{nnz}(C_{*,j})) \Big) \\
&= \sum_{I=1}^{\sqrt{k}} \Big(\sum_{i\in A^I} \tilde{\mathcal{O}}(\mathtt{nnz}(A_{i,*})) + \tilde{\mathcal{O}}(\mathtt{nnz}(C)) \Big) \qquad (3.3) \\
&= \tilde{\mathcal{O}}(\mathtt{nnz}(A) + \mathtt{nnz}(C)\sqrt{k}),
\end{aligned}$$

where (3.3) holds since $\mathbf{E}[X_{\ell=I\cap J}] = 1$ by Lemma 3.6. Hence, $\tilde{\mathcal{O}}(h\sqrt{k})$ time and $\tilde{\mathcal{O}}(h\sqrt{k}/B)$ I/Os. □

3.4.1 Hashing Based Estimators

Amossen et al. [ACP10] designed an algorithm for estimating the number of nonzero elements in a sparse Boolean matrix product. They formulate their analysis in terms of a join operation between two binary relations R_1 and R_2. However, we can rephrase their main results in terms of Boolean matrix multiplication.

Lemma 3.7 (Amossen et al. [ACP10])**.** *Let $A, C \in \{0,1\}^{n\times n}$, $h = \mathtt{nnz}(A) + \mathtt{nnz}(C)$ and assume $k = \mathtt{nnz}(AC)$. Let ε be a given constant, with $0 < \varepsilon < 1/4$. There are algorithms that run in expected $\mathcal{O}(h)$ time in the RAM model, and expected $\mathcal{O}(\mathrm{sort}(h))$ I/Os in the Cache Oblivious model, that output $\tilde{k}$ such that $\mathbf{Pr}[(1-\varepsilon)k < \tilde{k} < (1+\varepsilon)k] = 1 - \mathcal{O}(1/\sqrt{h})$.*

We provide an intuition of the idea from [ACP10]. For a set U, let $\eta_1, \eta_2 \colon U \to [0,1]$ be hash functions chosen independently at random from a pairwise independent family, and define $\eta \colon U \times U \to [0,1]$ by

$$\eta(x,y) = (\eta_1(x) - \eta_2(y)) \pmod 1. \qquad (3.4)$$

Note that, by taking the $(\mathrm{mod}\,1)$ in Formula 3.4 we have $\eta(x,y) \in [0,1]$. Let $W = \{w \in [n] \mid A_{i,w} \neq 0 \vee C_{w,j} \neq 0\}$. Observe that, the set of witnesses, see Definition 1.1, is a subset of W where both $A_{i,w}$ and $C_{w,j}$ are required to be different than zero.

For each $w \in W$ let $A_w = \{i \in [n] \mid A_{i,w} \neq 0\}$ and $C_w = \{j \in [n] \mid C_{w,j} \neq 0\}$. The intuition behind [ACP10] is to efficiently iterate over all pairs $(x,y) \in A_w \times C_w$, with $w \in W$, for which $\eta(x,y)$ is smaller than a threshold $t = \min\{1/\sqrt{h}, \sqrt{h}/(|A_w||C_w|)\}$, see [ACP10, Lemma 2, Section 2.4]. While finding the relevant pairs, Amossen et al. maintain the x smallest hash values in an unordered buffer in constant amortized time per insertion.[2] By setting x to $\sqrt{h}$, Lemma 3.7 holds. Note that, in terms of x, the algorithm from Lemma 3.7 estimates the number of nonzero entries using expected $\mathcal{O}(x^2)$ time with error probability of $\mathcal{O}(1/x)$.

Lemma 3.7, as is, cannot replace in Theorem 3.1 the estimator algorithm, as it outputs a cumulative estimate of the nonzero entries of the matrix product instead of an estimate vector $\tilde{k}_1, \ldots, \tilde{k}_n$. An extension of Lemma 3.7 allows to compute estimate vectors.

Lemma 3.8. *Let $A, C \in \{0,1\}^{n\times n}$ be two Boolean matrices and let ε be a given constant, with $0 < \varepsilon < 1/4$. There are algorithms that run in expected $\mathcal{O}(h\sqrt{k})$ time in the RAM model, and expected $\mathcal{O}(\mathrm{sort}(h) + h\sqrt{k}/B)$ I/Os in the Cache Oblivious model, that computes $\tilde{k}_j$ such that $\mathbf{Pr}[(1-\varepsilon)\,\mathtt{nnz}((AC)_{*,j})) < \tilde{k}_j < (1+\varepsilon)\,\mathtt{nnz}((AC)_{*,j})] \geq 1 - \mathcal{O}(1/k^{1/4})$.*

Proof. Estimate the number of nonzero entries in AC using the algorithm from Lemma 3.7 and setting x to $\sqrt{h}$. That is, we have access to an estimate k of $\mathtt{nnz}(AC)$ with error probability $\mathcal{O}(1/\sqrt{h})$. This estimate is used to initialize buffers for the following step. Construct the matrix $C' \in \{0,1\}^{n\times n}$ as follows

$$C'_{*,\ell} = \begin{cases} C_{*,j} & \text{if } \ell = j, \\ 0^{n\times 1} & \text{otherwise,} \end{cases} \qquad \text{with } \ell \in [n].$$

Estimate the number of nonzero entries in (AC') with the algorithm from Lemma 3.7, setting x to $h^{1/2}k^{1/4}/n^{1/2}$ where k is the estimate obtained in the preliminary estimation. After $\mathcal{O}(h\sqrt{k}/n)$ time we have access to estimate $\tilde{k}$ with error probability of at most $\mathcal{O}(1/k^{1/4})$. It holds $\tilde{k}_j = \tilde{k}$. By repeating the estimation process for the n columns of C we have the claim. □

[2] Amossen et al. [ACP10] uses k for the number of hash values stored in the buffer and n as the number on nonzero entries in A and C.

Lemma 3.8 gives guarantees in terms of the size of the product matrix, i.e. k. For $k = n$, the overall number of hash values stored is at least $n^{5/4}$, a factor of $n^{3/4}$ more than in Lemma 3.7. Hence, we expect our estimators to be more accurate. Conversely, for highly sparse submatrices, e.g. when k is constant, as in Theorem 3.1, the probability of detecting nonzero entries as in Lemma 3.8 degrades. The intuition is to binary search among the n columns of AC using Lemma 3.7 to answer membership queries.

Lemma 3.9. *Let $A, C \in \{0,1\}^{n\times n}$ be two Boolean matrices and assume $AC \in \{0,1\}^{n\times n}$ has a single nonzero entry. Let ε be a given constant, with $0 < \varepsilon < 1/4$. There are algorithms that run in expected $\mathcal{O}(h)$ time in the RAM model, and expected $\mathcal{O}(\mathrm{sort}(h))$ I/Os in the Cache Oblivious model, that computes AC with probability at least $1 - \mathcal{O}(1/n^{1/4})$.*

Proof. Divide the matrix C into two evenly divided submatrices C' and C'' of size $n \times n/2$. Estimate the number of nonzero entries in (AC') and (AC'') with the algorithm from Lemma 3.7, setting x to $h^{1/2}/\log^{1/2} n$. The submatrix containing the single nonzero entry is detected with probability $1 - \mathcal{O}(\log^{1/2} n/h^{1/2}) \geq 1 - \mathcal{O}(1/n^{1/4})$ using $\mathcal{O}(h/\log n)$ time. After $\log n$ queries, we detect the column $j \in [n]$ such that $\mathtt{nnz}((AC)_{*,j}) \neq 0$. We repeat such process for the rows of AC thus finding $i \in [n]$ such that $\mathtt{nnz}((AC)_{i,*}) \neq 0$. □

In contrast to Lemma 3.3, where the estimation was performed $d = \log n$ times, Lemma 3.8 gives weaker guarantees whenever the set of witnesses for the entry $(AC)_{i,j}$ has few elements or, equivalently, when $A_{i,*}$ and $C_{*,j}$ are highly sparse. Conversely, the denser the submatrix (analogously, the larger the ratio h/n) the more hash values are collected in the buffer.

Before restating the main theorem we prove that the estimation algorithm does not yield false positives, i.e. $\mathtt{nnz}((AC)_{*,j}) = 0$ while $\tilde{k}_j \neq 0$.

Lemma 3.10. *Let $A, C \in \{0,1\}^{n\times n}$ be matrices with pairwise orthogonal vectors, i.e. $AC = 0^{n\times n}$. Compute the estimate $\tilde{k}$ using the algorithm from Lemma 3.7. It holds $\tilde{k} = 0$.*

Proof. Let $W_A = \{w \in [n] \mid A_{i,w} \neq 0\}$ and $W_C = \{w \in [n] \mid C_{w,j} \neq 0\}$ as defined in the algorithm of Lemma 3.7. It holds $W = W_A \cup W_C$. Unless $A, C \in 0^{n\times n}$, $W \neq \emptyset$. Conversely, $W_A \cap W_C = \emptyset$. Assume, by contradiction, $W_A \cap W_C \neq \emptyset$. Then, there exists $w \in [n]$ such that

$A_{i,w} \neq 0$ and $C_{w,j} \neq 0$. Hence, $(AC)_{i,j} \neq 0$ since w is a witness for the inner product $\langle A_{i,*}, C_{*,j} \rangle$. Therefore, we reached a contradiction since we assumed $\texttt{nnz}((AC)_{i,*}) = 0$. It follows, $A_w \times C_w = \emptyset$ since A_w and C_w are mutually disjoint. □

Theorem 3.2. *Let $A, C \in \{0,1\}^{n \times n}$ be Boolean matrices and assume A and C have h nonzero entries. It is possible to compute all the k nonzero entries of $AC \in \{0,1\}^{n \times n}$ using expected $\mathcal{O}(h + h\sqrt{k} + k)$ time and expected $\mathcal{O}(\text{sort}(h) + h\sqrt{k}/B + k/B)$ I/Os cache obliviously.*

Proof. The proofs follows the outline of Theorem 3.1. We generate uniformly distributed random permutations π_r and π_c and we permute the rows and columns of A and C. In Algorithm 3.1, we replace the estimator of step **(2)** with algorithm from Lemma 3.8 and the estimator of step **(7)** with the algorithm from Lemma 3.9. The $\text{sort}(h)$ term in the I/O complexity stems from the sorting phase required by the estimator from Lemma 3.7, where sets A_w and C_w are sorted according to the witnesses $w \in W$. □

3.5 Probabilistic Analysis

The algorithms from Theorem 3.1 and Theorem 3.2 have two degrees of randomness. The first, related with the correctness of the output, has been analyzed in Chapter 3.4. There, we gave a probability of $1 - 1/n$ for detecting single entries with the algorithm from Theorem 3.1 and a probability of $1 - \mathcal{O}(1/n^{1/4})$ with the algorithm from Theorem 3.2. The second degree of randomness concerns the running time and the number of I/Os, which depends on the sparseness of the submatrices. We can recover entries from ambiguous estimators by running the estimation algorithm for each entry. This requires linear time in the size of the input matrices. If the random generated permutations do not distribute uniformly the entries across the matrix, submatrices may be denser than expected, leading to a worst case of k nonzero in $(AC)^\ell$ and $\mathcal{O}(hk)$ time and I/Os for recovery. In general, more than $\sqrt{k}$ nonzero per submatrix worsen our time bounds. In the following, we study tail bounds of our distribution and, under specific constraints, we give an exponentially decreasing probability of having highly dense submatrices in AC.

We would like to apply tail bounds à la Chernoff on the random variables $X_\ell = \sum_{z=1}^{k} X_{\ell,z}$. Unfortunately, the random variables

$X_{\ell,1}, \ldots, X_{\ell,k}$ are not independent. While independence was never required by Lemma 3.6, which follows from the linearity of expectation, Chernoff bounds require the random variables to be independent and tail bounds cannot be studied directly. Nevertheless, Dubhashi and Panconesi [DP09] show how to bypass independence whenever the random variables are negatively associated, see Definition 1.5. We proceed by following the analysis of [DP09].

As discussed in Section 1.6, for negatively associated random variable, we can still apply Chernoff-Hoeffding bounds and obtain exponentially decreasing probabilities for our probability distribution. In a restricted setting, we show how to prove that the random variables $X_{\ell,z}$ are negatively associated.

Lemma 3.11. *In the settings of Lemma 3.6, if entries on the same row or on the same column are forced not to share the same submatrix, then the random variables $X_{\ell,z}$, with $\ell, z \in [k]$, are negatively associated.*

Proof. Let $I, J \in \mathcal{F}$ be disjoint subsets from a family of disjoint subsets $\mathcal{F}$ of $[k]$ and let f, g be nondecreasing functions such that $f(0) = g(0) = 0$. Consider two entries $z \in I$ and $z' \in J$. We need to prove that the variables $X_{\ell,z}$ and $X_{\ell,z'}$ are negatively associated. If z and z' are in the same row/column, then $\mathbf{Pr}[(X_{\ell,z} = 1) \wedge (X_{\ell,z'} = 1)] = 0$ by hypothesis. Hence, $\mathbf{E}[f(X_{\ell,z})g(X_{\ell,z'})] = 0$. If the entries do not reside on the same row and on the same column, then $\mathbf{Pr}[(X_{\ell,z} = 1) \wedge (X_{\ell,z'} = 1)] = 1/k^2$. Hence, $\mathbf{E}[f(X_{\ell,z})g(X_{\ell,z'})] = 1/k^2$. The probability of an entry z being mapped in a submatrix $(AC)^\ell$ is given by $\mathbf{Pr}[X_{\ell,z} = 1] = 1/k$. It follows,

$$\mathbf{E}[f(X_{\ell,z})g(X_{\ell,z'})] \leq \mathbf{E}[f(X_{\ell,z})]\,\mathbf{E}[g(X_{\ell,z'})].$$

Similarly, consider I, J and f, g as above. Let $i \in I$ and $j \in J$. If $X_{i,z} = 1$ then $X_{j,z} = 0$ since, given $z \in (AC)^i$, $z \notin (AC)^j$. Hence $\mathbf{E}[f(X_{i,z})g(X_{j,z})] = 0$. It follows,

$$\mathbf{E}[f(X_{i,z})g(X_{j,z})] \leq \mathbf{E}[f(X_{i,z})]\,\mathbf{E}[g(X_{j,z})],$$

since $\mathbf{E}[f(X_{i,z})] \geq 0$ and $\mathbf{E}[g(X_{j,z})] \geq 0$ and f, g are nondecreasing. □

Since each variable is negatively associated then the entire family $X_{\ell,z}$, with $\ell, z \in [k]$, is negatively associated (closure under products, Definition 1.6). Additionally, since the variables $X_{\ell,z}$ are negatively associated from a family of disjoint subsets $\mathcal{F}$ then the variables

$X_\ell = \sum_{z=1}^{k} X_{\ell,z}$, with $\ell \in [k]$, are negatively associated (disjoint monotone aggregation, Definition 1.7).

The constraints introduced in Lemma 3.11 are restrictive as they force entries from the same row/column not to be placed in the same submatrix. Nevertheless, given two entries z and z' on the same row/column, if $z \in (AC)^\ell$ and $z' \in (AC)^\ell$ then proving Chernoff-Hoeffding bounds is nontrivial as $X_{\ell,z}$ and $X_{\ell,z'}$ are not even negatively associated.

Lemma 3.12. *Let $AC \in \{0,1\}^{n\times n}$ be the product of two Boolean matrices $A, C \in \{0,1\}^{n\times n}$ and let $k = \mathtt{nnz}(AC)$. Assume every row and every column of AC has at most $\sqrt{k}$ nonzero entries. Assume AC is the matrix resulting from permuting uniformly at random the rows of A and the columns of C. Divide AC into k submatrices $(AC)^\ell$ of size $n/\sqrt{k} \times n/\sqrt{k}$. If entries in the same row or in the same column do not share the same submatrix then the probability that a submatrix has strictly more than $\sqrt{k}$ entries is at most $1/2^{\sqrt{k}}$ for $k > 4e^2$.*

Proof. By Lemma 3.6, $\mathbf{E}[X_\ell] = 1$. The lemma follows from Lemma 3.11, Theorem 1.8 and the standard Chernoff-Hoeffding bound, see Theorem 1.4. □

In the general case, i.e. whenever entries in the same row or column share the same submatrix, we can still prove tail bounds. The latter are only polynomially decreasing on k and they are derived from Markov's inequality, Theorem 1.3.

Lemma 3.13. *In the settings of Lemma 3.6, if two entries in the same row or in the same column share the same submatrix, then the probability that a submatrix has more than $\sqrt{k}$ entries is at most $1/\sqrt{k}$.*

Proof. By Lemma 3.6, $\mathbf{E}[X_\ell] = 1$. The lemma follows from Markov's inequality, Theorem 1.3, by setting $\delta = \sqrt{k}$. □

3.6 Empirical Study of Sparse Submatrices

The sparseness of the submatrices is validated using empirical evidence. Figure 3.3 shows a script that, given an arbitrary $n \times n$ matrix A with $n = 10^2$, k nonzero entries and at most $\sqrt{k}$ nonzero entries per row and column, permute the rows and the columns uniformly at random. The matrix is then split in k submatrices of size $n/\sqrt{k} \times n/\sqrt{k}$ and, for each submatrix A^ℓ, we increase the counter relative to $\mathtt{nnz}(A^\ell)$ obtaining the

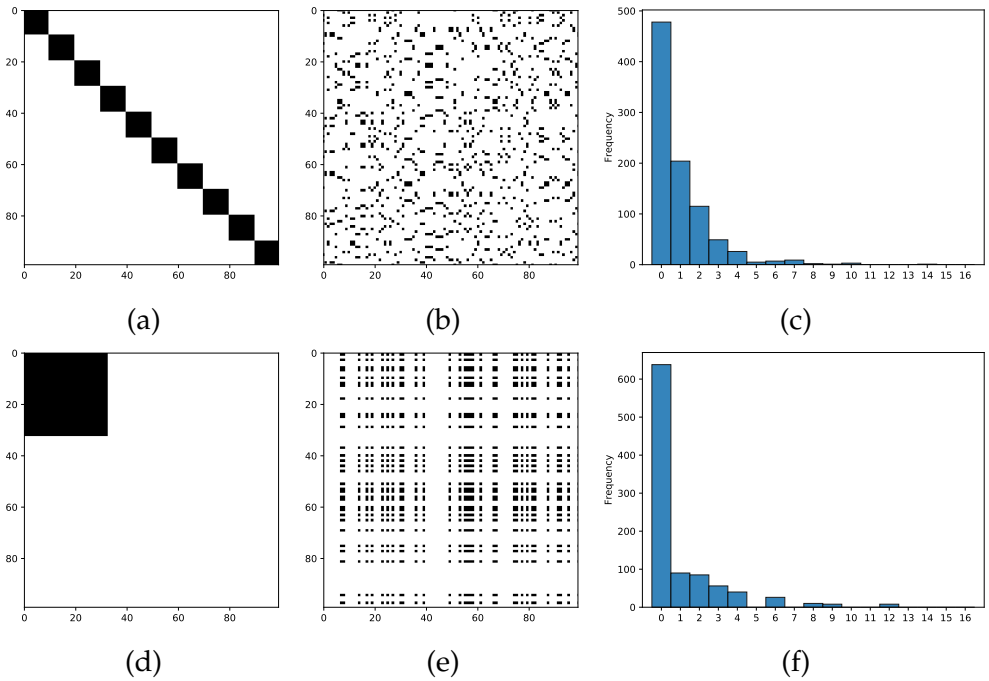


Figure 3.3: Block diagonal matrix before (a) and after (b) the rows and the columns were permuted uniformly at random. The frequency histogram (c) counts the number of submatrices with $x \in \{0, \ldots, 16\}$ nonzero entries. Block matrix before (d) and after (e) the rows and the columns were permuted with related frequency histogram (f).

related frequency histogram. While it is not straightforward to identify hard instances for this problem, we investigates how random permutation distribute entries for block diagonal and block matrices.

The block diagonal matrix has $k = 1000$ nonzero entries. Each row and column is d-sparse, meaning that there are exactly $d = 10$ nonzero entries. The matrix is split in k submatrices using $\lceil \sqrt{k} \rceil = 31$ row/column splitters thus obtaining submatrices of 4×4 size. The block matrix has $k = 1024$ nonzero entries. Each row and column has $\sqrt{k} = 32$ nonzero entries. The matrix is split in k submatrices using $\sqrt{k}$ row/column splitters thus obtaining submatrices of 4×4 size.

The frequency histograms show that the majority of the submatrices are empty. By Lemma 3.4 and Lemma 3.10, the estimation algorithms do not report nonzero entries for such submatrices. Conversely, the number of dense submatrices approaches zero rapidly. Note that, according to our parameters n and k, the probability of having more than $\sqrt{k}$ entries

is zero, since the submatrices have size 4×4. However, for different parameters we expect to see a more evident decreasing probability.

3.7 Conclusions

In this chapter we considered the problem of Boolean matrix multiplication in the Word RAM model and in the Cache Oblivious model. We presented a randomized algorithm that computes the product of Boolean matrices with worst case complexity matching the state of the art algorithm for Boolean matrix multiplication by [VG$^+$15] with the advantage of using, as a subroutine, estimation algorithms only. What is more, we improve over the aforementioned solution by providing a second, more refined algorithm that improves over our solution and over [VG$^+$15] in the polylogarithmic factors in regard to time complexity.

The partitioning step of our algorithms divides the output matrix in meta-rows and meta-columns of size $n/\sqrt{k}$. This technique seems to be relatively standard as it appears similarly in Van Gucht et al. [VG$^+$15] and Lingas [Lin09]. Amossen et al. [AP09] exploit improved partitioning techniques for isolating dense submatrices which are computed using fast matrix multiplication. This technique does not apply directly to our framework. We ask whether using estimators differently, e.g. by detecting zeros in highly dense submatrices, achieves improved bounds, similar to [AP09], e.g. $\mathcal{O}(h + h^{2/3}k^{2/3} + k)$.

The random permutation, when applied to input matrices, may not spread the nonzero entries of the output uniformly across the matrix. Dense submatrices with more than $\sqrt{k}$ nonzero entries worsen our time and I/O bounds. Although we prove polynomially decreasing probability of having dense submatrices, we ask whether we can achieve a partial derandomization by designing deterministic algorithms to partition the output matrix in submatrices with $\mathcal{O}(1)$ nonzero entries.

In relation to the sparseness of the submatrices, the tail bounds guarantee either polynomially or exponentially decreasing probability of having dense submatrices. In order to guarantee that time and I/O bounds deviate from their expectation with low probability, we have to prove that the probability of having at least one dense submatrix, i.e. with more than $\sqrt{k}$ nonzeros, is approaching zero rapidly as k grows. For our exponential tail bounds such probability is $1 - \sqrt{k}/2^{\sqrt{k}}$. Conversely, for our polynomial tail bounds the probability is unsatisfactory.

Consequently, we leave open the problem of proving stronger bounds (e.g. exponential tail inequalities) for the general case.

The algorithms presented in this chapter are Atlantic City, i.e. time, I/Os and output are random variables. On the front of derandomization, we ask whether we can design full Las Vegas algorithms, where only running time and number of I/Os are random variables while the output is always correct. The main hurdle for this problem is to provide deterministic algorithms for detecting nonzero entries in highly sparse submatrices. Note that, this problem is presumably easier compared to approximating the number of nonzero entries in a matrix product.

The framework we used for computing the Boolean matrix product can be extended straightforwardly to matrices with entries from a field: apply the algorithm from Theorem 3.1 and compute k inner products using kn additional operations. That said, the algorithm from Theorem 3.2 can only be extended to semirings as it does not capture cancellation of terms. The questions that arise are whether we can extend the size estimator from Lemma 3.7 to algebraic structures that allow cancellation of terms and whether this framework can be used to improve over state of the art algorithms for sparse matrix multiplication, e.g. [DJ18, Pag12].

Chapter 4

Compressed Sparse Matrix Multiplication

In this chapter we study the problem of sparse matrix multiplication in the Word RAM model and in the External Memory model when a fast matrix multiplication subroutine à la Coppersmith and Winograd [CW87] is used. We investigate how combining the combinatorial framework of Yuster and Zwick [YZ05] and Amossen and Pagh [AP09] with randomized compression techniques from Jacob and Stöckel [JS15] yields improved bounds for sparse matrix multiplication.

Let $A, C \in \mathbf{F}^{n \times n}$ be two matrices from a field $\mathbf{F}$ with $h = \mathtt{nnz}(A) + \mathtt{nnz}(C)$ nonzero entries in total. For balanced product matrices, we are able to compute all the k nonzero entries of the matrix product $AC \in \mathbf{F}^{n \times n}$ using $\tilde{\mathcal{O}}(hk^{(\omega-1)/4})$ time and $\tilde{\mathcal{O}}(hk^{(\omega-1)/4}/M^{(\omega-2)/4}B)$ I/Os for sparse output matrices and $\tilde{\mathcal{O}}(h^{2(\omega-2)/(\omega-1)}k^{1/(\omega-1)})$ time and $\tilde{\mathcal{O}}(h^{2(\omega-2)/\omega-1}k^{1/\omega-1}/M^{(\omega-2)/2(\omega-1)}B)$ I/Os for dense output matrices, where ω is the current exponent of fast matrix multiplication [LG14]. By exploiting Coppersmith and Winograd's rectangular fast matrix multiplication algorithm [Cop97] as subroutine, we provide a second algorithm that incurs in $\tilde{\mathcal{O}}(h^{0.690}k^{0.604})$ operations where the analysis is independent on the sparseness of the input/output matrices.

For unbalanced product matrices, we are able to compute all the k nonzero elements of the output matrix $AC \in \mathbf{F}^{n \times n}$ using $\tilde{\mathcal{O}}(hk^{(\omega-1)/(\omega+1)})$ time and $\tilde{\mathcal{O}}(hk^{(\omega-1)/(\omega+1)}/M^{(\omega-2)/2(\omega+1)}B)$ I/Os for sparse output matrices and $\tilde{\mathcal{O}}(h^{2(\omega-1)/(\omega+1)}k^{2/(\omega+1)})$ time and $\tilde{\mathcal{O}}(h^{2(\omega-1)/\omega+1}k^{2/\omega+1}/M^{(\omega-2)/2(\omega+1)}B)$ I/Os for dense output matrices.

The guarantees for computing the matrix product AC follow from the analysis of [JS15] and are given in terms of the size of the matrices and state a probability of at least $1 - 1/n$ to compute the matrix product.

4.1 Introduction

The technique exploited for deriving new algorithms for sparse matrix multiplication since [YZ05] was to partition the matrices into submatrices, apply fast matrix multiplication on dense parts and compute the remaining products using direct matrix multiplication algorithms.

The choice of fast matrix multiplication as a subroutine is straightforward: it provides the best asymptotic upper bound for multiplying matrices. Nevertheless, when it comes to real life applications, fast matrix multiplication does not perform well in practice as it only provides an advantage for matrices so large that cannot be processed by modern hardware [Rob05]. Due to the level of complexity, to the best of our knowledge, there are no implementation of fast matrix multiplication.

The choice of algorithms based on fast matrix multiplication seems therefore to be unreasonable. Nevertheless, algorithm designers use fast matrix multiplication as a black box and, theoretically, any algorithm for multiplying matrices can be substitute as a subroutine obtaining algorithms that may perform well in practice yet yield presumably suboptimal bounds. For instance, the algorithms presented in this chapter, as well as in [YZ05, AP09], can be redesigned to exploit Strassen's algorithm therefore obtaining a simpler yet suboptimal solution. In this regard, the algorithms presented in this chapter provide more of a framework for sparse matrix multiplication that can be adapted to any matrix multiplication subroutine.

4.2 Contributions

We study the problem of sparse matrix multiplication in the Word RAM model and in the External Memory model. We present an algorithm for multiplying matrices $A \in \mathbf{F}^{n\times n}$ and $C \in \mathbf{F}^{n\times n}$, with entries from a field $\mathbf{F}$ and $h = \mathtt{nnz}(A) + \mathtt{nnz}(C)$ nonzero entries which can be phrased in the following theorem.

Theorem 4.1. *Let $A, C \in \mathbf{F}^{n\times n}$ be sparse matrices with elements from a field $\mathbf{F}$ and $h = \mathtt{nnz}(A) + \mathtt{nnz}(C)$ nonzero entries. Furthermore, assume that the product $AC \in \mathbf{F}^{n\times n}$ has k nonzero entries and $\Theta(k/n)$ nonzeros per row and per column. It is possible to compute all the k nonzero entries of AC using*

(i) $\tilde{\mathcal{O}}(hk^{(\omega-1)/4})$ time for $h \geq k^{(\omega+1)/4}$ and $\tilde{\mathcal{O}}(hk^{(\omega-1)/4}/M^{(\omega-2)/4}B)$ I/Os for $h \geq k^{(\omega+1)/4}/M^{(\omega-2)/4}$;

(ii) $\tilde{\mathcal{O}}(h^{2(\omega-2)/(\omega-1)}k^{1/(\omega-1)})$ *time for* $h < k^{(\omega+1)/4}$ *and* $\tilde{\mathcal{O}}(h^{2(\omega-2)/(\omega-1)}k^{1/(\omega-1)}/M^{(\omega-2)/2(\omega-1)}B)$ *I/Os, for* $h < k^{(\omega+1)/4}/M^{(\omega-2)/4}$.

For the ease of exposition, we omit in the bounds the terms h and k which are implicit. With the current exponent of fast matrix multiplication, i.e. $\omega = 2.372$ from [LG14], the bounds of Theorem 4.1 correspond to $\tilde{\mathcal{O}}(hk^{0.343})$ operations for $h \geq k^{0.843}$ and $\tilde{\mathcal{O}}(hk^{0.343}/BM^{0.093})$ I/Os for $h \geq k^{0.843}/M^{0.093}$ and $\tilde{\mathcal{O}}(h^{0.542}k^{0.728})$ operations for $h < k^{0.843}$ and $\tilde{\mathcal{O}}(h^{0.542}k^{0.728}/BM^{0.186})$ I/Os for $h < k^{0.843}/M^{0.093}$.

The algorithm from Theorem 4.1 makes use of fast matrix multiplication invoked on square submatrices. By exploiting rectangular fast matrix multiplication [Cop97], we achieve the following bound.

Theorem 4.2. *Let* $A, C \in \mathbf{F}^{n\times n}$ *be sparse matrices with elements from a field* $\mathbf{F}$ *and* $h = \mathtt{nnz}(A) + \mathtt{nnz}(C)$ *nonzero entries. Furthermore, assume that the product* $AC \in \mathbf{F}^{n\times n}$ *has* k *nonzero entries and* $\Theta(k/n)$ *nonzeros per row and per column. It is possible to compute all the* k *nonzero entries of* AC *using* $\tilde{\mathcal{O}}(h^{0.690}k^{0.604})$ *time.*

For general product matrices, i.e. without any assumptions on balanced output, we design an algorithm with worst case complexity summarized in the following theorem.

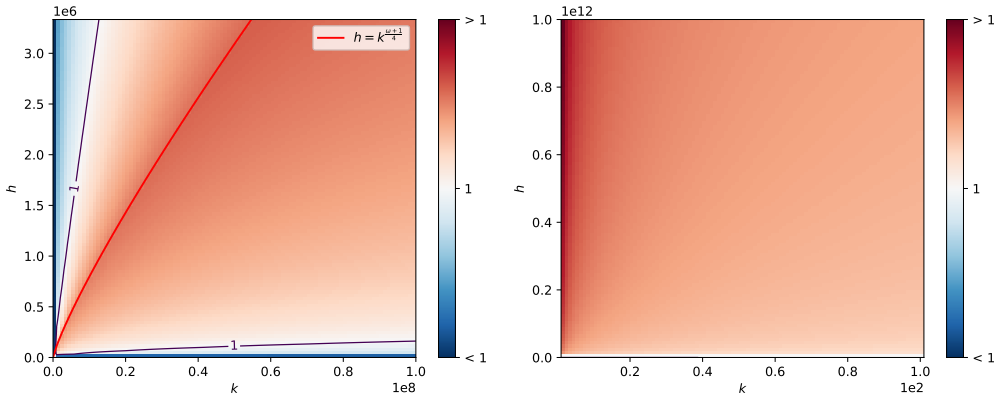
Theorem 4.3. *Let* $A, C \in \mathbf{F}^{n\times n}$ *be sparse matrices with elements from a field* $\mathbf{F}$ *and* $h = \mathtt{nnz}(A) + \mathtt{nnz}(C)$ *nonzero entries. It is possible to compute all the* k *nonzero entries of* AC *using*

(i) $\tilde{\mathcal{O}}(hk^{(\omega-1)/(\omega+1)})$ *time and* $\tilde{\mathcal{O}}(hk^{(\omega-1)/(\omega+1)}/M^{(\omega-2)/2(\omega+1)}B)$ *I/Os, for* $h \geq k$*;*

(ii) $\tilde{\mathcal{O}}(h^{2(\omega-1)/(\omega+1)}k^{2/(\omega+1)})$ *time and* $\tilde{\mathcal{O}}(h^{2(\omega-1)/(\omega+1)}k^{2/(\omega+1)}/M^{(\omega-2)/2(\omega+1)}B)$ *I/Os, for* $h < k$.

With the current exponent of fast matrix multiplication the bounds of Theorem 4.3 correspond to $\tilde{\mathcal{O}}(hk^{0.406})$ operations and $\tilde{\mathcal{O}}(hk^{0.406}/M^{0.093}B)$ I/Os for $h \geq k$ and $\tilde{\mathcal{O}}(h^{0.814}k^{0.593})$ operations and $\tilde{\mathcal{O}}(h^{0.814}k^{0.593}/M^{0.135}B)$ I/Os for $h \leq k$.

The guarantees for computing the matrix product are given in terms of the size of the input matrices and are derived from the analysis of [JS15]. Jacob and Stöckel guarantee an error probability of $1/n$.



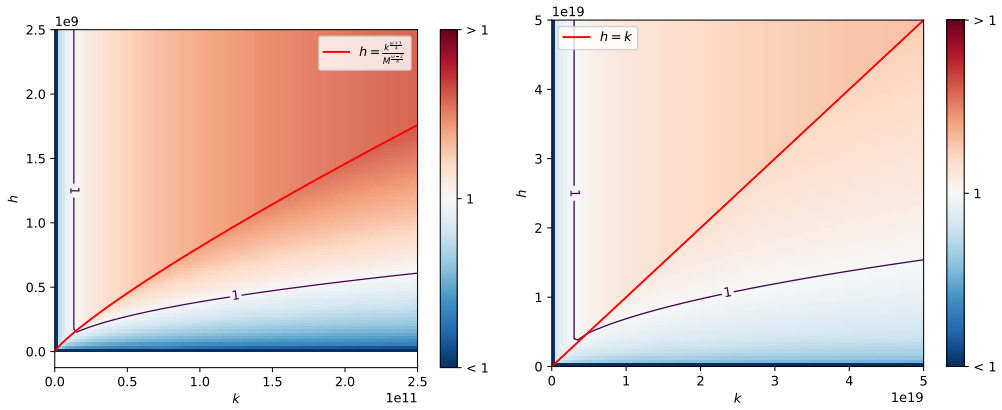
(a) (b)

Figure 4.1: Comparison between $f(h,k) = h^{2/3}k^{2/3} + h^{0.859}k^{0.406}$ and (a) $g(h,k) = hk^{(\omega-1)/4}$ for $h \geq k^{(\omega+1)/4}$ and $g(h,k) = h^{2(\omega-2)/(\omega-1)}k^{1/(\omega-1)}$ for $h \leq k^{(\omega+1)/4}$ and (b) $g(h,k) = h^{0.690}k^{0.604}$. Functions $f(h,k)$ and $g(h,k)$ are compared by computing the ratio function $R(h,k) = f(x,k)/g(h,k)$ and studying where $R(h,k) > 1$. The contour lines define boundaries where the function becomes smaller than one. The absence of contours in (b) indicates that the function $R(h,k)$ is always strictly larger than one, hence $f(h,k) > g(h,k)$.

4.3 Comparison with the Related work

Given two $n \times n$ matrices A and C, with $h = \texttt{nnz}(A) + \texttt{nnz}(C)$ nonzero entries, Yuster and Zwick [YZ05] designed a very simple algebraic algorithm for sparse matrix multiplication. Their idea was to partition the inner products into disjoint sets and perform direct matrix multiplication on one set and fast matrix multiplication on the other. This simple combinatorial idea yields an improved bound for sparse matrix multiplication, namely $\tilde{\mathcal{O}}(h^{0.7}n^{1.2} + n^{2+o(1)})$ operations which becomes $\tilde{\mathcal{O}}(h^{0.689}n^{1.209} + n^{2+o(1)})$ with the current ω from [LG14].

Amossen and Pagh [AP09] extended the result of Yuster and Zwick by considering a bivarite combinatorial problem, where they take advantage of outer products. The latter are split into two disjoint subsets where one subset is computed using direct matrix multiplication while the other is computed by partitioning inner products as in [YZ05]. This leads to an improved algorithm for sparse matrix multiplication that incurs in $\tilde{\mathcal{O}}(h^{2/3}k^{2/3} + h^{0.862}k^{0.408})$ operations (with ω



(a) (b)

Figure 4.2: Comparison between $f(h,k) = (h/B)\min\{\sqrt{k}/\sqrt{M}, h/M\}$ and (a) $g(h,k) = hk^{(\omega-1)/4}/M^{(\omega-2)/4}B$ for $h \geq k^{(\omega+1)/4}/M^{(\omega-2)/4}$ and $g(h,k) = h^{2(\omega-2)/(\omega-1)}k^{1/(\omega-1)}/M^{(\omega-2)/2(\omega-1)}B$ for $h < k^{(\omega+1)/4}/M^{(\omega-2)/4}$ and (b) $g(h,k) = hk^{(\omega-1)/(\omega+1)}/M^{(\omega-2)/2(\omega+1)}B$ for $h \geq k$ and $g(h,k) = h^{2(\omega-1)/(\omega+1)}k^{2/(\omega+1)}/M^{(\omega-2)/2(\omega+1)}B$ for $h < k$. The contour lines define boundaries where the function becomes smaller than one. We set the values of $M = 8000$ and $B = 16$ to simulate a real hardware setting with 32-bit integers, a 32 KB cache and a 16 Bytes cache line.

from [CW87]) which is equal to $\tilde{\mathcal{O}}(h^{2/3}k^{2/3} + h^{0.859}k^{0.406})$ with the current $\omega = 2.372$. Additionally, Amossen and Pagh provided an I/O complexity of $\tilde{\mathcal{O}}(h\sqrt{k}/M^{1/8}B)$ I/Os in the External Memory model.

Compared to Yuster and Zwick and Amossen and Pagh, we use randomized compression techniques from [JS15] and obtain improved bounds on sparse matrix multiplication. For the sparse case, namely when $h = k = \mathcal{O}(n)$, [YZ05] achieves $\tilde{\mathcal{O}}(n^{1.898})$ operations, [AP09] achieves $\tilde{\mathcal{O}}(n^{1.333})$ operations and $\tilde{\mathcal{O}}(n^{1.5}/M^{0.125}B)$ I/Os. For balanced output matrices, i.e. when every row and column of AC has $\Theta(k/n)$ nonzeros, our algorithm uses $\tilde{\mathcal{O}}(n^{1.270})$ operations and $\tilde{\mathcal{O}}(n^{1.270}/M^{0.093}B)$ I/Os, given $h \geq k^{(\omega+1)/4}$ and $h \geq k^{(\omega+1)/4}/M^{(\omega-2)/4}$, unless M is large. Using fast rectangular matrix multiplication, we achieve $\mathcal{O}(n^{1.294})$ operations independently from the input and output sparseness. In the general case, i.e. without assumptions on balanced output, we achieve $\tilde{\mathcal{O}}(n^{1.406})$ which is asymptotically worse than [AP09].

Nevertheless, since the exponents are dissimilar between the factors h and k, we still achieve an improvement in a limited parameter range.

Lingas [Lin09] presented an output-sensitive, randomized algorithm for computing the product of two $n \times n$ boolean matrices using $\tilde{\mathcal{O}}(n^2 k^{\omega/2-1})$ operations, which corresponds to $\mathcal{O}(n^{2.186})$ for $k = \mathcal{O}(n)$. For $k^{\omega+1/4} \leq h \leq n^2/k^{(3-\omega)/4}$ and $h \leq \min\{n/k^{1-\omega}, k^{(\omega+1)/4}\}$ our algorithm is asymptotically faster than [Lin09]. Moreover, our algorithm is designed for arbitrary fields and it is not restrict to Boolean matrices.

Jacob and Stöckel [JS15] presented a Monte Carlo algorithm that uses $\tilde{\mathcal{O}}(nk^{(\omega-1)/2})$ operations and, with high probability, outputs the nonzero elements of the matrix product. In addition, they state an I/O complexity of $\tilde{\mathcal{O}}(nk^{(\omega-1)/2}/M^{\omega/2-1}B)$ I/Os. We show that the same technique yields improved bounds for sparse matrix multiplication when combined to the combinatorial framework of [YZ05, AP09]. Namely, for sparse input matrices and balanced output, we achieve $\tilde{\mathcal{O}}(nk^{(\omega-1)/4})$ operations and $\tilde{\mathcal{O}}(nk^{(\omega-1)/4}/M^{(\omega-2)/4}B)$ I/Os, a factor of $k^{(\omega-1)/4}$ operations and $k^{(\omega-1)/4}/M^{(\omega-2)/4}$ I/Os faster than [JS15]. For generic output matrices, when $h = \mathcal{O}(n)$, we improve over [JS15] by a factor of $k^{(\omega-1)^2/2(\omega+1)}$ operations and $k^{(\omega-1)^2/2(\omega+1)}/M^{\omega(\omega-2)/2(\omega+1)}$ I/Os.

Van Gucht et al. [VG+15] presented a randomized algorithm for multiplying two Boolean matrices in $\tilde{\mathcal{O}}(h + h\sqrt{k} + k)$ time. This result was later improved by Dusefante [Dus18a] in the logarithmic factors, see also Chapter 3. Compared to their results, we are asymptotically faster than [VG+15, Dus18a] in almost every parameter range and our algorithms are not restricted to the Boolean semiring.

In the External Memory model, Pagh and Stöckel [PS14] provided an I/O optimal algorithm for matrix multiplication that incurs in $\tilde{\mathcal{O}}((h/B)\min\{\sqrt{k}/\sqrt{M}, h/M\})$ I/Os. Their algorithm is designed for semiring algebras and, therefore, does not allow cancellation of terms. Compared to their I/O bound, we improve over [PS14] unless M is large, see Figure 4.2 for a more thorough comparison.

4.4 Algorithms

The algorithms presented in this chapter are close in spirit with [YZ05, AP09] and they are design around three subroutines:

1. rectangular fast matrix multiplication (RFMM), via a blocking algorithm (Lemma 4.4), or actual RFMM (Lemma 4.7),

2. compressed matrix multiplication (Lemma 4.5 and Lemma 4.9),
3. matrix product estimation (Lemma 3.3 from Chapter 3).

The estimation algorithm is used solely for the unbalanced case. In the following we present the aforementioned primitives.

Algorithms overview The intuition for our algorithms is as follows: we filter out dense columns of A and dense rows of C which are computed directly using outer products. For the balanced case, the remaining rows and columns are compressed and computed using fast matrix multiplication. In the event of an unbalanced output matrix, we require an extra step. The rows of A and columns of C are split depending on the number of nonzero entries they produce in the output. Dense rows and dense columns of AC are computed with the direct matrix multiplication algorithm. The resulting sparse submatrix of AC is compressed using the data structure from [JS15] and computed using fast matrix multiplication. The balanced case is presented in detail in Algorithm 4.1 while the unbalanced case is presented in Algorithm 4.2.

4.4.1 Balanced Output Matrix

Fast matrix multiplication algorithms are commonly designed specifically for square matrices. A straightforward idea allows to apply these algorithms to the rectangular case: simply divide the matrices into square submatrices and perform fast matrix multiplication. Although this idea can be considered folklore, it appeared formally in [JS15, Fact 1]. We rephrase their result in an alternate form in the following lemma.

Lemma 4.4. *Let ω be the current exponent for fast matrix multiplication on two square $n \times n$ matrices. Consider two matrices of size $m \times n$ and $n \times p$. Fast rectangular matrix multiplication has $f(m,n,p) = \mathcal{O}(m^{\omega-2}np + mn^{\omega-2}p + mnp^{\omega-2})$ time complexity and $g(m,n,p) = \mathcal{O}(f(m,n,p)/M^{(\omega-2)/2}B)$ I/O complexity.*

Proof. Let $\mu = \min\{m,n,p\}$. Divide the matrices into submatrices of size $\mu \times \mu$. The number of subproblems is given by mnp/μ^3 and each subproblem can be solved using $\mathcal{O}(\mu^\omega)$ operations. Without loss of generality, assume $\mu = m < n < p$. It holds

$$\frac{mnp}{m^{3-\omega}} > \frac{mnp}{n^{3-\omega}} > \frac{mnp}{p^{3-\omega}}.$$

Therefore, the term $mnp \cdot \mu^{\omega-3}$ dominates which leads to the following complexity $\mathcal{O}(m^{\omega-2}np + mn^{\omega-2}p + mnp^{\omega-2})$. Regarding the I/O bound, consider a blocking argument similar to [JWK81]. The number of subproblems remains the same, i.e. mnp/μ^3. The number of I/Os is given by

$$\frac{mnp}{\mu^3} \cdot \left(\frac{\mu}{\sqrt{M}}\right)^{\omega} \cdot \frac{M}{B} = \frac{f(m,n,p)}{M^{\frac{\omega-2}{2}}B}$$

since each subproblem requires to multiply tiles of size $\sqrt{M} \times \sqrt{M}$. □

Jacob and Stöckel [JS15] proved that, for balanced output matrices, it is possible to multiply matrices faster using a clever compression technique. Their key result for balanced product matrix [JS15, Theorem 2] can be rephrased as follows.

Lemma 4.5. *Let* $\mathbf{F}$ *be an arbitrary field and let* $A \in \mathbf{F}^{n\times x}$ *and* $C \in \mathbf{F}^{x\times n}$ *where* $h = \mathtt{nnz}(A) + \mathtt{nnz}(C)$ *and* $k = \mathtt{nnz}(AC)$*. Assume each row and each column of* AC *has* $\Theta(k/n)$ *nonzero entries. Let* ω *be the current exponent of fast matrix multiplication. There is a data structure that can be initialized using* $f(\sqrt{k}, x, \sqrt{k}) = \tilde{\mathcal{O}}(x^{\omega-2}k + xk^{(\omega-1)/2})$ *time and can answer queries for entries in* AC *in time* $\mathcal{O}(\log n)$*.*

The intuition behind [JS15] is to collapse rows of A and columns of C, thereby obtaining a compressed version of the same problem. This compression technique is used by our algorithms to obtain improved bounds for sparse matrix multiplication. Therefore, the probabilistic nature of our algorithms rely solely on the the Monte Carlo data structure from [JS15]. The failure probability is derived by Jacob and Stöckel's analysis and state a probability of $1 - 1/n$ of computing the matrix product.

Theorem 4.1. *Let* $A, C \in \mathbf{F}^{n\times n}$ *be sparse matrices with elements from a field* $\mathbf{F}$ *and* $h = \mathtt{nnz}(A) + \mathtt{nnz}(C)$ *nonzero entries. Furthermore, assume that the product* $AC \in \mathbf{F}^{n\times n}$ *has* k *nonzero entries and* $\Theta(k/n)$ *nonzeros per row and per column. It is possible to compute all the* k *nonzero entries of* AC *using*

(i) $\tilde{\mathcal{O}}(hk^{(\omega-1)/4})$ *time for* $h \geq k^{(\omega+1)/4}$ *and* $\tilde{\mathcal{O}}(hk^{(\omega-1)/4}/M^{(\omega-2)/4}B)$ *I/Os for* $h \geq k^{(\omega+1)/4}/M^{(\omega-2)/4}$*;*

(ii) $\tilde{\mathcal{O}}(h^{2(\omega-2)/(\omega-1)}k^{1/(\omega-1)})$ *time for* $h < k^{(\omega+1)/4}$ *and* $\tilde{\mathcal{O}}(h^{2(\omega-2)/(\omega-1)}k^{1/(\omega-1)}/M^{(\omega-2)/2(\omega-1)}B)$ *I/Os, for* $h < k^{(\omega+1)/4}/M^{(\omega-2)/4}$*.*

Algorithm SBMM(A, C)
Input $A, C \in \mathbf{F}^{n\times n}$
Output $AC \in \mathbf{F}^{n\times n}$

(1) Let $A' = \{A_{*,j}\colon \mathtt{nnz}(A_{*,j}) \geq \tau\}$ and $C' = \{C_{i,*}\colon \mathtt{nnz}(C_{i,*}) \geq \tau\}$.
(2) Let $\bar{A}' = \{A_{*,j}\colon \mathtt{nnz}(A_{*,j}) < \tau\}$ and $\bar{C}' = \{C_{i,*}\colon \mathtt{nnz}(C_{i,*}) < \tau\}$.
(3) Compute $A'C'$ using the data structure of Lemma 4.5.
(4) Compute directly the matrix product $\bar{A}'\bar{C}'$ through outer products.
(5) Combine the matrix products $A'C'$ and $\bar{A}'\bar{C}'$.

Algorithm 4.1: Sparse Balanced Matrix Multiplication.

Proof. Steps **(1)**, **(2)** and **(5)** from Algorithm 4.1 require linear time in terms of the size of the input/output matrices. Step **(4)** requires $\mathcal{O}(h\tau)$ time and $\tilde{O}(h\tau/B)$ I/Os, see [AP09, Lemma 3.3]. In addition, the output matrix A'' has size $n \times h/\tau$ and the matrix C'' has size $h/\tau \times n$, by a counting argument. By hypothesis, the product matrix $A''C''$ of step **(3)** has $\Theta(k/n)$ nonzeros per row and per column. Therefore, we can apply the data structure of Lemma 4.5. Note that, in the settings of Lemma 4.5, $x = h/\tau$. The time required by the algorithm is

$$f(\sqrt{k}, h/\tau, \sqrt{k}) + h\tau. \tag{4.1}$$

We want to find τ such that it minimizes the function from Formula (4.1). With respect to τ, the function $f(\sqrt{k}, h/\tau, \sqrt{k})$ is monotonically decreasing while the function $h\tau$ is monotonically increasing. It follows that the function from Formula (4.1) is minimized whenever $f(\sqrt{k}, h/\tau, \sqrt{k}) = h\tau$. By Lemma 4.4 it holds

$$f(\sqrt{k}, h/\tau, \sqrt{k}) + h\tau = \frac{hk^{\frac{\omega-1}{2}}}{\tau} + \frac{h^{\omega-2}k}{\tau^{\omega-2}} + h\tau. \tag{4.2}$$

In order to minimize the complexity for the first term in (4.2) we set $(hk^{(\omega-1)/2})/\tau = h\tau$ and we solve for τ,

$$\frac{hk^{\frac{\omega-1}{2}}}{\tau} = h\tau$$
$$k^{\frac{\omega-1}{4}} = \tau,$$

which yields the bound $hk^{\frac{\omega-1}{4}}$. Similarly, to minimize the complexity in the second term of (4.2) we set $k(h/\tau)^{\omega-2} = h\tau$ and we solve for τ,

$$\begin{aligned} \frac{h^{\omega-2}k}{\tau^{\omega-2}} &= h\tau \\ h^{\omega-3}k &= \tau^{\omega-1} \\ h^{\frac{\omega-3}{\omega-1}}k^{\frac{1}{\omega-1}} &= \tau. \end{aligned}$$

which gives a bound equal to $h^{2(\omega-2)/(\omega-1)}k^{1/(\omega-1)}$. Combining the aforementioned bounds it holds

$$hk^{\frac{\omega-1}{4}} + h^{\frac{2(\omega-2)}{\omega-1}}k^{\frac{1}{\omega-1}}. \tag{4.3}$$

We now proceed to study Formula (4.3) in order to obtain a function $\mu(h,k)$ such that, for positive values of μ the first term dominates while for negative values the second term in (4.3) dominates the complexity.

$$\begin{aligned} hk^{\frac{\omega-1}{4}} &\geq h^{\frac{2(\omega-2)}{\omega-1}}k^{\frac{1}{\omega-1}} \\ h^{\frac{3-\omega}{\omega-1}} &\geq k^{\frac{4-(\omega-1)^2}{4(\omega-1)}} \\ h^{3-\omega} &\geq k^{\frac{3-\omega^2+2\omega}{4}} \\ h^{-(\omega-3)} &\geq k^{\frac{-(\omega-3)(\omega+1)}{4}} \\ h &\geq k^{\frac{\omega+1}{4}} \end{aligned}$$

If $h \geq k^{(\omega+1)/4}$, then the first term in Formula (4.3) dominates which yields an $\tilde{\mathcal{O}}(hk^{(\omega-1)/4})$ algorithm and the result from (i). If $h < k^{(\omega+1)/4}$, then the second term in Formula (4.3) dominates which gives a worst case complexity of $\tilde{\mathcal{O}}(h^{2(\omega-2)/(\omega-1)}k^{1/(\omega-1)})$ and the result from (ii). Regarding the I/O bounds, by Lemma 4.4, Formula (4.2) becomes

$$\frac{f(\sqrt{k}, h/\tau, \sqrt{k})}{M^{\frac{\omega-2}{2}}B} + \frac{h\tau}{B} = \frac{hk^{\frac{\omega-1}{2}}}{\tau M^{\frac{\omega-2}{2}}B} + \frac{h^{\omega-2}k}{\tau^{\omega-2}M^{\frac{\omega-2}{2}}B} + \frac{h\tau}{B}.$$

By the same argument, we obtain $\tau = k^{(\omega-1)/4}/M^{(\omega-2)/4}$ which yields the bound from the case (i). Similarly, for the case (ii), by substituting $\tau = h^{(\omega-3)/(\omega-1)}k^{1/(\omega-1)}/M^{(\omega-2)/2(\omega-1)}$ it holds the related complexity. The function for studying the dominant terms is the following $\mu(h,k) = h - k^{(\omega+1)/4}/M^{(\omega-2)/4}$ and it is derived by a similar analysis. □

Amossen and Pagh [AP09] proved that, if the exponent of matrix multiplication is $\omega = 2 + o(1)$, then there is an algorithm for matrix multiplication with worst case complexity of $\tilde{\mathcal{O}}(h^{2/3}k^{2/3})$ operations. Assuming $\omega = 2 + o(1)$, our algorithm improves over [AP09] as stated in the following corollary.

Corollary 4.6. *If $\omega = 2 + o(1)$ then Algorithm 4.1 has worst case complexity of $\tilde{\mathcal{O}}(hk^{1/4})$, for $h \geq k^{3/4}$, and worst case complexity of $\tilde{\mathcal{O}}(h + k)$, for $h \leq k^{3/4}$.*

As a consequence and considering $h = k = \mathcal{O}(n)$, Algorithm 4.1 requires $\tilde{\mathcal{O}}(n^{1.25})$ operations for dense output matrices ($h \geq k^{3/4}$) and $\tilde{\mathcal{O}}(n)$ operations for sparse output matrices ($h \leq k^{3/4}$).

The upper bounds derived in Theorem 4.1 are based on a blocking algorithm that solves rectangular matrix multiplication by dividing the matrices into square submatrices and performs fast (square) matrix multiplication on them, see Lemma 4.4. Alternatively, one may apply fast rectangular matrix multiplication on the input matrices.

Let $\omega(r, s, t)$ be the minimal exponent ω for which $f(n^r, n^s, n^t) = \mathcal{O}(n^{\omega+o(1)})$. Accordingly, $\omega(1, r, 1)$ is the exponent of rectangular matrix multiplication. Following [YZ05], let $\alpha = \max\{0 \leq r \leq 1 \colon \omega(1, r, 1) = 2\}$ and $\beta = (\omega - 2)/(1 - \alpha)$. Coppersmith [Cop97] proved that $\alpha > 0.294$. Hence, let

$$\omega(1, r, 1) \leq \begin{cases} 2 & \text{if } 0 \leq r \leq \alpha, \\ 2 + \beta(r - \alpha) & \text{otherwise.} \end{cases}$$

With $\omega = 2.372$ and $\alpha = 0.294$, we have $\beta = 0.526$.

Starting from $\omega(1, r, 1)$, Huang and Pan [HP98] proved the following result about fast rectangular matrix multiplication.

Lemma 4.7 (Huang and Pan [HP98])**.** *Consider two matrices of size $m \times n$ and $n \times m$. Fast matrix multiplication has*

$$f(m, n, m) = m^{2-\alpha\beta+o(1)}n^{\beta} + m^{2+o(1)}$$

running time.

Given the result from Lemma 4.7, we can modify Algorithm 4.1 to exploit actual rectangular matrix multiplication. The result is summarized in the following theorem.

Theorem 4.2. *Let $A, C \in \mathbf{F}^{n\times n}$ be sparse matrices with elements from a field $\mathbf{F}$ and $h = \mathtt{nnz}(A) + \mathtt{nnz}(C)$ nonzero entries. Furthermore, assume that the product $AC \in \mathbf{F}^{n\times n}$ has k nonzero entries and $\Theta(k/n)$ nonzeros per row and per column. It is possible to compute all the k nonzero entries of AC using $\tilde{\mathcal{O}}(h^{0.690}k^{0.604})$ time.*

Proof. In Theorem 4.1 we proved that the time required by Algorithm 4.1 is $f(\sqrt{k}, h/\tau, \sqrt{k}) + h\tau$. By Lemma 4.7, we have

$$f(\sqrt{k}, h/\tau, \sqrt{k}) + h\tau = k^{\frac{2-\alpha\beta}{2}} \left(\frac{h}{\tau}\right)^{\beta} + k + h\tau \tag{4.4}$$

In order to minimize the complexity we set $f(\sqrt{k}, h/\tau, \sqrt{k}) = h\tau$ and we solve for τ.

$$\begin{aligned} f(\sqrt{k}, h/\tau, \sqrt{k}) &= h\tau \\ k^{\frac{2-\alpha\beta}{2}} \left(\frac{h}{\tau}\right)^{\beta} &= h\tau \\ k^{\frac{2-\alpha\beta}{2}} h^{\beta-1} &= \tau^{\beta+1} \\ k^{\frac{2-\alpha\beta}{2(\beta+1)}} h^{\frac{\beta-1}{\beta+1}} &= \tau \end{aligned}$$

which simplifies to $\tau = k^{0.604}/h^{0.310}$ with the current values of α and β. By substituting τ in Formula (4.4) we have the claim. □

Corollary 4.8. *If $\omega = 2 + o(1)$ then Algorithm 4.1 has complexity $\tilde{\mathcal{O}}(\sqrt{hk})$.*

4.4.2 Unbalanced Output Matrix

Algorithm 4.1 relies on the assumption of a balance output. Without knowledge on the number of nonzero entries in the rows and columns of AC, we cannot apply Lemma 4.5. In order to overcome this issue, we need to extrapolate information on the structure of the output matrix.

Pagh and Stöckel [PS14] proved that is possible to compute a $1 \pm \varepsilon$ approximation of the number of nonzero entries in the rows and in the columns of the product matrix using quasilinear time. The result, proposed in Chapter 3 Lemma 3.3, sates that we can compute estimates $\tilde{k}_1, \ldots, \tilde{k}_n$ using $\mathcal{O}(\varepsilon^{-3} h \log(n) \log n)$ time and $\tilde{\mathcal{O}}(\varepsilon^{-3} h/B)$ I/Os. The estimates satisfy the following condition $(1-\varepsilon)\,\mathtt{nnz}((AC)_{i,*}) \leq \tilde{k}_i \leq (1+\varepsilon)\,\mathtt{nnz}((AC)_{i,*})$, for all $i \in [n]$, with probability $1 - 1/n$.

Algorithm SMM(A, C)
Input $A, C \in \mathbf{F}^{n \times n}$
Output $AC \in \mathbf{F}^{n \times n}$

(1) Let $A' = \{A_{*,j} \colon \mathtt{nnz}(A_{*,j}) \geq \tau\}$ and $C' = \{C_{i,*} \colon \mathtt{nnz}(C_{i,*}) \geq \tau\}$.
(2) Let $\bar{A}' = \{A_{*,j} \colon \mathtt{nnz}(A_{*,j}) < \tau\}$ and $\bar{C}' = \{C_{i,*} \colon \mathtt{nnz}(C_{i,*}) < \tau\}$.
(3) Compute directly the matrix product $\bar{A}'\bar{C}'$ through outer products.
(4) Compute estimates $\tilde{k}_1^T, \ldots, \tilde{k}_n^T$ for the number of nonzero entries in the rows of $(A'C')$ and estimates $\tilde{k}_1, \ldots, \tilde{k}_n$ for the number of nonzero entries in the columns of $(A'C')$.
(5) Let $A'' = \{A'_{i,*} \colon \tilde{k}_i^T \geq k/\tau\}$ and $C'' = \{C'_{*,j} \colon \tilde{k}_j \geq k/\tau\}$.
(6) Let $\bar{A}'' = \{A'_{i,*} \colon \tilde{k}_i^T < k/\tau\}$ and $\bar{C}'' = \{C'_{*,j} \colon \tilde{k}_j < k/\tau\}$.
(7) Compute directly the matrix product $A''C''$.
(8) Compute $\bar{A}''\bar{C}''$ using the data structure of Lemma 4.9.
(9) Combine the matrix products $\bar{A}'\bar{C}'$, $A''C''$ and $\bar{A}''\bar{C}''$.

Algorithm 4.2: Sparse Matrix Multiplication.

The primitive from [PS14] is used by Algorithm 4.2 to detect dense rows and columns in the output. The latter are computed directly. Conversely, sparse rows and columns are compressed using a randomized data structure by Jacob and Stöckel [JS15]. We propose their result in an alternate form, which follows directly from [JS15, Theorem 2] and Lemma 4.5.

Lemma 4.9. *Let* $\mathbf{F}$ *be an arbitrary field and let* $A \in \mathbf{F}^{n \times x}$ *and* $C \in \mathbf{F}^{x \times n}$ *where* $h = \mathtt{nnz}(A) + \mathtt{nnz}(C)$ *and* $k = \mathtt{nnz}(AC)$. *Assume each row and each column of* AC *has at most* δ *nonzero entries. Let* ω *be the current exponent of fast matrix multiplication. There is a data structure that can be initialized using* $f(\delta, x, \delta) = \tilde{\mathcal{O}}(x^{\omega-2}\delta^2 + x\delta^{\omega-1})$ *time and can answer queries for entries in* AC *in time* $\mathcal{O}(\log n)$.

The compression data structure of Lemma 4.9, together with the estimate algorithm from [AP09], give us all the necessary tools to prove our last main result.

Theorem 4.3. *Let $A, C \in \mathbf{F}^{n\times n}$ be sparse matrices with elements from a field $\mathbf{F}$ and $h = \mathtt{nnz}(A) + \mathtt{nnz}(C)$ nonzero entries. It is possible to compute all the k nonzero entries of AC using*

(i) $\tilde{\mathcal{O}}(hk^{(\omega-1)/(\omega+1)})$ time and $\tilde{\mathcal{O}}(hk^{(\omega-1)/(\omega+1)}/M^{(\omega-2)/2(\omega+1)}B)$ I/Os, for $h \geq k$;

(ii) $\tilde{\mathcal{O}}(h^{2(\omega-1)/(\omega+1)}k^{2/(\omega+1)})$ time and $\tilde{\mathcal{O}}(h^{2(\omega-1)/(\omega+1)}k^{2/(\omega+1)}/M^{(\omega-2)/2(\omega+1)}B)$ I/Os, for $h < k$.

Proof. Steps **(1)**, **(2)**, **(5)**, **(6)** and **(9)** from Algorithm 4.2 require linear time in terms of the size of the input/output matrices. In the proof of Theorem 4.1 we showed that computing the outer products of step **(3)** requires $\mathcal{O}(h\tau)$ time and $\tilde{\mathcal{O}}(h\tau/B)$ I/Os. By Lemma 3.3, computing estimates for the number of nonzero entries in the rows and columns of AC in step **(4)** requires $\tilde{\mathcal{O}}(h)$ operations and $\tilde{\mathcal{O}}(h/B)$ I/Os. The matrices A'' and C'' collect rows of A and columns of C that generate more than k/τ nonzero entries in rows and columns of AC. By a counting argument, A'' and C'' have at most τ rows and columns respectively. If not, there would be more than k nonzero entries in AC. Hence, step **(7)** requires $\tilde{\mathcal{O}}(h\tau)$ operations and $\tilde{\mathcal{O}}(h\tau/B)$ I/Os. By construction, the output submatrix $\bar{A}''\bar{C}''$ have at most k/τ nonzero entries per row and per column. Therefore, we can apply the data structure of Lemma 4.9. Recall that $\bar{A}''$ has h/τ columns and $\bar{C}''$ has h/τ rows. Summarizing, the complexity of Algorithm 4.2 is

$$f(k/\tau, h/\tau, k/\tau) + h\tau \tag{4.5}$$

Following the proof of Theorem 4.1, the function from Formula 4.5 is minimized when $f(k/\tau, h/\tau, k/\tau) = h\tau$. By Lemma 4.4 it holds

$$f(k/\tau, h/\tau, k/\tau) + h\tau = \frac{hk^{\omega-1}}{\tau^{\omega}} + \frac{h^{\omega-2}k^2}{\tau^{\omega}} + h\tau \tag{4.6}$$

For the firs term, we set $hk^{\omega-1}/\tau^{\omega} = h\tau$ and we solve for τ.

$$\begin{aligned} \frac{hk^{\omega-1}}{\tau^{\omega}} &= h\tau \\ k^{\omega-1} &= \tau^{\omega+1} \\ k^{\frac{\omega-1}{\omega+1}} &= \tau \end{aligned}$$

Similarly, we set $h^{\omega-2}k^2/\tau^\omega = h\tau$ and we solve for τ.

$$\begin{aligned} \frac{h^{\omega-2}k^2}{\tau^\omega} &= h\tau \\ h^{\omega-3}k^2 &= \tau^{\omega+1} \\ h^{\frac{\omega-3}{\omega+1}}k^{\frac{2}{\omega+1}} &= \tau \end{aligned}$$

In contrast to the proof of Theorem 4.1, where we needed to find a function $\mu(h,k)$ to discern which term dominates, the function is simply $\mu(h,k) = h-k$, since the minimum in Formula (4.5) is given by $\min\{h/\tau, k/\tau\} = \min\{h,k\}$. Accordingly, the time complexity follows by substituting the values of τ in Formula (4.5). Regarding the I/O complexity, the bounds follow from Lemma 4.4 and by the same argument from the proof of Theorem 4.1. □

Theorem 4.3 generates a compressed matrix of size $k/\tau \times k/\tau$. Such compression is consistent only if $k\tau < n$, namely if $k^{0.594} < n$ or $k^{0.406}/h^{0.813} < n$. Observe that, although we are introducing an additional constraint in Theorem 4.3, compression techniques are only compatible with sparse matrices, as for $\sqrt{k} = n$ no compression can be performed. We conclude our analysis with the following corollary that partially matches the result from [AP09].

Corollary 4.10. *If $\omega = 2 + o(1)$ then Algorithm 4.2 has worst case complexity of $\tilde{\mathcal{O}}(hk^{1/3})$, for $h \geq k$, and worst case complexity of $\tilde{\mathcal{O}}(h^{2/3}k^{2/3})$, for $h \leq k$.*

4.5 Conclusions

In this chapter we considered the problem of sparse matrix multiplication in the Word RAM model and in the External Memory model. We showed how to combine the combinatorial framework of Yuster and Zwick [YZ05] and Amossen and Pagh [AP09], with randomized compression techniques from Jacob and Stöckel [JS15] to obtain improved bounds for sparse matrix multiplication. The result is a collection of algorithms that exploit fast matrix multiplication kernels in order to multiply sparse matrices asymptotically faster in the balanced case and in the general case.

Motivated by the studies of [JS15] on balanced output matrices, we showed that, if the product matrix exhibits a balanced structure, then we

are able to multiply sparse matrices asymptotically faster compared to the state of the art and to the unbalanced case. Unfortunately, we were not able to replicate the same time bounds for unbalanced output matrices. Consequently, we leave open the problem of designing algorithms for the unbalanced case that achieve the same bound of the balanced one. In relation to this, the bound derived in the proof of Theorem 4.3 is not sharp as the compression is only down to $k/\tau \times k/\tau$ which, for $\tau < \sqrt{k}$, is suboptimal. Alternatively, one could think of proving lower bounds either on the combinatorial framework of [YZ05, AP09] or on the compression technique of [JS15]. As a matter of fact, the marginal improvements gained by our randomized algorithms over [AP09] suggest little hope for significant further improvements in this direction.

The algorithms have been designed and analyzed in the External Memory model. An interesting future direction is to extend the analysis to the Parallel External Memory model and the Cache Oblivious model. The absence of parallel algorithms for fast matrix multiplication leave little room for the distribution step. That is, each processor has to be responsible for fast-multiply batches of submatrices. For blocked-based rectangular fast matrix multiplication, this seems doable. For actual rectangular fast matrix multiplication à la Coppersmith [Cop97] this seems to require major efforts. On the oblivious front, in the Cache Oblivious model, one would need to find partitioning techniques, similar to Lemma 4.4, oblivious to memory and block size.

Algorithms that exploit fast matrix multiplication as a kernel are able to outperform most of the (if not every) present matrix multiplication algorithms. The impracticality of fast matrix multiplication makes these algorithms of theoretical interest only. The question we leave open is whether we can design algorithms matching the bounds of, e.g., [AP09, Dus18b] that do not rely on fast matrix multiplication primitives.

Chapter 5

Permuting in Parallel External Memory

In this chapter we investigate empirically the problem of permuting in a concurrent cache aware model. We present sorting-based algorithms for permuting elements in the Parallel External Memory model.

The result is a collection of I/O optimal algorithms that permute records in $\Theta(\min\{n/P, (n/PB)\log_d(n/B)\})$ I/Os with $d = \max\{2, \min\{M/B, n/PB\}\}$. Furthermore, we present efficient implementations of the aforementioned algorithms and we study how optimal concurrent out of memory permutation algorithms perform on modern parallel architectures.

We conduct extensive experiments for main-memory permutations where the algorithms are profiled according to several metrics, such as running times, cache and TLB misses. Finally, we test our algorithms against a state of the art numerical library for matrix permutation.

5.1 Introduction

Permuting is one of the simplest problems in algorithm theory, yet frequently used, e.g permutation matrices. The task of permuting consists of arranging n input records according to a permutation function. The problem is trivially solved with $\Theta(n/P)$ operations in PRAM, where P is the number of processors, and therefore it is not of algorithmic interest.

Nevertheless, when this problem is analyzed in the External Memory Model and, specifically, in the Parallel External Memory model, it becomes more challenging, [AGNS08, Guo11, Gre12, BF03]. Despite the simplicity of the task and its trivial complexity in PRAM, permuting in the Parallel External Memory model becomes nearly as hard as sorting.

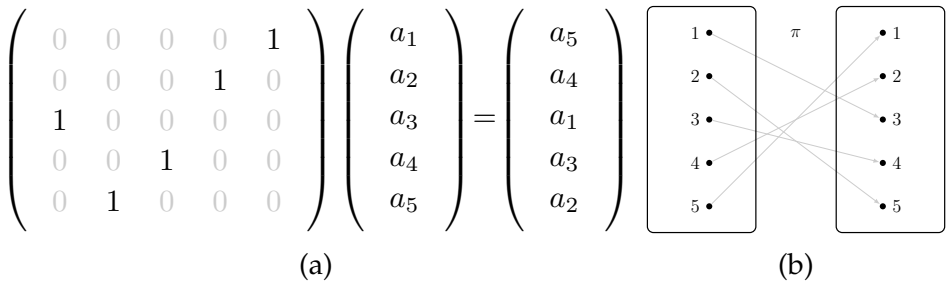


Figure 5.1: Permutation matrix Π (a) and its functional representation (b) as a map $\pi\colon [n] \to [n]$ which can be stored concisely as a vector $(3,5,4,2,1)$.

The problem of permuting can be defined as follows: given n records $a = (a_1, \ldots, a_n)$ and a permutation function $\pi\colon [n] \to [n]$, compute $c = (c_1, \ldots, c_n)$ such that $a_i = c_j$ if and only if $\pi(i) = j$, for $i, j \in [n]$.

The problem of permuting can also be defined from a linear algebra perspective. In this regard, consider $a = (a_1, \ldots, a_n)$ as above. The permutation is now defined as a $n \times n$ matrix Π. The task now becomes computing the sparse matrix vector product $\Pi a = c$ such that $a_i = c_j$ if and only if $\langle \Pi_{i,*}, a \rangle = j$, for $i, j \in [n]$. Intuitively, the problems are equivalent. Given a permutation matrix Π, it is always possible to represent Π concisely as a function $\pi\colon [n] \to [n]$ and vice versa. Such function is generated as follows: $\Pi^T_{i,j} = 1$ if and only if $\pi(i) = j$. Observe that it is sufficient to store the j values only.

For the ease of exposition, we consider only the case where $2 \leq M/B \leq n/PB$. This allows us to use interchangeably the variables $q = M/B$ and $d = \max\{2, \min\{M/B, n/PB\}\}$.

5.2 Contributions

Sorting-based algorithms are proven to be optimal in external memory models, under certain circumstances, for several problems, e.g. permuting or matrix computations [AV88, BBF+10, Gre12]. However, in parallel architectures, sorting-based techniques require thread synchronization in order to prevent race conditions and exploit high levels of parallelism. This may lead to severe performance reductions that can even cancel the benefits of optimal I/O algorithms.

In this chapter we provide deterministic data structures for thread synchronization applied to the problem of permuting. We present a first data structure that consists of a multi-layer table computed via tracing the positions the elements are assigned to, when a sorting-based algorithm is executed on the input records.

Similarly, we provide a second data structure that consist of a multi-layer table initialized with memory locations that identify conflict free regions. These regions are then exploited by a sorting-based algorithm to permute the input records. The data structure exploits a multi-level approach where each layer of the table induces a finer-grained partitioning of the elements into contiguous memory locations.

Finally, starting from the two previous ideas, we provide a hybrid data structure that combines the benefits of traces and thread synchronization tables. By using these data structures, we are able to overcome synchronization issues between processors, to exploit even further the parallelism provided by the architecture and to take advantage of I/O optimal, sorting-based algorithms.

Empirically, we address the question whether optimal algorithms in theoretical models, e.g. the Parallel External Memory model, exhibit high performance on modern parallel architectures. To do so, we provide C++ code that exploits memory management directives and algorithm engineering techniques in order to reduce performance issues such as false sharing and concurrent memory access. The implementation is then profiled using performance counters where we trace cache and TLB misses as well as running times. In order to highlight the efficiency of our implementation, we test our algorithms against a state of the art library for linear algebra that provides directives for matrix manipulation.

5.3 Comparison with the Related Work

Aggarwal and Vitter [AV88] studied the problem of permuting in the External Memory model. Specifically, they provided tight upper and lower bounds for the number of I/Os for several sorting-related problems including sorting, permutation networks, permuting and matrix transposition. For permuting in external memory, they proved an I/O bound of $\Theta(\min\{n, (n/B)\log_{M/B}(n/B)\})$ using a counting argument.

Nodine and Vitter [NV93] proposed a single disk, multiple processor model. They provide a multiprocessor generalization of the I/O model

in which each of the processors controls one disk and has an internal memory of size M/P. In this model they presented a load balancing strategy applied to the problem of external sorting with multiple disks and parallel processors. Vitter and Shriver [VS94], described the Parallel Disk model, a single processor, multiple disk model, where D blocks can simultaneously be read or written from/to D different disks. They provide upper and lower bounds for permuting, sorting and Fast Fourier Transform in the Parallel Disk model.

Brodal and Fagerberg [BF03] showed that permuting is not possible cache-obliviously using $\Theta(\min\{n, (n/B)\log_{M/B}(n/B)\})$, not even under the tall-cache assumption, i.e $M \geq B^{1+\epsilon}$. Intuitively, the minimum $\min\{n, (n/B)\log_{M/B}(n/B)\}$ cannot be decided without knowledge of M and B in order to apply either a sorting-based algorithm or perform the direct approach. Bender et al. [BFGK05] extended the cache oblivious model of Frigo et al. to a parallel setting consisting of P processors.

Arge et. al [AGNS08] studied parallel algorithms for private-cache chip multiprocessors (CMPs). They introduced the Parallel External Memory (PEM) model, and they provided lower bounds for several fundamental problems such as sorting and permuting. Bender et al. [BBF$^+$10] provided optimal algorithm for sparse matrix vector multiplication in the I/O model when different layouts are considered.

Guo [Guo11] performed empirical as well as theoretical studies on permuting in external memory. He proposed algorithms exploiting two different techniques, namely *funnels* and *buckets*. In addition, Guo's analysis covers upper bounds and experiments on several case studies. Greiner [Gre12], in his PhD dissertation, analyzed I/O algorithms in external memory. His effort is specifically focused on matrix multiplication in the Parallel External Memory model, however, his work covers lower bounds for permuting in Parallel External Memory and *bit-matrix-multiply/complement* permutations.

5.4 Algorithms

The direct algorithm for permuting, given n records and a permutation function, rearranges elements by directly applying the permutation to the input entries. As a matter of fact, it does not produce intermediate results and its trivial implementation can be summarized in the following formula:

$$c(\pi(i)) \leftarrow a(i), \quad \text{for } i \in [n]. \tag{5.1}$$

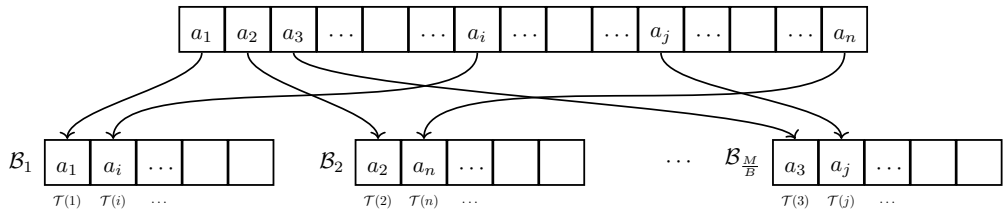


Figure 5.2: A distribution step of a sorting-based algorithm, where the elements are partitioned in M/B buckets. The TRACE data structure stores the intermediate positions, i.e. the trace, induced by the partitioning algorithm.

Despite its simplicity, the direct algorithm for permuting requires $\mathcal{O}(n/PB)$ I/Os in the best case and $\mathcal{O}(n/P)$ both in the average and worst case. The proof for the average case follows in the next section. The worst case occurs when the records are arranged in a way such that, to write each element a_i in position $\pi(i)$, it is necessary to perform an I/O, i.e. the memory address is not mirrored in internal memory.

The Formula (5.1) can be modified as follows $c(i) \leftarrow a(\pi^{-1}(i))$, where $\pi^{-1}(j) = i$ if and only if $\pi(i) = j$ is the inverse, in order to achieve optimal I/O behavior during write operations while loosing memory contiguity when reading the entries. However, the same analysis and considerations apply.

5.4.1 Permuting using Traces

Permutation algorithms that use random memory accesses do not ensure an optimal number of I/Os. Intuitively, two consecutive input records can be placed in two non-contiguous memory locations in the output that are not mirrored in internal memory.

In order to induce a more cache-efficient behavior during the permutation process, we provide a ℓ-layer data structure $\mathcal{T}$ where every level induces a finer-grained partitioning of the records into contiguous memory locations. More precisely, given a permutation $\pi = \pi(1), \pi(2), \ldots, \pi(n)$ we *decompose* the permutation in ℓ levels producing the following data structure, namely a $\ell \times n$ table, $\mathcal{T} = \mathcal{T}_1(1), \ldots, \mathcal{T}_1(n), \ldots, \mathcal{T}_\ell(1) \ldots \mathcal{T}_\ell(n)$, where $\mathcal{T}_l(i_l) = i$ denotes that, in the l-th layer, the record in i_l-th position is placed in position i. It

holds $\pi(i) = \mathcal{T}_\ell(i_{\ell-1}) \circ \mathcal{T}_{\ell-1}(i_{\ell-2}) \circ \cdots \circ \mathcal{T}_1(i)$ which can be rewritten as $\pi(i) = \mathcal{T}_\ell(\mathcal{T}_{\ell-1}(\cdots \mathcal{T}_1(i) \cdots))$ for every $i \in [n]$.

```
1  for l ← 1 to ℓ − 1 do
2      for b ← 1 to q^{l−1}
3          for i ← (b − 1) · δ + 1 to b · δ in parallel do        ▷ δ = n/q^{l−1}
4              bucket ← π_l(i)/q^l
5              mutex_lock(bucket)
6              index ← B(bucket) ← B(bucket) + 1
7              mutex_unlock(bucket)
8              T_l(i) ← index
9              π_{l+1}(index) ← π_l(i)
10 for i ← 1 to n in parallel grouped_by P do
11     T_ℓ(i) ← π_ℓ(i)
12 return T
```

Preprocessing 5.1: Initialization of the Trace data structure $\mathcal{T}$.

We denote with $\pi_l(i)$ the state of the permutation in the i-th position at the l-th level. Equivalently, we use the following notation $a_l(i)$. Preprocessing 5.1 works as follows: for every level $l \in [\ell - 1]$ (line 1), we divide the n records into subproblems of size $\delta = n/q^{l-1}$ (lines 2-3) and for each subproblem we partition in parallel the elements into buckets (lines 3-9). Such partitioning is computed by diving the elements into $q = M/B$ buffers (line 6), according to the value of *bucket* (line 4). The trace is then stored into the data structure (line 8) and the permutation is updated (line 9). The last layer ℓ stores the permutation that satisfies the original permutation (line 11). After $\mathcal{T}$ is computed, every layer of the data structure is applied to the records by Permuting 5.1 (lines 1-4), eventually arranging the elements according to the given permutation function.

```
1  for l ← 1 to ℓ do
2      for b ← 1 to q^{l−1}
3          for i ← (b − 1) · δ + 1 to b · δ in parallel do        ▷ δ = n/q^{l−1}
4              a_{l+1}(T_l(i)) ← a_l(i)
5  return c ← a_{ℓ+1}
```

Permuting 5.1: Permuting using the Trace data structure $\mathcal{T}$.

5.4.2 Permuting using Thread Synchronization Tables

The data structure initialization from Preprocessing 5.1 computes explicitly the permutation decomposition which is stored in the data structure

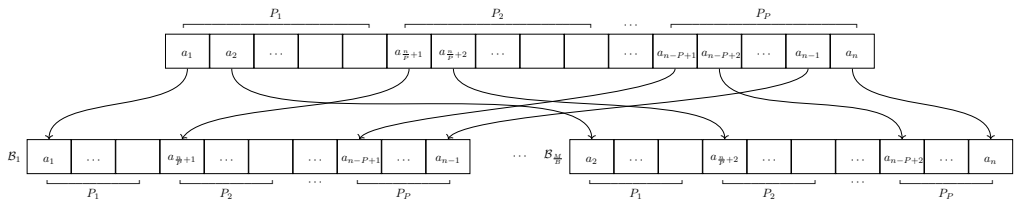


Figure 5.3: A distribution step of a sorting-based algorithm, where the elements are partitioned in M/B buckets. The Sync data structure stores the pointers to concurrent-free locations where each processor operates exclusively.

$\mathcal{T}$ and then applied, by Permuting 5.1, to the input records. Alternatively, in order to avoid storing the trace, we can compute intermediate positions on the fly.

```
1  for l ← 1 to ℓ − 1 do
2      for b ← 1 to q^{l−1}
3          for i ← (b − 1) · δ + 1 to b · δ in parallel do        ▷ δ = n/q^{l−1}
4              p ← GetProcessor()
5              bucket ← π_l(i)/q^l
6              I(p, bucket) ← I(p, bucket) + 1
7          S_l ← S_l^◇ ← ParallelPrefixSum(I)
8          for i ← (b − 1) · δ + 1 to b · δ in parallel do
9              p ← GetProcessor()
10             bucket ← π_l(i)/q^l
11             index ← S_l^◇(p · q^l + bucket) ← S_l^◇(p · q^l + bucket) + 1
12             π_{l+1}(index) ← π_l(i)
13 return S, π
```

Preprocessing 5.2: Initialization of the Sync data structure $\mathcal{S}$.

To manage concurrent memory access, it is possible to allocate, for each bucket, a fixed number of entries where each processor will write to. Therefore, we partition the permutation entries in P segments and for each segment we count the number of records for each bucket. Formally, we denote with $\mathcal{B}_j$ the j-th bucket, with P_i the i-th processor and with $q = M/B$ the number of buckets. The resulting table $\mathcal{I} = I(1,1), \ldots, I(1,q), \ldots, I(P,1), \ldots, I(P,q)$ consists of Pq elements where the generic entry $I(i,j)$ denotes the number of entries the processor P_i will write in bucket $\mathcal{B}_j$. A prefix sum of $\mathcal{I}$ gives us Pq pointers to as many memory locations where each processor will write regardless race conditions. This leads to the construction of the following ℓ-layer

data structure, with $l \in [\ell]$,

$$\mathcal{S}_l = S(1,1), \ldots, S(1,q), \ldots, S(P,1), \ldots, S(P,q), \text{ where } S(i,j) = \sum_{k=1}^{i} I(k,j).$$

Note that, the table $\mathcal{I}$ does not store the elements $I(1,i), \ldots, I(P,i)$ contiguously. Nevertheless, we can still compute parallel prefix sums using $\mathcal{O}(q/B + \log P)$ I/Os [Ble90, AGNS08], see also Theorem 5.3.

```
1  for l ← 1 to ℓ − 1 do
2      for b ← 1 to q^{l−1}
3          for i ← (b − 1) · δ + 1 to b · δ in parallel do
4              p ← GetProcessor()
5              bucket ← π_l(i)/q^l                          ▷ π_l from Preprocessing 5.2
6              index ← S_l(p, b · q^l + bucket)
7              S_l(p, b · q^l + bucket) ← S_l(p, b · q^l + bucket) + 1
8              a_{l+1}(index) ← a_l(i)
9  for b ← 1 to q^{ℓ−1}
10     for i ← (b − 1) · δ + 1 to b · δ in parallel do
11         c(π_ℓ + (i)) ← a_ℓ(i)
12 return c
```

Permuting 5.2: Permuting using the Sync data structure $\mathcal{S}$.

We use the following notation $\mathcal{S}_l(i,j)$, where $\mathcal{S}_l$ denotes the l-th level of the data structure. We start with Preprocessing 5.2. As in Preprocessing 5.1, at the l-th level, we generate q^{l-1} subproblems of size n/q^{l-1} (lines 2-3). Each subproblem is then divided in P contiguous segments and assigned to each processor. For each element in the segment, we compute its bucket and we increment the corresponding counter in $\mathcal{I}$ (lines 5-6). Once the table $\mathcal{I}$ is computed, we prefix-sum over $\mathcal{I}$ thus producing $\mathcal{S}_l$ (line 7). The permutation is then updated according to $\mathcal{S}_l$ using a local copy $\mathcal{S}_l^{\diamond}$ (lines 8-12). Permuting 5.2 applies the permutation to the input exploiting the data structure $\mathcal{S}$. Note that Preprocessing 5.1 and Permuting 5.2 are identical except for the locking phases that are replaced by the data structure $\mathcal{S}$ and for the trace which is stored in Preprocessing 5.1 (line 8) while it is directly applied to the input records in Permuting 5.2 (line 8).

5.4.3 Combining the Data Structures

The Trace data structure $\mathcal{T}$, obtained by decomposing the permutation function, introduces a performance issue known as *false sharing*. False

```
1  for l ← 1 to ℓ − 1 do
2      for b ← 1 to q^{l−1}
3          for i ← (b − 1) · δ + 1 to b · δ in parallel do        ▷ δ = n/q^{l−1}
4              p ← GetProcessor()
5              bucket ← π_l(i)/q^l
6              I(p, bucket) ← I(p, bucket) + 1
7          S_l ← S_l^◇ ← ParallelPrefixSum(I)
8          for i ← (b − 1) · δ + 1 to b · δ in parallel do
9              p ← GetProcessor()
10             bucket ← π_l(i)/q^l
11             H_l(i) ← index ← S_l^◇(p · q^l + bucket) ← S_l^◇(p · q^l + bucket) + 1
12             π_{l+1}(index) ← π_l(i)
13 for i ← 1 to n in parallel do
14     H_ℓ(i) ← π_ℓ(i)
15 return H
```

Preprocessing 5.3: Initialization of the Hybrid data structure $\mathcal{H}$.

sharing occurs on symmetric multiprocessing systems, where each processor has a local cache, when threads on different processors modify variables that reside on the same cache line.

Conversely, the thread synchronization table $\mathcal{S}$ does not cause such issue as it induces rigid concurrent-free locations where threads operate exclusively. Nevertheless, the data structure $\mathcal{S}$ requires more computational effort, compared to data structure $\mathcal{T}$, as intermediate positions need to be computed during the permutation process.

We provide a data structure $\mathcal{H}$ obtained by combining the ideas from the previous algorithms meeting the best of both worlds. The data structure $\mathcal{H}$ is a ℓ-layer data structure obtained by decomposing the permutation function using the trace defined by the data structure $\mathcal{S}$.

```
1  for l ← 1 to ℓ do
2      for b ← 1 to q^{l−1}
3          for i ← (b − 1) · δ + 1 to b · δ in parallel do
4              a_{l+1}(H_l(i)) ← a_l(i)
5  return c ← a_{ℓ+1}
```

Permuting 5.3: Permuting using the Hybrid data structure $\mathcal{H}$.

5.5 Theoretical Analysis

In this section we study the I/O complexity of algorithms of Section 5.4. We prove that Permuting 5.1, Permuting 5.2 and Permuting 5.3 are optimal in the PEM model. Conversely, we prove that, even in the average case, the direct algorithm for permuting is not I/O optimal.

Lemma 5.1. *Given n elements to permute with* Direct *using a random generated permutation, the expected number of blocks required to perform k write operations is $\Omega(k)$, with $1 \leq k \leq n/B$.*

Proof. Let π be a permutation chosen uniformly at random from the $n!$ feasible permutations. Define an indicator random variable

$$X_i = \begin{cases} 1 & \text{if block } i \text{ is required by at least one of the } k \text{ operations,} \\ 0 & \text{otherwise.} \end{cases}$$

The probability that a block i among the n/B total blocks is not used during the k operations is given by $(1 - B/n)^k$. It follows that the probability that the block i is required is $1 - (1 - B/n)^k$. Let $X = \sum_{i=1}^{k} X_i$ be the number of blocks required to perform k write operations with Direct, i.e. $c(\pi(i)) \leftarrow a(i)$. The expected number of blocks required for k operations is given by

$$\mathbf{E}[X] = \mathbf{E}\left[\sum_{i=1}^{n/B} X_i\right] = \sum_{i=1}^{n/B} \mathbf{E}[X_i] = \frac{n}{B}\left(1 - \left(1 - \frac{B}{n}\right)^k\right).$$

Given $(1 - x/n)^n = 1/e^x$, it follows,

$$\mathbf{E}[X] = \frac{n}{B}\left(1 - \left(1 - \frac{B}{n}\right)^k\right) = \frac{n}{B}\left(1 - \left(1 - \frac{B}{n}\right)^{\frac{kn}{n}}\right) = \frac{n}{B}\left(1 - \frac{1}{e^{\frac{Bk}{n}}}\right).$$

Since $1/e^x \geq 1 - x$, then

$$\mathbf{E}[X] = \frac{n}{B}\left(1 - \frac{1}{e^{\frac{Bk}{n}}}\right) \geq \frac{n}{B}\left(\frac{Bk}{n}\right) = k.$$

Hence, $\mathbf{E}[X] = \Omega(k)$. □

In the following, we study the I/O complexity of data structure initialization and permutation process of the algorithms of Section 5.4. As a consequence, we prove that permuting elements with the data structures of Section 5.4 induces an optimal number of I/Os.

Theorem 5.2. *Let $P \leq n/B$, after initializing the* Trace *data structure with Preprocessing 5.1 using $\mathcal{O}((n/PB)\log_d(n/B))$ I/Os, Permuting 5.1 permutes n records using $\mathcal{O}((n/PB)\log_d(n/B))$ I/Os and using $\mathcal{O}(n\log_d(n/B))$ space for the data structure, with $d = \max\{2, \min\{M/B, n/PB\}\}$.*

Proof. Let q be the distribution degree in each of the ℓ levels. Partitioning the n elements through the q buckets requires at most n/PB I/Os. Let $f = f(n, M, B, P)$ be the number of I/Os for parameters n, M, B and P. If $n/B \leq 1$ then $f = 1$. If $n/B > 1$ then we generate q subproblems of size n/qB. In general, q^l subproblems of size $n/q^l B$ are generated, with $l \in [\ell]$. Since $n/q^\ell B = 1$ then $\ell = \log_q(n/B)$. Hence, we derive

$$f(n, M, B, P) \leq \sum_{l=1}^{\ell} \mathcal{O}\left(\frac{n}{PB}\right).$$

It yields $f = \mathcal{O}((n/PB)\log_q(n/B))$. If $n/P < (n/PB)\log_q(n/B)$ then we can permute using the direct algorithm. Hence, it holds $f = \mathcal{O}(\min\{n/P, (n/PB)\log_q(n/B)\})$.

We can sharpen our analysis by studying how the number of buckets affect the complexity. If $M/B > n/PB$ then the number of buckets is greater than the number of I/Os required to scan the input. Hence, we consider $d' = \min\{M/B, n/PB\}$. If $d' < 2$ then we are considering only one bucket which means that no partitioning is performed on any of the ℓ layers. Therefore, we assume to always have at least two buckets available. Combining everything together we have $\ell = \log_d(n/B)$, where $d = \max\{2, \min\{M/B, n/PB\}\}$. It follows $f = \mathcal{O}(\min\{n/P, (n/PB)\log_d(n/B)\})$. The I/O complexity for Preprocessing 5.1 and Permuting 5.1 follows. Space complexity immediately follows by observing that the data structure requires $\mathcal{O}(n)$ space for each layer. □

Theorem 5.3. *Let $2 \leq P \leq n/B$, after initializing the* Sync *data structure with Preprocessing 5.2 using $\mathcal{O}((n/PB)\log_d(n/B) + n/B + \log P \log_d(n/B))$ I/Os, Permuting 5.2 permutes n records using $\mathcal{O}((n/PB)\log_d(n/B))$ I/Os and using $\mathcal{O}(Pn/B)$ space for the data structure, with $d = \max\{2, \min\{M/B, n/PB\}\}$.*

Proof. We divide the n entries between the P processors. Following the proof of Theorem 5.2, the distribution degree is given by $d = \max\{2, \min\{M/B, n/PB\}\}$. At the l-th level, every processors computes at most Pd^l pointers corresponding to table $\mathcal{I}$ using $\mathcal{O}(n/PB)$ I/Os. In order to initialize $\mathcal{S}_l$ we have to calculate a prefix sums of Pd^l total elements, namely $S_l(i, j) = \sum_{k=1}^{n} I(k, j)$, $\forall j \in [d^l]$. Since $P > d^l/B$, we need an extra $\log P$ term for *down-sweep* and *up-sweep* operations [Ble90, AGNS08] on blocks of size B. Therefore, $\mathcal{O}(d^l/B + \log P)$ I/Os

are required. Let $f = f(n, M, B, P)$ be the number of I/Os, for parameters n, M, B and P, then

$$\begin{aligned} f(n, M, B, P) &= \sum_{l=1}^{\ell-1} \left(\frac{n}{PB} + \frac{d^l}{B} + \log P \right) \\ &\leq \frac{n}{PB} \log_d \frac{n}{B} + \frac{(d^\ell - 1)}{d-1} + \log P \log_d \frac{n}{B} \\ &\leq \frac{n}{PB} \log_d \frac{n}{B} + \frac{n}{B} + \log P \log_d \frac{n}{B} \end{aligned}$$

where the first inequality hold since $\ell = \log_d(n/B)$ and we used $\sum_{i=1}^{k-1} x^i = (x^k - 1)/x - 1$. Regarding Permuting 5.2, the bound follows from the proof of Theorem 5.2. Space complexity follows by observing that, for each bucket, we store P pointers and there are d^l buckets at the l-th level. Hence,

$$\sum_{l=1}^{\ell-1} Pd^l = P \sum_{l=1}^{\ell-1} d^l = \frac{P(d^\ell - 1)}{d-1} \leq Pd^\ell = \frac{Pn}{B}$$

since we assumed $\ell = \log_d(n/B)$. □

Theorem 5.4. *Let $2 \leq P \leq n/B$, after initializing the* Hybrid *data structure with Preprocessing 5.3 using $\mathcal{O}((n/PB)\log_d(n/B) + n/B + \log P \log_d(n/B))$ I/Os, Permuting 5.3 permutes n records using $\mathcal{O}((n/PB)\log_d(n/B))$ I/Os and using $\mathcal{O}(n \log_d(n/B))$ space for the data structure, with $d = \max\{2, \min\{M/B, n/PB\}\}$.*

Proof. The proof follows from Theorem 5.2 and Theorem 5.3. □

Theorem 5.5 (Greiner [Gre12, Theorem 2.7])**.** *The average and worst case number of parallel I/Os required to permute n records in the PEM model with $P \leq n/B$ processors is $\Omega(\min\{n/P, (n/PB)\log_d(n/B)\})$ where $d = \max\{2, \min\{M/B, n/PB\}\}$.*

The proof of Theorem 5.5 relies on a counting argument through a time forward analysis. Briefly, given a family of programs that can generate every permutation of n records, there are at least $n!$ different configurations reached by the programs. The bound on the number of I/Os derives by considering the number of configurations reached by programs with ℓ I/Os [Gre12]. We conclude the section with the following result.

Corollary 5.6. *Permuting 5.1, Permuting 5.2 and Permuting 5.3 are optimal in the Parallel External Memory model in terms of parallel I/Os.*

5.6 Experiments

We conduct extensive experiments using the algorithms of Section 5.4. We identify the following experimental objectives that are addressed through empirical evaluation.

Q1: *How to specify the parameters to the algorithms?* Can we exploit theoretical insights in order to tune our algorithms and achieve optimal performance in practice?

Q2: *Is there a correlation between algorithms optimal in theory and optimal in practice?* Can we achieve a significant speedup by designing external memory algorithms? Are external memory algorithms efficient in practice or is their nature merely theoretical?

Q3: *How do data structures perform against state of the art permutation algorithms?* Can our data structures outperform the state of the art numerical algorithms for matrix manipulations?

Q4: *How do data structures perform when executed on different architectures?* We address the question whether different architectures yield different experimental results.

The questions we want to answer are strictly correlated. A theoretical model that reflects the structure of the physical architecture allows to tune the algorithms optimally which, in turn, correspond to high performance in practice.

5.6.1 Experimental setup

The algorithms of Section 5.4 are tested on architectures listed in Table 5.1. We implement the algorithms using C++11 and compile using g++5.1.1 with optimization flags `-O3` and `-march=native`. For Broadwell-EP we compile using g++5.4.0. Moreover, we use OpenMP 4.5, an API for multi-platform shared-memory parallel programming, to instruct our code with parallel directives.

Experiments make use of statistical data. Every data point is the result of the median among k repetitions. We set $11 \leq k \leq 51$ according to the size of the problem. In order to profile the code we use PAPI 5.2.0.0 while we validate the experimental results using Cachegrind from Valgrind 3.10.0. PAPI, Performance API, is a standard API for accessing

hardware performance counters while Cachegrind is a cache simulator. We use PAPI thread initialization together with thread inheritance in order to enable thread support and to collect, through hardware counters, cumulative cache misses deriving from multiple threads. The events we consider during profiling are `PAPI_Lx_TCM` for Lx cache misses and `PAPI_TLB_DM` for data TLB misses. `PAPI_L1_TCM` accounts for all demand requests and any code read to L2 cache. `PAPI_L2_TCM` accounts for each request originating from the core to reference a cache line in the L3 cache. `PAPI_L3_TCM` counts each cache miss condition for references to the last level cache. All the above events do not account for cache line fills due to prefetching. Lastly, `PAPI_TLB_DM` provides support fo data TLB load and store misses. To collect running times we use `std::chrono::high_resolution_clock` from the standard library.

We use the LMbench3 [MS12, MS+96] benchmark suite to compute memory latencies of the entire memory hierarchy. We permute random vectors of n 32-bit integers, where $n \in [10^7, 10^9]$, i.e. we consider main memory computations only. We generate random permutation using the Knuth/Fisher-Yates algorithm to avert biased distribution in the permutation function. We set P equal to the number of physical cores and we bind threads using `OMP_PROC_BIND=true`, thus disabling Hyper-Threading. This setup correspond to a hybrid model where we consider a PEM model with P physical cores, L1 and L2 as a hierarchical private memory and L3 and main memory as a hierarchical external memory shared by the processors which identifies an External Memory model.

The primary objective of the experiments is to evaluate optimal I/O algorithms in modern parallel architectures. To achieve this, we measure time, cache and TLB misses stemming only from the permutation phase and not from the data structure initialization. It has been observed [WD10, WTM13] that hardware counters produce nondeterministic results. Therefore, in order to collect reliable and meaningful metrics, we instruct only small parts of our code with performance measurement directives.

5.6.2 Experimental Evaluation

How to specify the parameters to the algorithms? The process of specifying the parameters for the data structures is crucial in order to achieve optimal performance. Memory size M, cache line B and the number

Table 5.1: The architectures where the experiments are performed.

Code	Broadwell-EP	Haswell-EP	Haswell
Arch	Xeon® E5-2690 v4	Xeon® E5-1650 v3	Core™ i7-4790
CPU	2×14, 2.6 - 3.5 GHz†	6, 3.5 - 3.8 GHz‡	4, 3.6 - 4.0 GHz‡
Line	64 B (L1/L2/L3)	64 B (L1/L2/L3)	64 B (L1/L2/L3)
L1	32 KB – 8-way	32 KB – 8-way	32 KB – 8-way
L2	256 KB – 8-way	256 KB – 8-way	256 KB – 8-way
L3	35000 KB – 20-way	15360 KB – 20-way	8192 KB – 16-way
RAM	528 GB	65 GB	32 GB
OS	Xenial 16.04.2 LTS	CentOS 3.10.0.x86_64	

† Two sockets with private L3 cache.
‡ Single socket.

of processors are not user dependent. However, although the number of cores is fixed, the number of threads is user-dependent and Hyper-Threading allow us to assign multiple threads per core. We design an experiment where we fix the problem size while we select the number of threads in an arbitrary interval, e.g. $[2, 120]$ for Haswell-EP. The results for this experiment report a local minimum when P is equal to the number of physical cores while the performance degrades whenever $2 \leq P < 6$, as the architecture is not fully exploited, and when $6 < P \leq 120$, as increasing the number of threads causes *cache trashing*, i.e. data eviction from the cache, in the levels of the hierarchy. This is in line with our expectations as permuting is an I/O-bounded problem, rather than CPU-bounded problem. As a result, we set P equal to the number of physical cores.

We now focus our tuning on the number of buckets and the number of levels ℓ. In terms of buckets, the theory suggests to set $q = M/B$, such that the internal memory can host at least a cache line from each bucket. In our experimental setup, using 32-bit integers and considering L2 as internal memory, $M = 65536$ and $B = 16$ for the machines listed in Table 5.1. It follows $q = 4096$. The theory suggests to set $\ell = \log_q(n/B)$. That is, in line with our experimental setup, $\ell \in \{1, 2\}$, since $\log_q(n/B) \in [1.60, 2.16]$ for $n \in [10^7, 10^9]$. Although this analysis is optimal for the number of the parallel I/Os between L2 and L3, it does not consider the cache misses generated between L3 and main memory which account for the most expensive operations for main memory computations [Lev09]. To do so, we consider $q = \min\{M/B, M_3/PB\}$,

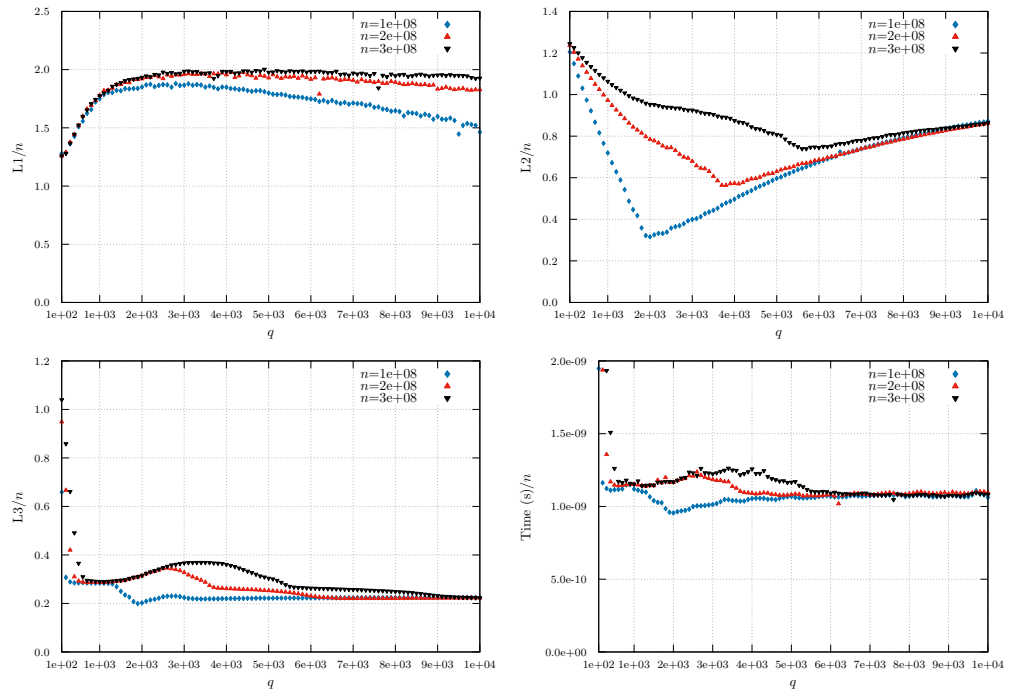


Figure 5.4: Permuting using the HYBRID data structure on Haswell-EP with fixed problem size $n \in \{1e{+}08, 2e{+}08, 3e{+}08\}$ and $q \in [10^2, 10^4]$.

where M_i specifies the number of elements in the Li cache. Every choice of q in the following interval $[M_2/B, M_3/PB]$ will increase the number of L2 cache misses while leaving the number of L3 cache misses unaltered. In other words, instead of increasing the number of layers whenever $\log_q(n/B)$ increases, which will unavoidably increase the number of L3 cache misses, we consider $\ell = \min\{\log_q(n/B), \log_{\bar{q}}(n/B)\}$, where $\bar{q} = M_3/PB$. According to our experimental setup and to the above analysis, we set $q = 4096$ and $\ell = 1$. Note that, permuting using an ℓ-layer data structure generates $\ell + 1$ different configurations, with $\ell = 0$ corresponding to permuting using DIRECT. We do not include results for $\ell = 2$ since we do not record any improvement when our data structures exploit more than one layer.

We now address the question whether the choice of q obtained via a theoretical analysis can be confirmed by the experiments. The distribution degree is as follows: $q = \min\{M_2/B, M_3/PB\}$ where M_2 and M_3 are the size of L2 and L3 cache respectively. This analysis provides more of an upper bound rather than an approximate result of q. Therefore we

design an experiment where we fix the problem size, i.e. n, while q ranges in a fixed subset of values, e.g. $[10^2, 10^4]$. We choose Permuting 5.3 for this task since it is not affected by false sharing, as Permuting 5.1, and it minimizes the computational effort, in contrast with Permuting 5.2. The results of such experiments are depicted in Figure 5.4. By comparing the best values for running times and cache misses we notice that, although our choice of q is not optimal, the advantage of using optimal values of q is negligible and in general, an exhaustive search is the only approach to ensure the optimality of the distribution degree q.

Is there a correlation between algorithms optimal in theory, namely in PEM, and optimal in practice? Once the parameters are set for the data structures, we address the question whether there is an actual improvement when using data structures to permute. To do so, we compare Permuting 5.1, Permuting 5.2 and Permuting 5.3 with DIRECT.

```
1  for i ← 1 to n in parallel do
2      c(π(i)) ← a(i)
```

Permuting 5.4: Permuting n records usingDIRECT.

The choice of DIRECT as the main competitor for our benchmarks is no coincidence. The memory access pattern induced by DIRECT on a random generated permutation can be considered as a generic cache-inefficient algorithm on a random input. For instance, sorting is a special case of permuting where the permutation induces an ordering of the elements. Hence, every sorting algorithm that is not optimized for I/O efficiency is lower bounded by DIRECT. This makes permuting an appropriate candidate for studying algorithms for data manipulation.

The experiments show that data structures ensure a reduction of cache misses for L2 and L3 caches. We do not report information on L1 cache misses as we are never able to outperform over L1 misses. In general, cache hits in the lower levels of the memory hierarchy account for the most expensive operations, namely 40 to 100 cycles for L3 on Intel® Xeon® processors [Lev09]. Hence, optimization in such levels accounts for the most effective. In order to measure the efficiency of our algorithms we rely on the following function to compute improvements factors

$$g(\mathbf{x}, \mathbf{y}) = \sqrt[n]{\prod_{i=1}^{n} \frac{x_i}{y_i}}$$

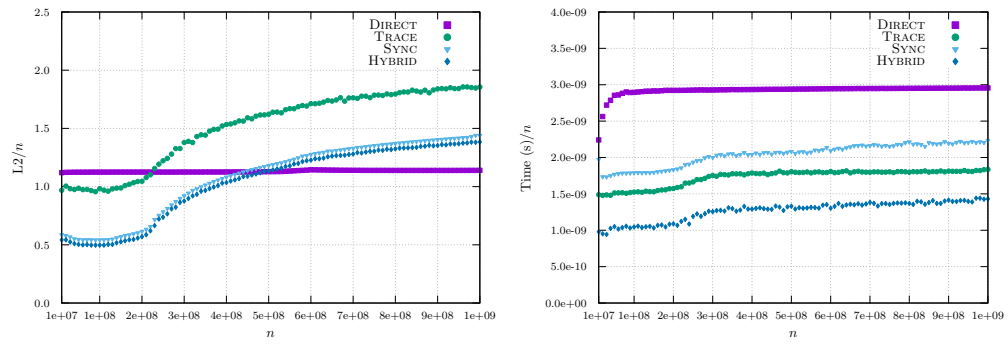


Figure 5.5: Permuting using the TRACE, SYNC and HYBRID data structures compared to DIRECT on Haswell-EP with a random generated permutation, $P = 6$ and $q = 4096$. Note the effect of false sharing on TRACE that induces an increase in the number of L2 cache misses.

where $\mathbf{x} = \{x_1, \ldots, x_n\}$, $\mathbf{y} = \{y_1, \ldots, y_n\}$ are the data points we want to compare pairwise. Given n different problem sizes, the function g computes the geometric mean over single improvement factors. The results are shown in Table 5.2. Experiments show improvement factors of more than 3.00 on the number of L3 cache misses for Haswell-EP and from 1.38 to 2.31 on running times for Permuting 5.2 and Permuting 5.3 respectively. The improvement over running times is even more prominent for Haswell, see also Figure 5.6, where the factors span from 2.72 of Permuting 5.2 to 4.06 of Permuting 5.3. Despite the high level of parallelism, we record slight improvement factors for Broadwell-EP with 1.54 and 1.86 for Permuting 5.1 and Permuting 5.2 respectively and 2.93 for Permuting 5.3. Such improvements are even more significant when the L3 cache stores only prefetched data. In addition, experiments show that a reduction on cache misses for Permuting 5.2, compared to Permuting 5.1, does not correspond to a reduction on running times. Although this may seem counterintuitive, Permuting 5.2 requires more computational effort which affects running times. The attempt to relate cache miss reduction with running time improvements, in order to strengthen the experimental results, was unsuccessful. As Levinthal [Lev09] reports, L3 cache hits have different latencies depending on whether cache lines are unshared, shared or modified by different cores thereby making any reverse analysis not reliable. A downside of algorithms from Section 5.4 is certainly data structures initialization which accounts for most of the computational process. In order to amortize the preprocessing time, we

Table 5.2: Performance improvement factors for permuting using Permuting 5.1, Permuting 5.2 and Permuting 5.3 compared to DIRECT for running times and cache misses using the geometric mean as an estimator for the average improvement.

	TRACE			SYNC			HYBRID		
	Time	L2	L3	Time	L2	L3	Time	L2	L3
Haswell	3.31	0.69	3.03	2.72	0.90	3.03	4.06	1.14	3.20
Haswell-EP	1.64	0.67	3.05	1.38	0.90	3.05	2.31	1.16	3.31
Broadwell-EP	1.54	0.65	1.30	1.86	1.05	2.92	2.93	1.09	2.92

acknowledge that applications of such data structures are restricted to scenarios where the permutation has to be applied repeatedly. We emphasize that all the experiments do not consider preprocessing time but only the mere permutation process.

How do data structures perform against state of the art permutation algorithms? In order to compare data structures of Section 5.4, we would like to benchmark different linear algebra libraries that provide directives for permutation matrices. To our knowledge, there is no C++ library that features parallel algorithms for permutation matrices.[1] EIGEN3 [GJ+10] features a `PermutationMatrix` with related member functions which does not support multithreading. However, it provides operators for matrix vector multiplication. We compare EIGEN3 `PermutationMatrix` with Permuting 5.1 setting $P = 1$. We do not compare Permuting 5.2 as the data structure $\mathcal{S}$ acts as a thread synchronization table which would be useless with a single thread. In addition, note that, for $P = 1$, $\mathcal{T} = \mathcal{H}$.

1 **Eigen**::**PermutationMatrix**<**Eigen**::**Dynamic**, **Eigen**::**Dynamic**, **uint32_t**> $\pi' = \pi$
2 **Eigen**::**VectorXuint32** $a' = A$ ▷ Custom 32-bit integer type
3 **Eigen**::**VectorXuint32** c
4 $c = \pi' \times a'$

Permuting 5.5: Permuting n records using EIGEN.

The data structure $\mathcal{T}$ is initialized using $q = 4096$, i.e. we consider the L2 cache as internal memory while the L3 cache is treated as external memory so as to preserve consistency with the experimental

[1] Boost [Sch11] linear algebra library UBLAS provides a data type `permutation_matrix` wich does not provide operators for matrix vector multiplication.

Table 5.3: Performance improvement factors for permuting using HYBRID compared to EIGEN3 for running times and cache misses using the geometric mean as an estimator for the average improvement.

	Time	L2	L3
Haswell	1.89	1.14	3.33
Haswell-EP	1.49	1.16	3.34
Broadwell-EP	1.40	1.10	2.88

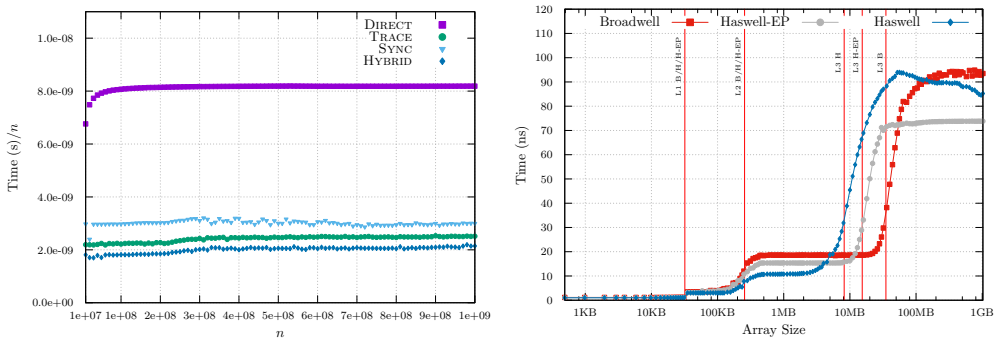
setup. By analyzing EIGEN3 source code, we verify that the C++ template function used by EIGEN3 to permute is indeed the direct algorithm for permuting using `std::swap` as routine to permute entries. Therefore, we claim that a possible multithreading implementation of EIGEN3 `PermutationMatrix` would not outperform our algorithms. For the sake of completeness, we stress that our implementations are not focused on single thread computations and we claim that experimental results can be improved even further for serial setups.

How do data structures perform when executed on different architectures? The data structures are architecture dependent. Hence, we expect to find dissimilarities in the experimental results when we permute using algorithms of Section 5.4 on machines listed in Table 5.1. Unfortunately, the difference in architecture between Haswell and Haswell-EP is not clean-cut. Conversely, Broadwell-EP is dissimilar to the other architectures as it features a two socket CPU with private L3 cache and the level of parallelism is higher compared to Haswell and Haswell-EP. The machines provide a three level memory hierarchy with identical cache sizes for L1 and L2 while Haswell has a L3 cache half the size of Haswell-EP. Broadwell-EP has a L3 cache of 35MB divided between private to each sockets. However, q remains unchanged as the minimum $\min\{M_2/B, M_3/PB\}$ is still determined by the size of the L2 cache. Conversely, the number of processors changes and consequently the size of data structure $\mathcal{S}$. Figure 5.6 shows the execution of Permuting 5.1, Permuting 5.2 and Permuting 5.3 on Haswell. By comparing the experiments from Haswell and Haswell-EP, we identify more similarities than dissimilarities. The difference on the number of processors reduces the size of data structure $\mathcal{S}$, however, the improvement is at most of $8 \cdot 10^{-4}$ cache misses per element. The size of the L3 cache influences the execution only partially. Read and write operations in L3 are performed sequentially, therefore the number of cache misses is

not dependent on M. However, using a larger L3 cache slightly improves upon the number of L2 and L3 cache misses for prefetched data. The improvement factors between L1, L2 and L3 match for the two architectures, even for the serial setup. Conversely, running times differ and Haswell is roughly a factor of 2.00 faster than Haswell-EP when both architectures are compared to DIRECT. Since there is no improvement in cache misses, as highlighted by the improvement factors, and since permuting is not a CPU bounded problem we conjecture that such factor is due to memory latency. To investigate this issue we use LMbench3 benchmark suite to compute memory latencies from L1 to main memory, see Figure 5.6. LMbench3 `lat_mem_rd` benchmark measures memory read latencies by traversing an array of fixed size with steps the size of the stride. By fixing the stride to 64 bytes, i.e. the size of the cache line, we ensure that every read operation is performed on a different cache line. Figure 5.6 shows the result of LMbench3 on both architectures and clearly demonstrates that Haswell main memory is slower than Haswell-EP while the contrary holds for L2 and L3 caches. As a results algorithms that rely on random accesses are penalized by high memory latencies and the improvement is more prominent when the difference in latencies is sharper. On the contrary, algorithms that rely on locality of memory benefit from fast caches. Nevertheless, such benchmark measures only "clean read" latencies while, in our experiments, writes are responsible for the majority of cache misses. When the difference in architectures is more consistent, i.e. Haswell/Haswell-EP and Broadwell-EP, dissimilarities are more evident. As highlighted in Section 5.4, Permuting 5.1 is affected by false sharing. This performance issue is evident for Haswell and Haswell-EP from the L2 counts while it is even sharper for Broadwell-EP where false sharing affect even the L3 counts as the L3 cache is private to each socket. As a consequence, despite the high level of parallelism, Broadwell-EP scores the lowest results for running times, L2 and L3 cache misses for Permuting 5.1.

5.7 Empirical Validation

The experiments make use of statistical measurements in order to reduce the noise deriving from the background activity of the operative system. The median of k repetitions gives an accurate estimate of both running times and cache misses. However, we detected significant oscillations in some experiments. Such oscillations derive from several aspects. The



(a) (b)

Figure 5.6: (a) Permuting using the TRACE, SYNC and HYBRID data structures compared to DIRECT on Haswell with $P = 4$ and $q = 4096$. (b) Latency benchmark obtained with LMbench3 using an array of 1GB and a stride of 64 bytes. The boundary lines indicate L1, L2 and L3 memory sizes for each architecture. As LMbench benchmark shows, Haswell has the fastest L2 and L3 caches which translates to the most significant improvements.

technology developed in order to improve cached data can influence data measurements. For instance, prefetchers are commonly used in caches and they can alter the number of cache misses as they are invoked in kernel-space instead of being user-space operations. The latter are generally the only activity accounted by profilers.

Cache replacement strategies affect I/O complexity only by a constant factor. Nevertheless, when it comes to profiling, cache replacement strategies can alter the results of the experiments. Replacement policies are often unknown, however, some Intel architectures have been reverse engineered. As a result, the Haswell architecture uses adaptive cache replacement policies which only behave as pseudo-LRU in some situations [GMM16]. Pseudo-LRU refers to an optimized LRU strategy where the age of the cache lines is approximated rather than computed exactly.

We acknowledge that the tools we used to produce the results suffer from precision issues and the measurements are not guaranteed to be exact throughout the whole execution. In addition, as Weaver et al. [WD10, WTM13] suggest, for several performance events, hardware counters cannot produce expected and deterministic results. We can overcome hardware counters and exploit cache simulators in order to profile the execution. Nevertheless, despite the precision and the re-

Table 5.4: Number of cache misses per element when permuting n records using a random generated permutation and the identity permutation using the direct algorithm on Haswell-EP. We set $P = 6$ and we use Cachegrind to profile the execution. Cache misses are listed divided by read and write operations for L1, L2 and L3.

n	Direct (Random)					
	L1W	L2W	L3W	L1R	L2R	L3R
10^6	0.993	0.946	0.062	0.125	0.125	0
10^7	0.999	0.994	0.680	0.125	0.125	0.125
10^8	0.999	0.999	0.967	0.125	0.125	0.125
10^9	0.999	0.999	0.996	0.125	0.125	0.125
n	Direct (Identity)					
	L1W	L2W	L3W	L1R	L2R	L3R
10^6	0.062	0.062	0.062	0.125	0.125	0
10^7	0.062	0.062	0.062	0.125	0.125	0.125
10^8	0.062	0.062	0.062	0.125	0.125	0.125
10^9	0.062	0.062	0.062	0.125	0.125	0.125

peatability of the results obtained using simulators we acknowledge that they do not reflect a true execution on the real hardware. For instance, Cachegrind schedules threads differently from the native configuration. It accounts neither for kernel activity nor for virtual-to-physical address mappings nor for cache misses arising from TLB misses. Regarding replacement strategies, Cachegrind simulates LRU and, as our experiments confirm, it can closely approximate Haswell replacement algorithm. Despite the limitations of Cachegrind, we use cache simulators in order to validate our results up to a certain accuracy.

5.7.1 Cache Misses Validation

Given n 32-bit integers and a permutation function expressed as n 32-bit integers, the direct algorithm reads sequentially the permutation function and the input records using n/B I/Os each. According to Table 5.1, $B = 16$, hence, we predict $2/16 = 0.125$ I/Os per element for L1, L2 and L3, see Table 5.4. This analysis is identical for both random and identity permutations as the permutation function does not affect the number of reading operations. On the other hand, writing is affected

by the permutation map and the number of writings can be predicted only for the identity while it can be approximated for the random permutation. Permuting with the identity permutation induces sequential writes. Hence, as above, we predict $1/16 = 0.0625$ write misses per element. Regarding random permutations, on average, the direct algorithm for permuting induces one write miss per element which matches with writing operations of Table 5.4, as long as the problem size saturates the caches. As expected, the number of write operation is settled for L1, as it can store approximately $8 \cdot 10^3$ 32-bit integers. Regarding L3, note that, on Haswell-EP, the cache can store less than $4 \cdot 10^6$ integers. Consequently, for 10^6 records, the number of L3 writes corresponds to scanning while it reaches approximately one write miss per element for $n \geq 10^8$. By comparing Figure 5.7 and Figure 5.8 with Table 5.4, we can see that the data matches for the random permutation, while we identify a significant gap between PAPI and Cachegrind for the identity permutation. The explanation relies on the hardware prefetcher. Prefetchers work by predicting certain memory patterns and preloading cache lines in memory in order to reduce memory latencies. In the case of the identity permutation, the direct algorithm exploits prefetched data as it access the input in a streaming fashion. On the other hand, random permutations induce random memory patterns and there is no guarantee that prefetched data will be used by the algorithm.

We can apply the same analysis to Permuting 5.1, Permuting 5.2 and Permuting 5.3, see Table 5.5.[2] Unfortunately, as the complexity of the algorithms increases it becomes more difficult to predict the number of cache misses. Permuting 5.1 works by reading sequentially the input records and the data structure $\mathcal{T}$ and writing sequentially in the output vector. Since $\ell = 1$ and $B = 16$, we predict $4/16 = 0.250$ read misses and $2/16 = 0.125$ write misses per element. Similarly, Permuting 5.2 works by reading sequentially the input records, the permutation function and the data structure $\mathcal{S}$ and writing sequentially in the output vector. Write misses account for $2/16 = 0.125$ as above, for $\ell = 1$ and $B = 16$. Conversely, we would expect to have $6/16 = 0.375$ read misses per elements since we are considering more memory locations. However, according to our experimental setup, $\mathcal{S}$ stores only $2.4 \cdot 10^4$ pointers which accounts for $1.5 \cdot 10^{-4}$ additional cache misses per element for $n = 10^7$. This argument apply only for L2 and L3 since $\mathcal{S}$ fits in memory while it does not

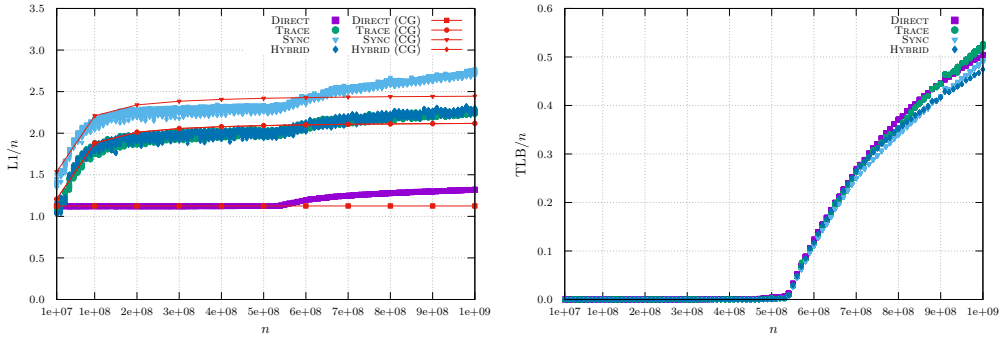
[2]Note that, from a theoretical standpoint, Permuting 5.1 and Permuting 5.3 are identical. Hence from now on we only consider Permuting 5.1.

Table 5.5: Number of cache misses per element when permuting n records with a random generated permutation using Permuting 5.1 and Permuting 5.2 on Haswell-EP. We set $P = 6$, $\ell = 1$ and $q = 4096$ and we use Cachegrind to profile the execution. Cache misses are listed divided by read and write operations for L1, L2 and L3.

n	Trace, Hybrid					
	L1W	L2W	L3W	L1R	L2R	L3R
10^6	0.958	0.354	0.125	0.250	0.250	0.170
10^7	0.958	0.360	0.127	0.250	0.250	0.249
10^8	1.631	0.365	0.125	0.250	0.250	0.249
10^9	1.867	1.092	0.125	0.250	0.250	0.250
n	Sync					
	L1W	L2W	L3W	L1R	L2R	L3R
10^6	0.994	0.387	0.125	0.543	0.251	0.067
10^7	0.994	0.391	0.127	0.542	0.250	0.249
10^8	1.667	0.398	0.125	0.542	0.250	0.250
10^9	1.904	1.121	0.125	0.540	0.250	0.250

apply for L1 which can store only $8 \cdot 10^3$ elements. In general, our predictions are accurate for reading operations on all levels of the memory hierarchy. On the other hand, writing operations are predictable only for L3 cache while they can be approximated for the higher levels of the memory hierarchy. According to Table 5.5, write operations account for one cache miss per element for L1 cache as the problem size increases. This has to be expected since we are not optimizing for the higher levels of the hierarchy.

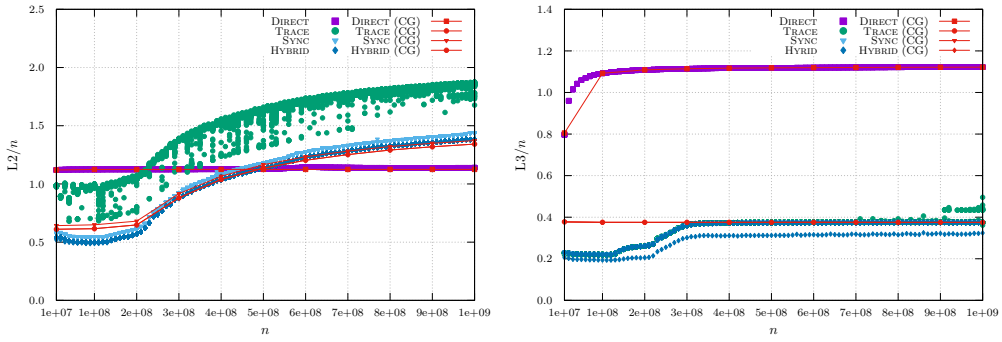
The data collected through cache simulators has proven to be predictable for certain operations and caches. However, we already emphasize how cache simulators do not always reflect a true execution on the real hardware. We address the issue whether data collected via hardware counters is somehow comparable to Cachegrind data. Figure 5.7 and Figure 5.8 show the comparison between PAPI and Cachegrind when profiling the exact same algorithms. The figures reveal clear discrepancies between the profilers and confirm the limitation of cache simulators. Starting from L1 cache misses, Figure 5.8, we identify a dissimilarity between graphs for $n \geq 5 \cdot 10^8$ that correspond to an increment on TLB misses. As stated, Cachegrind does not account for cache misses



(a) (b)

Figure 5.7: Raw data of L1 (a) and TLB (b) misses from permuting using the Trace, Sync and Hybrid data structures compared to Direct on Haswell-EP with a random generated permutation with parameters $P = 6$ and $q = 4096$. Each x-coordinate collects k repetitions of the same algorithm, with $k = 51$ for Permuting 5.1 and Permuting 5.2 and $k = 11$ for the direct algorithm. The reference plots (CG) are obtained by simulating the same algorithms using Cachegrind, see Table 5.4 and Table 5.5. Graphs are scaled by a $1/n$ factor to depict information per element.

arising from TLB misses, hence, such dissimilarity has to be expected. Regarding L3 cache misses, Figure 5.8, we identify some inconsistencies as well. The number of L3 cache misses is lower than what reported by Cachegrind for $n \leq 3 \cdot 10^8$ and $q = 4096$. The explanation relies on the hardware prefetcher that preloads in memory cache lines to improve memory management. The result is a reduction on cache misses which are not accounted by profilers as they are performed in kernel-space. In addition, note the similarities between Haswell and Haswell-EP. The number of L3 cache misses converges to approximately 0.375 cache misses per element for $n \sim 3 \cdot 10^8$ for both machines as they arise from the L2 cache. As a matter of fact, when $q = 4096$ the size of the buckets ranges between $5 \cdot 10^4$ and $7.5 \cdot 10^4$, for $2 \cdot 10^8 \leq n \leq 3 \cdot 10^8$, and the L2 cache gets filled in such interval. The result for L2 cache misses show some interesting behavior as well. The number of cache misses for Direct and Permuting 5.2 is approximate with high precision. However, Permuting 5.1 diverges from Cachegrind measurements by approximately 0.5 additional cache misses per element. In addition, the raw data collected from such experiment showed significant oscil-



(a) (b)

Figure 5.8: Raw data of L2 (a) and L3 (b) cache misses from permuting using the TRACE, SYNC and HYBRID data structures compared to DIRECT on Haswell-EP with a random generated permutation with parameters $P = 6$ and $q = 4096$. Each x-coordinate collects k repetitions of the same algorithm, with $k = 51$ for Permuting 5.1 and Permuting 5.2 and $k = 11$ for DIRECT. The reference plots (CG) are obtained by simulating the same algorithms using Cachegrind, see Table 5.4 and Table 5.5.

lations for the L2 cache misses of Permuting 5.1. This behavior is canceled whenever we consider single thread computations and the data overlaps with the one from Permuting 5.2. We conjecture that our data structure is sensitive to false sharing during the process of computing the intermediate positions of the records. Such behavior is not present in Permuting 5.2 as the data structure $\mathcal{S}$ induces a strong thread synchronization. Conversely, thread synchronization for Permuting 5.1 is managed by the OpenMP directive `schedule(static,segment)`. However, switching from the compiler optimization -03 to a less aggressive one, i.e. -01, cancels the problem while it occurs again for -02. We claim that the set of instructions included in the -02 optimization forces the compiler to optimize thread scheduling thus enhancing false sharing.

The process of investigating the -02 optimization in order to identify the instructions that causes performance degradation in the L2 cache is far for being trivial if not unfeasible. The -02 optimization enables almost ninety flags, fifty more than -01. While we attempted to identify such flag with a trial and error approach, the results were inconclusive and we conjecture that such behavior is caused by a combination of optimization flags. Nevertheless, we highlight that such performance degradation affect only partially running times.

5.8 Conclusions

The studies conducted on permuting have confirmed the motivation for designing External Memory algorithms. Permuting using data structures and sorting-based algorithms ensures a reduction in the number of I/O operations. In particular, exploiting data structures allow us to provide I/O optimal algorithms that are efficient in practice. Experiments show that, even if the level of parallelism changes, the data structures achieve satisfactory performances. Nevertheless, the difference in the architectures is not clean-cut and it is not know what level of parallelism such data structures can achieve on massively parallel systems. As a matter of fact, our data structures are architecture dependent as they rely on number of cores and memory size. Hence, we formalize the problem of designing data structures that are architecture independent as well as optimal for different levels of parallelism.

Sorting-based algorithms in parallel systems introduce synchronization issues that are commonly solved via atomic operations. Data structures overcome this issue by exploiting index tables that grow considerably as the input size increases. Data structures initialization is, indeed, time consuming and, in order to amortize the preprocessing time, the applications of such data structures are restricted to scenarios where the permutation is applied repeatedly. We leave open the question whether the trade off between time and space for process synchronization is still well established or algorithms that permute I/O optimally on the fly can achieve high performance in practice.

The problem of permuting is oriented entirely to memory transfers, hence, it does not reflect the structure of standard numerical problems, e.g. matrix computations. We formalize the question whether we can achieve the same performance for tasks that require intensive computation, such as matrix multiplication algorithms. While the algorithms in this chapter aim at shedding some light on external memory models and the relation between algorithms optimal in theory and in practice, our future work will extend the focus on architecture independent algorithms for intensive computation problems.

Chapter 6

Conclusions

This chapter aims at collecting the contributions and the future research directions highlighted in each chapter. For each research output, we give a brief overview of the contributions and we introduce the reader to open problems that are addressed through a detailed discussion.

Chapter 2

In Chapter 2 we presented new randomized algorithms for sparse matrix multiplication. Specifically, we presented new data structures that extend the algorithm from Williams and Yu [WY14] to the sparse input case. Our analysis covers the Word RAM model, the Cache Oblivious model and the Parallel External Memory model.

The algorithms presented do not exploited any structure of the input/output matrices, such as number of nonzero entries per column or balanced distribution of the nonzero elements. As addressed in Chapter 4, balanced assumptions yield improved bounds for sparse matrix multiplication. It is natural to believe that, by exploiting the anatomy of the matrices, one may derive improved bounds for sparse matrix multiplication, e.g., in the logarithmic factors.

Open Problem 1. *Can we exploit the structure of the input/output matrices in order to improve the bounds of Chapter 2?*

In the Cache Oblivious model we are able to multiply matrices with no knowledge of M and B during the computation. Conversely, Pagh and Stöckel [PS14] provided an optimal cache aware algorithm for multiplying sparse matrices that uses blocking arguments and matrix size

estimations. The limitations of the Cache Oblivious are well understood. However, we address the issue of designing improved algorithms in oblivious models. The latter are commonly a non trivial framework for parallel algorithms, given the concurrency issues that arise from concurrent cache oblivious models. Hence, an interesting research direction is to study more in depth parallel, cache oblivious models.

Open Problem 2. *Can we obtain bounds similar to [PS14] in the Cache Oblivious model? Can we extend the bounds obtained in the Parallel External Memory model to a concurrent cache oblivious model?*

The algorithms presented scale efficiently to multiple processors. The Cache Oblivious algorithm of Chapter 2 can be easily derived from Parallel External Memory analysis by setting $P = 1$. Nevertheless, for parallel, cache aware models, the bounds one wants to achieve are of the form $\tilde{\mathcal{O}}((h/PB)\min\{\sqrt{k}/\sqrt{M}, h/M\})$ which follow straightforwardly from [PS14].

Open Problem 3. *Can we extend Pagh and Stöckel's techniques to the Parallel External Memory model, to obtain a bound similar to [PS14], with matching lower bound?*

Chapter 3

In Chapter 3 we presented new algorithms for sparse Boolean matrix multiplication. Our bounds are asymptotically worse than the state of the art. Nevertheless, our algorithms are designed with the purpose of practical efficiency. As a matter of fact, we use only estimation algorithms as subroutines.

The partitioning techniques used by our algorithms seem to be relatively standard as they appear similarly in Van Gucht et al. [VG$^+$15] and Lingas [Lin09]. Nonetheless, Amossen et al. [AP09] exploit improved partitioning techniques for isolating submatrices which are computed using fast matrix multiplication. This technique does not apply directly to our framework.

Open Problem 4. *Can we use estimators differently, e.g. by detecting zeros in highly dense submatrices, and obtain bounds similar to [AP09], e.g.* $\mathcal{O}(h + h^{2/3}k^{2/3} + k)$*?*

The algorithms presented in this chapter are Atlantic City, i.e. time, I/Os and output are random variables. On the front of derandomization, we would like to reduce the amount of randomness during the execution by having deterministic output results. The main hurdle for this problem is to provide deterministic algorithms for detecting nonzero entries in highly sparse submatrices.

Open Problem 5. *Can we design full Las Vegas algorithms, where only running time and number of I/Os are random variables while the output is always correct?*

The random permutation, when applied to input matrices, may not spread the nonzero entries of the output uniformly across the matrix. Dense submatrices with more than $\sqrt{k}$ nonzero entries worsen our time bounds. Our analysis guarantees either a polynomially or exponentially decreasing probability of having highly dense submatrices. An interesting future direction is to partially derandomize the permutation process to deterministically obtain sparse submatrices.

Open Problem 6. *Can we achieve a partial derandomization by designing deterministic algorithms to partition the output matrix in submatrices with* $\mathcal{O}(1)$ *nonzero entries?*

The restriction on submatrices such that they do not contain entries on the same row/column, allows us to prove exponentially decreasing probability of having dense submatrices. In the general case, we proved only a polynomially decreasing probability. For our polynomial tail bounds, the chances of deviating from the expected overall time and I/O bounds are still undesirable.

Open Problem 7. *Can we prove stronger tail inequalities, e.g. exponential, for the general case?*

The framework we used for computing the Boolean matrix product can be extended straightforwardly to matrices with entries from a field by simply computing k inner products using kn additional operations. Unfortunately, cancellation of terms is not supported by the estimation algorithm from [ACP10] which is designed specifically for semiring algebras.

Open Problem 8. *Can we extend the size estimator from [ACP10] to allow cancellation of terms and use it as a subroutine for sparse matrix multiplication algorithms?*

Chapter 4

In Chapter 4 we investigated how exploiting compression techniques from [JS15], combined with the combinatorial framework of [YZ05, AP09] yields improved bounds for sparse matrix multiplication.

Motivated by the studies of [JS15] on balanced output matrices, we showed that, exploiting the balanced structure of product matrices, yield asymptotically better bounds compared to the state of the art and to the unbalanced case. When no information on the output matrix is available, we are not able to replicate the same time bounds and we improve over the state of the art only for a specific parameter range.

Open Problem 9. *Can we design algorithms for the unbalanced case that achieve the same time bounds of the balanced one?*

An alternative research direction is to prove lower bounds either on the combinatorial framework of [YZ05, AP09] or on the compression technique of [JS15]. The motivation for this line of work is given by the marginal improvements gained by our randomized algorithms over [AP09] that does not suggest significant further improvements.

Open Problem 10. *Can we prove lower bounds either on the combinatorial framework of [YZ05, AP09] or on the compression techniques of [JS15]?*

The literature is lacking with parallel algorithms for fast matrix multiplication This leaves little room for the distribution step as each processor can only fast-multiply batches of submatrices. For blocked-based rectangular fast matrix multiplication, this seems doable. For actual rectangular fast matrix multiplication à la Coppersmith [Cop97] this seems to require major efforts. Similarly, in the Cache Oblivious model, one would need to find partitioning techniques, similar to the block-based of Lemma 4.4, oblivious to memory and block size.

Open Problem 11. *Can we extend the analysis to the Parallel External Memory model and to the Cache Oblivious model?*

Algorithms that exploit fast matrix multiplication as a kernel are able to outperform almost every present matrix multiplication algorithms. The impracticality of fast matrix multiplication makes these algorithms of theoretical interest only.

Open Problem 12. *Can we design algorithms matching the bounds of, e.g., [AP09, Dus18b] that do not rely on fast matrix multiplication subroutines?*

Chapter 5

In Chapter 5 we studied empirically the problem of permuting in the Parallel External Memory model. We provide sorting based data structures and efficient algorithm implementation for permuting elements on multicore architectures.

The data structures rely on parameters defined by the hardware architecture. That is, we adapt our algorithms to memory structure and level of parallelism to obtain high performance in the permutation process. It is clear that, our data structures are architecture dependent. Nevertheless, the difference in the architectures we tested is not clean-cut and it is not know what level of parallelism such data structures can achieve on massively parallel systems.

Open Problem 13. *Can we designing data structures for the problem of permuting that are architecture independent as well as efficient for different levels of parallelism?*

Algorithms in parallel systems introduce synchronization issues that are commonly solved via atomic operations. The algorithms in Chapter 5 are no exception and our data structures overcome this issue by exploiting synchronization tables that grow considerably as the input size increases. Data structures initialization is the most time consuming phase and, in order to amortize the preprocessing time, the applications of such data structures are restricted to scenarios where the permutation is applied repeatedly.

Open Problem 14. *Is the trade off between time and space for process synchronization still well established? Can we achieve high performance in practice with algorithms that permute I/O optimally on the fly?*

The problem of permuting is an I/O bounded problem, rather than a CPU bounded one. As a consequence, it does not reflect the structure of standard numerical problems, e.g. matrix computations, and we expect further improvements for sorting-based algorithms applied to more computational demanding problems.

Open Problem 15. *Can we achieve high performance by using sorting-based data structures in more intensive computational problems, such as matrix multiplication?*

Bibliography

[ACP10] Rasmus Resen Amossen, Andrea Campagna, and Rasmus Pagh. Better Size Estimation for Sparse Matrix Products. In *Approx-Random*, pages 406–419. Springer, 2010.

[ADFK70] V.L. Arlazarov, E.A. Dinic, I.A. Faradzev, and A. Kronrod. On Economical Construction of Transitive Closure of an Oriented Graph. *Doklady Akademii Nauk SSSR*, 194(3):487–88, 1970.

[AFLG15] Andris Ambainis, Yuval Filmus, and François Le Gall. Fast Matrix Multiplication: Limitations of the Coppersmith-Winograd Method. In *Proceedings of the 47th annual ACM Symposium on Theory of Computing*, pages 585–593. ACM, 2015.

[AGNS08] Lars Arge, Michael T Goodrich, Michael Nelson, and Nodari Sitchinava. Fundamental Parallel Algorithms for Private-Cache Chip Multiprocessors. In *Proceedings of the 20th Annual Symposium on Parallelism in Algorithms and Architectures*, pages 197–206. ACM, 2008.

[AP09] Rasmus Resen Amossen and Rasmus Pagh. Faster Join-projects and Sparse Matrix Multiplications. In *Proceedings of the 12th International Conference on Database Theory*, pages 121–126. ACM, 2009.

[ASU13] Noga Alon, Amir Shpilka, and Christopher Umans. On Sunflowers and Matrix Multiplication. *Computational Complexity*, 22(2):219–243, 2013.

[AV88] Alok Aggarwal and Jeffrey Vitter. The Input/Output Complexity of Sorting and Related Problems. *Communications of the ACM*, 31(9):1116–1127, 1988.

[AW14] Amir Abboud and Virginia Vassilevska Williams. Popular Conjectures Imply Strong Lower Bounds for Dynamic Problems. In *Proceedings of the 55th Annual Symposium on Foundations of Computer Science*, pages 434–443. IEEE, 2014.

[BBF+10] Michael A Bender, Gerth Stølting Brodal, Rolf Fagerberg, Riko Jacob, and Elias Vicari. Optimal sparse matrix dense vector multiplication in the I/O-model. *Theory of Computing Systems*, 47(4):934–962, 2010.

[BCRL79] Dario Bini, Milvio Capovani, Francesco Romani, and Grazia Lotti. $\mathcal{O}(n^{2.7799})$ Complexity for $n \times n$ Approximate Matrix Multiplication. *Information processing letters*, 8(5):234–235, 1979.

[BF02] Gerth Stølting Brodal and Rolf Fagerberg. Cache Oblivious Distribution Sweeping. In *International Colloquium on Automata, Languages, and Programming*, pages 426–438. Springer, 2002.

[BF03] Gerth Stølting Brodal and Rolf Fagerberg. On the Limits of Cache-Obliviousness. In *Proceedings of the 35th Annual ACM Symposium on Theory of Computing*, pages 307–315. ACM, 2003.

[BFGK05] Michael A Bender, Jeremy T Fineman, Seth Gilbert, and Bradley C Kuszmaul. Concurrent Cache-Oblivious B-trees. In *Proceedings of the 17th annual ACM Symposium on Parallelism in Algorithms and Architectures*, pages 228–237. ACM, 2005.

[BG12] Aydin Buluç and John R Gilbert. Parallel Sparse Matrix-Matrix Multiplication and Indexing: Implementation and Experiments. *SIAM Journal on Scientific Computing*, 34(4):C170–C191, 2012.

[Blä99] Markus Bläser. A $\frac{5}{2}n^2$-Lower Bound for the Rank of $n \times n$-Matrix Multiplication over Arbitrary Fields. In *Proceedings*

of the 40th Annual Symposium on Foundations of Computer Science, pages 45–50. IEEE, 1999.

[Blä01] Markus Bläser. A $\frac{5}{2}n^2$-Lower Bound for the Multiplicative Complexity of $n \times n$-Matrix Multiplication. In *Annual Symposium on Theoretical Aspects of Computer Science*, pages 99–109. Springer, 2001.

[Ble90] Guy E. Blelloch. Prefix Sums and Their Applications. Technical report, Synthesis of Parallel Algorithms, 1990.

[BW09] Nikhil Bansal and Ryan Williams. Regularity Lemmas and Combinatorial Algorithms. In *Proceedings of the 50th Annual Symposium on Foundations of Computer Science*, pages 745–754. IEEE, 2009.

[CG86] Bernard Chazelle and Leonidas J Guibas. Fractional Cascading: I. A Data Structuring Technique. *Algorithmica*, 1(1):133–162, 1986.

[Cha15] Timothy M Chan. Speeding Up the Four Russians Algorithm by about One More Logarithmic Factor. In *Proceedings of the 26th annual ACM-SIAM symposium on Discrete algorithms*, pages 212–217. Society for Industrial and Applied Mathematics, 2015.

[Cop97] Don Coppersmith. Rectangular Matrix Multiplication Revisited. *Journal of Complexity*, 13(1):42–49, 1997.

[CU03] Henry Cohn and Christopher Umans. A Group-Theoretic Approach to Fast Matrix Multiplication. In *Proceedings of the 44th Annual Symposium on Foundations of Computer Science*, pages 438–449. IEEE, 2003.

[CW82] Don Coppersmith and Shmuel Winograd. On the Asymptotic Complexity of Matrix Multiplication. *SIAM Journal on Computing*, 11(3):472–492, 1982.

[CW87] Don Coppersmith and Shmuel Winograd. Matrix Multiplication via Arithmetic Progressions. In *Proceedings of the nineteenth annual ACM symposium on Theory of computing*, pages 1–6. ACM, 1987.

[DGH15] Erik D Demaine, Vineet Gopal, and William Hasenplaugh. Cache-Oblivious Iterated Predecessor Queries via Range Coalescing. In *Workshop on Algorithms and Data Structures*, pages 249–262. Springer, 2015.

[DJ17] Matteo Dusefante and Riko Jacob. An Empirical Evaluation of Permuting in Parallel External Memory. Manuscript, 2017.

[DJ18] Matteo Dusefante and Riko Jacob. Cache Oblivious Sparse Matrix Multiplication. In *Latin American Symposium on Theoretical Informatics*, pages 437–447. Springer, 2018.

[DP09] Devdatt P. Dubhashi and Alessandro Panconesi. *Concentration of Measure for the Analysis of Randomized Algorithms*. Cambridge University Press, 2009.

[Dus18a] Matteo Dusefante. Atlantic City Boolean Matrix Multiplication. Manuscript, 2018.

[Dus18b] Matteo Dusefante. Compressed Sparse Matrix Multiplication. Manuscript, 2018.

[ER60] Paul Erdős and Richard Rado. Intersection Theorems for Systems of Sets. *Journal of the London Mathematical Society*, 1(1):85–90, 1960.

[FLPR99] Matteo Frigo, Charles E Leiserson, Harald Prokop, and Sridhar Ramachandran. Cache-oblivious Algorithms. In *Proceeding of the 40th Annual Symposium on Foundations of Computer Science*, pages 285–297. IEEE, 1999.

[Fre77] Rusins Freivalds. Probabilistic Machines Can Use Less Running Time. In *IFIP congress*, volume 839, page 842, 1977.

[Fur70] ME Furman. Application of a method of rapid multiplication of matrices to the problem of finding the transitive closure of a graph. In *Doklady Akademii Nauk*, volume 194, pages 524–524. Russian Academy of Sciences, 1970.

[FW90] Michael L Fredman and Dan E Willard. Blasting Through the Information Theoretic Barrier with Fusion Trees. In *Proceedings of the twenty-second annual ACM symposium on Theory of Computing*, pages 1–7. ACM, 1990.

[GJ+10] Gaël Guennebaud, Benoît Jacob, et al. Eigen v3. http://eigen.tuxfamily.org, 2010.

[GMM16] Daniel Gruss, Clémentine Maurice, and Stefan Mangard. Rowhammer. js: A remote software-induced fault attack in javascript. In *Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 300–321. Springer, 2016.

[Gre12] Gero Greiner. *Sparse Matrix Computations and their I/O Complexity*. PhD thesis, Dissertation, Technische Universität München, München, 2012.

[Guo11] Yang Guo. Tuning Funnelsort for Permuting. Master's thesis, Fakultät für Informatik der Technischen Universität München, 2011.

[HP98] Xiaohan Huang and Victor Y Pan. Fast Rectangular Matrix Multiplication and Applications. *Journal of complexity*, 14(2):257–299, 1998.

[IPZ01] Russell Impagliazzo, Ramamohan Paturi, and Francis Zane. Which Problems Have Strongly Exponential Complexity? *Journal of Computer and System Sciences*, 63(4):512–530, 2001.

[IS09] Mark A Iwen and Craig V Spencer. A Note on Compressed Sensing and the Complexity of Matrix Multiplication. *Information Processing Letters*, 109(10):468–471, 2009.

[JS15] Riko Jacob and Morten Stöckel. Fast Output-Sensitive Matrix Multiplication. In *European Symposium on Algorithms*, pages 766–778. Springer, 2015.

[JWK81] Hong Jia-Wei and Hsiang-Tsung Kung. I/O Complexity: The Red-Blue Pebble Game. In *Proceedings of the 13th Annual ACM Symposium on Theory of Computing*, pages 326–333. ACM, 1981.

[Lan14] Joseph M Landsberg. New Lower Bounds for the Rank of Matrix Multiplication. *SIAM Journal on Computing*, 43(1):144–149, 2014.

[Lev09] David Levinthal. Performance Analysis Guide for Intel® Core™ i7 Processor and Intel® Xeon™ 5500 processors. *Intel Performance Analysis Guide*, 30:18, 2009.

[LG14] François Le Gall. Powers of Tensors and Fast Matrix Multiplication. In *Proceedings of the 39th International Symposium on Symbolic and Algebraic Computation*, pages 296–303. ACM, 2014.

[Lin09] Andrzej Lingas. A Fast Output-Sensitive Algorithm for Boolean Matrix Multiplication. In *European Symposium on Algorithms*, pages 408–419. Springer, 2009.

[LK14] Jure Leskovec and Andrej Krevl. SNAP Datasets: Stanford Large Network Dataset Collection. `http://snap.stanford.edu/data`, June 2014.

[LLDM09] Jure Leskovec, Kevin J Lang, Anirban Dasgupta, and Michael W Mahoney. Community Structure in Large Networks: Natural Cluster Sizes and the Absence of Large Well-Defined Clusters. *Internet Mathematics*, 6(1):29–123, 2009.

[LM12] Jure Leskovec and Julian J Mcauley. Learning to Discover Social Circles in Ego Networks. In *Advances in neural information processing systems*, pages 539–547, 2012.

[Mol02] Richard A Mollin. *RSA and Public-Key Cryptography*. CRC Press, 2002.

[MR13] Alex Massarenti and Emanuele Raviolo. The Rank of $n \times n$ Matrix Multiplication is at least $3n^2 - 2\sqrt{2}n^{\frac{3}{2}} - 3n$. *Linear Algebra and its Applications*, 438(11):4500–4509, 2013.

[MS+96] Larry W McVoy, Carl Staelin, et al. LMbench: Portable Tools for Performance Analysis. In *USENIX Annual Technical Conference*, pages 279–294. San Diego, CA, USA, 1996.

[MS12] Larry W McVoy and Carl Staelin. LMbench - Tools for Performance Analysis, 2012.

[MU05] Michael Mitzenmacher and Eli Upfal. *Probability and computing: Randomized algorithms and probabilistic analysis*. Cambridge university press, 2005.

[Mun71] Ian Munro. Efficient determination of the transitive closure of a directed graph. *Information Processing Letters*, 1(2):56–58, 1971.

[NMAB18] Yusuke Nagasaka, Satoshi Matsuoka, Ariful Azad, and Aydın Buluç. High-Performance Sparse Matrix-Matrix Products on Intel KNL and Multicore Architectures. *arXiv preprint arXiv:1804.01698*, 2018.

[NV93] Mark H Nodine and Jeffrey Scott Vitter. Deterministic Distribution Sort in Shared and Distributed Memory Multiprocessors. In *Proceedings of the 5th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 120–129. ACM, 1993.

[OO73] Patrick O'Neil and Elizabeth O'Neil. A Fast Expected Time Algorithm for Boolean Matrix Multiplication and Transitive Closure. *Information and Control*, 22(2):132–138, 1973.

[Pag12] Rasmus Pagh. Compressed Matrix Multiplication. In *Proceedings of the 3rd Innovations in Theoretical Computer Science Conference*, pages 442–451. ACM, 2012.

[Pan78] V Ya Pan. Strassen's Algorithm Is not Optimal: Trililnear Technique of Aggregating, Uniting and Canceling for Constructing Fast Algorithms for Matrix Operations. In *Proceedings of the 19th Annual Symposium on Foundations of Computer Science*, pages 166–176. IEEE, 1978.

[Pan81] V Ya Pan. New Combinations of Methods for the Acceleration of Matrix Multiplications. *Computers & Mathematics with Applications*, 7(1):73–125, 1981.

[PS14] Rasmus Pagh and Morten Stöckel. The Input/Output Complexity of Sparse Matrix Multiplication. In *European Symposium on Algorithms*, pages 750–761. Springer, 2014.

[Raz02] Ran Raz. On the Complexity of Matrix Product. In *Proceedings of the 34th Annual ACM Symposium on Theory of Computing*, pages 144–151. ACM, 2002.

[Rob05] Sara Robinson. Toward an Optimal Algorithm for Matrix Multiplication. *SIAM news*, 38(9):1–3, 2005.

[Rom82] Francesco Romani. Some Properties of Disjoint Sums of Tensors Related to Matrix Multiplication. *SIAM Journal on Computing*, 11(2):263–267, 1982.

[Sch81] Arnold Schönhage. Partial and Total Matrix Multiplication. *SIAM Journal on Computing*, 10(3):434–455, 1981.

[Sch11] Boris Schling. *The Boost C++ Libraries*. XML Press, 2011.

[Sei95] Raimund Seidel. On the All-Pairs-Shortest-Path Problem in Unweighted Undirected Graphs. *Journal of Computer and System Sciences*, 51(3):400–403, 1995.

[Sha78] Michael Ian Shamos. *Computational Geometry*. PhD thesis, New Haven, CT, USA, 1978. AAI7819047.

[Sto10] Andrew James Stothers. *On the complexity of matrix multiplication*. PhD thesis, 2010.

[Str69] Volker Strassen. Gaussian Elimination is not Optimal. *Numerische mathematik*, 13(4):354–356, 1969.

[Str86] Volker Strassen. The Asymptotic Spectrum of Tensors and the Exponent of Matrix Multiplication. In *Proceedings of the 27th Annual Symposium on Foundations of Computer Science*, pages 49–54. IEEE, 1986.

[SZ99] Avi Shoshan and Uri Zwick. All Pairs Shortest Paths in Undirected Graphs with Integer Weights. In *Proceeding of the 40th Annual Symposium on Foundations of Computer Science*, pages 605–614. IEEE, 1999.

[VD03] Richard Wilson Vuduc and James W Demmel. *Automatic performance tuning of sparse matrix kernels*, volume 1. University of California, Berkeley, 2003.

[VG+15] Dirk Van Gucht et al. The Communication Complexity of Distributed Set-Joins with Applications to Matrix Multiplication. In *Proceedings of the 34th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*, pages 199–212. ACM, 2015.

[VS94] Jeffrey Scott Vitter and Elizabeth AM Shriver. Algorithms for Parallel Memory, I: Two-Level Memories. *Algorithmica*, 12(2-3):110–147, 1994.

[WD10] Vince Weaver and Jack Dongarra. Can Hardware Performance Counters Produce Expected, Deterministic Results. In *Proceedings of the 3rd Workshop on Functionality of Hardware Performance Monitoring*, 2010.

[Wil12] Virginia Vassilevska Williams. Multiplying Matrices Faster Than Coppersmith-Winograd. In *Proceedings of the 44th annual ACM Symposium on Theory of Computing*, pages 887–898. ACM, 2012.

[WTM13] Vincent M Weaver, Dan Terpstra, and Shirley Moore. Non-Determinism and Overcount on Modern Hardware Performance Counter Implementations. In *International Symposium on Performance Analysis of Systems and Software*, pages 215–224. IEEE, 2013.

[WW10] Virginia Vassilevska Williams and Ryan Williams. Subcubic Equivalences between Path, Matrix and Triangle Problems. In *Proceedings of the 51st Annual Symposium on Foundations of Computer Science*, pages 645–654. IEEE, 2010.

[WY14] Ryan Williams and Huacheng Yu. Finding Orthogonal Vectors in Discrete Structures. In *Proceedings of the 25th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 1867–1877. SIAM, 2014.

[Yu15] Huacheng Yu. An Improved Combinatorial Algorithm for Boolean Matrix Multiplication. In *International Colloquium on Automata, Languages, and Programming*, pages 1094–1105. Springer, 2015.

[YZ05] Raphael Yuster and Uri Zwick. Fast Sparse Matrix Multiplication. *ACM Transactions on Algorithms*, 1(1):2–13, 2005.

[Zwi02] Uri Zwick. All Pairs Shortest Paths using Bridging Sets and Rectangular Matrix Multiplication. *Journal of the ACM*, 49(3):289–317, 2002.