Technische Universität München
Fakultät für Informatik

Lehrstuhl für Effiziente Algorithmen

Sparse Matrix Computations and their I/O Complexity

Gero Greiner

Vollständiger Abdruck der von der Fakultät für Informatik der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften (Dr. rer. nat.)

genehmigten Dissertation.

Vorsitzender:		Univ.-Prof. Dr. Helmut Krcmar
Prüfer der Dissertation:		
	1.	TUM Junior Fellow Dr. Riko Jacob
	2.	Univ.-Prof. Dr. Ernst W. Mayr
	3.	Univ.-Prof. Dr. Dr. h.c. Javier Esparza

Die Dissertation wurde am 7. August 2012 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 10. November 2012 angenommen.

Abstract

For many programs in high performance computing, but also in every day computational tasks, the performance bottleneck is caused by memory accesses instead of CPU time. To tackle this problem in a theoretical way, the I/O-model (external memory model) was introduced which models a fast internal memory (cache) of limited size where computations are performed, and an infinite external memory (disk) organised in blocks. The number of block transfers between the two memory layers is called the I/O complexity. Recently, the parallel external memory model (PEM) was introduced to model a parallel structure consisting of multiple processors with caches and a shared disk. Tasks including sparse matrices often induce a very irregular memory access pattern leading to a bad I/O behaviour. A prominent non-obvious example related to sparse matrices is the MapReduce framework where intermediate results produced by Map can be seen as a sparse matrix that is transposed in the so called shuffle step.

This thesis considers the parallel I/O complexity of several tasks involving sparse matrices over a semiring: the multiplication of a sparse matrix with either multiple vectors, dense matrices or sparse matrices; creating the bilinear form of two vectors defined by a sparse matrix; the transposition of a sparse matrix. For all these tasks and all meaningful choices of dimensions, sparsity, memory size and block size, we determine the I/O complexity, i.e. we present lower bounds and algorithms that mostly differ only by a constant factor in the number of I/Os. Solely for the task of multiplying two sparse matrices, upper and lower bounds become tight only for very sparse matrices.

All the lower bounds are based on counting and can only show existence of worst-case instances but are not constructive. This leads to the question of identifying the difficulty of an instance. In this regard, we refute a natural conjecture regarding the expansion of an associated graph. Furthermore, we present a substantially simpler proof for the low I/O complexity of BMMC-permutations that gives an intuitive explanation for the easiness of these permutations.

The work is complemented by a consideration of the MapReduce framework. This consideration yields further insights on the I/O complexity of a task modelled in Map Reduce, and it strengthens the theoretical foundation of MapReduce by creating a comparison of the MapReduce model and the (parallel) external memory model.

Acknowledgment

It is often said in the acknowledgment that writing the thesis would have been absolutely impossible without the permanent help of others. Although this can sound slightly exaggerated, I think it is indeed true.

First of all and most importantly, I would like to thank my advisor Riko Jacob for his incredible support, his invaluable insights and ideas. He always found (or took) the time to discuss my problems and to give me helpful advice. I am deeply indebted to my former teacher Ingo Wegener for introducing me into the fields of complexity theory and efficient algorithms. Furthermore, my gratitude goes to Ernst W. Mayr for integrating us into his friendly chair. My dear colleagues Johannes Krugel, Tobias Lieber, Harald Räcke, Hanjo Täubig and Jeremias Weihmann deserve special thanks for reading and correcting drafts of this thesis. Moreover, I want to thank all the members of the *Chair for Efficient Algorithms* in Munich and the *Chair of Algorithms, Data Structures and Applications* in Zurich.

I am thankful for all the music and the nice hours and journeys with all my bandmates from *Panda in the Icebox*, *Gr8balls* and the *Express Brass Band*. Special thanks go to Neil Vaggers for his comments on my English. My flatmates and friends made the time to write this thesis an awesome experience.

Finally, my parents and my sister were of invaluable importance for this thesis, beginning with the pure existence. They always supported me in what I did, and what I am doing. At the very end, I would like to encourage my sister in writing her thesis.

Contents

1 Introduction

1.1 Motivation

Computing matrix products is a fundamental operation in many computational tasks. The broad field of matrix computation has a long tradition dating back over several centuries with first appearances even being found to be over 1500 years old. This shows its importance and it explains its strong linkage to other fields. The main focus of this thesis lies on the consideration of sparse matrix products where most of the matrix entries are zero. More precisely, we examine communication required for multiplying a sparse matrix with either one or several dense vectors, with a dense matrix, or with another sparse matrix.

Sparse matrix products are strongly involved in many areas, like scientific computing and engineering. Applications arise not only in linear systems, eigenvalue problems and least squares problems, but also in data mining and web search. In many cases, one sparse matrix is multiplied repeatedly with several vectors, thus justifying a preprocessing phase on the data structures. The insights gained are not only specific to the mathematical background, but can also be applied to describe communication in tasks like MapReduce [DG04].

Matrix multiplication is a deeply studied area in terms of computation involving a vast amount of research. Despite this, the computational complexity is still not settled, and it is not clear whether $\mathcal{O}\left(n^2\right)$ operations are sufficient to multiply two dense $n \times n$ matrices. It was not until 1969 that computational methods which require less than the trivial n^3 multiplications to compute the product of two $n \times n$ matrices were revealed. In his seminal paper [Str69], Strassen presented an algorithm which computes the matrix product with $\mathcal{O}\left(n^{\log_2 7}\right)$ arithmetic operations. Ever since, the algorithmic complexity has been decreased [Pan78, BCRL79, Sch81, CW90, Vas11] leading to $\mathcal{O}\left(n^{\omega}\right)$ for $\omega < 2.373$ recently. We survey this history briefly in Section 1.4.1. However, the best known upper bound has not been changed significantly for more than two decades, apart from recent rather small reductions of ω that rely on the same techniques as previously. On the other side, Bläser showed in [Blä99] that $\frac{5}{2}n^2 - 3n$ multiplications are required for matrix multiplication. For the multiplication of binary matrices over GF(2), Shpilka presented a lower bound of $3n^2 - o\left(n^2\right)$ on the number of multiplications in [Shp01]. He also slightly improves the bound of Bläser for finite fields in the factor of n^2. However, if the size of the field approaching infinity, the factor goes to $\frac{5}{2}$ like in [Blä99].

On the contrary, for sparse matrix dense vector multiplication, the trivial lower bound of accessing each of the matrix entries at least once is asymptotically matched by the direct algorithm of creating each elementary product explicitly. This leads to a computational complexity of $\Theta\left(H\right)$ for a sparse matrix with H non-zero entries.[1] When multiplying a sparse matrix with another matrix, tools like the Strassen algorithm [Str69] or the Coppersmith-Winograd algorithm [CW90] can become useful. However, to the best of the author's knowledge, up until now, no computational techniques to exploit sparsity of a matrix are known in this context. The known fast matrix multiplication algorithms (i.e. those that operate in time $o\left(n^3\right)$) all depend on a recursive structure where in each recursion step the density increases – leading in many cases to the same number of multiplications as dense matrix products (see Section 1.4 for details). For evenly distributed non-zero entries in both matrices, the increase of density in the Strassen algorithm was studied in [MS04]. Yuster and Zwick [YZ04] presented a combined approach that uses a fast matrix multiplication algorithm for denser parts of the product while applying the direct algorithm for the other part. Improving on the costs induced by the direct algorithm hence also improves on the costs of their approach. We also note that, depending on the sparsity of the multiplied matrices, a direct algorithm can still be faster.

In this thesis, we consider classes of algorithms that compute each elementary product explicitly at least once throughout the computation process. To this end, we restrict ourselves to computations over an arbitrary semiring where inverse elements are neither guaranteed for multiplication nor addition. The reader shall be aware that we will not tackle the problem

[1]Note that allowing a preprocessing step depending on the matrix, the multiplication process can be reduced [Wil07].

of a general lower bound for matrix multiplication.

For the naïve implementation of simply creating the required elementary products one after another in any given order, it has been observed that the CPU-usage is even less than 10% [JS10, Vud03]. This relies upon the design of current memory architectures, and the fact that sparse matrices can induce a totally irregular access pattern in memory. Tackling the problem of irregular accesses and stating complexity results for memory accesses will form the main focus of this work. Thus, let us consider today's memory hierarchy in the following pages. With this understanding, we derive some computational models culminating in the PEM model, which is mostly considered throughout this thesis.

It is a well-known fact that in comparison to the evolution of CPU clock speed, memory access time has always lagged behind. The famous Moore's law predicts an exponential growth of transistors density on a CPU die. It has been observed that the number of transistors on a chip have doubled about every 18-24 months. Until recent years, this improvement of transistors per space proportionally led to a doubling of CPU speed in about the same period. Despite its permanent further development, RAM access time could not catch up with this evolution. One reason is that, additionally to access time, a main focus of memory design is capacity. This forms a limiting factor in that high capacity requires more space, and memory cells need to be further away from the CPU. By the laws of physics, a higher latency follows immediately. Even though the increase of CPU speed has recently slowed down, Moore's law still holds on, but trends have changed towards providing multiple cores on a single chip.

This historical progress has led to a performance gap between CPU clock speed and memory access time of several orders of magnitude. At the time of writing this thesis, a typical working machine performed a CPU cycle within $3 \cdot 10^{-10}s$. Access times of solid-state-drives were in the order of $10^{-4}s$ and even working memory access time still laid roughly a factor 100 away from CPU cycles. On the other hand, multi-core architectures require the simultaneous access of data in order for the cores to compute (independently) in parallel.

Luckily, it can be observed that many programs reveal some type of locality in their induced memory accesses. In general, two types of locality are distinguished: **Temporal locality** refers to multiple accesses to the same data within a short time period. A program is said to have **spacial locality** if it is likely to access data in the close neighbourhood of an earlier access.

Temporal locality can be exploited using caches. This reflects current hardware design where it is common sense to have a small but very fast cache attached to the CPU. Spacial locality leads to the idea of memory transfer in blocks. If a single date is required for computation, not only the date but its neighbourhood up to some specific size is transferred into the cache. This behaviour can be found in so called cache-lines that are loaded simultaneously. The use of caches can thus take advantage of locality by simulating a fast, large memory. This is even brought further in that a hierarchy of memory layers is part of today's computer systems. The L1 cache, a small cache built of the fastest, most expensive memory cells, is located closest to the CPU. In order to provide fast access, these cells have to be wired in short distance to the CPU which, apart from its cost, is another limiting factor of the size of L1 cache. The L1 cache is followed by the L2 cache, and with growing consensus by another L3 cache – all being built on the CPU die. The most commonly known layers (outside the CPU) are the working memory aka RAM and a permanent mass storage device such as magnetic hard drives or solid-state drives. Since the access time of each layer is high compared to the next deeper layer (towards the CPU), the layers are connected with higher bandwidth to amortise memory accesses of programs with spacial locality. For multi-core processors, L1 cache is usually private to each core while L3 cache is shared by all of them. This provides fast private calculations, and communication via shared memory.

The mapping of data between layers of the hierarchy is done automatically by the hardware. If an element is required within a layer but not present, it is propagated from above through the layers automatically. This case is known as a **cache miss**: There is a request for an element which is currently not in the considered cache. If the element is found in cache, the event is called a **cache hit**. Cache hits usually account only for 2-3 CPU cycles. If a cache miss occurs, larger costs incur. The access time to the next memory layer requires the program to wait until data is present in the cache.

To exploit spacial locality, the transfer between memory layers is usually done in blocks of neighbouring data. In the L1 to L3 caches, these chunks are referred to as cache-lines. The pages in RAM lead to a similar behaviour between mass storage and RAM. Finally, it can be observed that the time to access a single random date from magnetic disks has a high latency for positioning the reading head and waiting for the right position to come on the disk. It then only takes a small amount of time to access all data that is on the same track on the disk. Usually, the further away from CPU, the larger the transferred blocks become in order to amortise latency.

As with the introduction of L3 cache, the trend seems to be that even more layers are added to the hierarchy. It is furthermore beyond dispute that parallelism will become an increasingly important factor in modern computers. Thus, parallel behaviour has to be exploited and independent threads of execution should be supported. Another interesting change taking place is the switch from magnetic disks to solid-state drives which induce a different, asymmetric memory access behaviour in that writing date is more expensive than reading.

In current environments, a **scan** through data, i.e. reading data in the order it is written on the disk, becomes quite fast with the help of the memory hierarchy. With optimised compilers and prefetching techniques, many simple tasks can be accelerated. However, it seems not plausible that good compilers will be able to optimise any implementation of a task towards locality in the future. Even if one counts on compilers optimisations, it is necessary to develop the right techniques to provide such optimisations.

Algorithms are, however, often constructed with the aim of minimising CPU cycles. The most commonly used model in algorithm analytics is the von Neumann aka **Random Access Machine** (RAM) model. This model describes a CPU with few registers and access to a large set of memory cells. An operation can perform calculations on elements in registers, or load records from memory into registers and write them back to memory respectively. The RAM model, however, does not reflect any terms of locality.

1.2 Models of Computation

A good model should be simple enough to allow for the design of algorithms. It is further desirable to provide comparability of algorithms to other models while it is comparable itself to other models. Yet, it should be capable of describing real hardware up to an extent that algorithms which perform well in the model are also efficient in practice. In the following, we describe a model that has served in a vast amount of work to design algorithms that optimise locality and are therefore well adapted to the memory hierarchy. Note, however, that the model does not reflect the costs of CPU operations.

1.2.1 The External Memory Model (EM or I/O-Model)

In a seminal paper by Hong and Kung [HK81] in 1981, a model was introduced which can be interpreted as an internal memory of limited capacity connected to an external memory of infinite size. Calculations can only be performed with records that reside in internal memory. To this end, records can be copied from external to internal memory and vice versa with an **input**, or **output** respectively (both are also referred to as **I/O operations**). Since internal memory size is restricted, for large enough problem instances, some records have to be evicted from internal memory to provide space for others during the computation. The ordering in which calculations are performed, and the decision which records are kept in memory therefore becomes crucial to the number of I/O operations. However, their model does not expose the ordering of records in external memory.

Several tasks such as matrix matrix multiplication[2] and FFT have been examined in this model, leading to an understanding of an optimal memory behaviour. For the considered tasks, both upper and lower bounds are derived. In order to obtain lower bounds, they assume what they call independent evaluation: For any given polynomial to be computed, all its monomials have to be calculated explicitly at some point within the computational process. Their model however encourages temporal locality only. It does not reflect the tracks or cache lines present in real hardware.

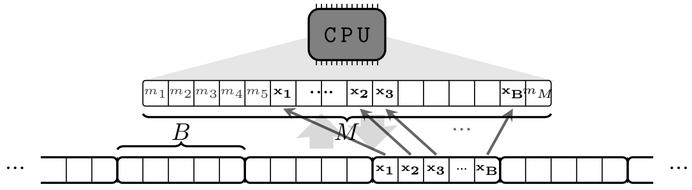

CPU
m_1 m_2 m_3 m_4 m_5 x_1 ⋯ x_2 x_3 x_B m_M
M
B
⋯
x_1 x_2 x_3 ⋯ x_B

Figure 1.1: The I/O-model (External Memory model) with an input operation.

In 1988, Aggarwal and Vitter [AV88] introduced the **External Memory model** (EM) a.k.a. **I/O-model** which consists of the same components, but additionally external memory is organised in **blocks**. Each input and output operation moves a whole block between internal and external memory (cf. Figure 1.1). This reflects real hardware settings and provides a good model to design algorithms where locality of memory accesses is optimised. Since then, the I/O-model served for a broad field of research to provide an understanding of the I/O behaviour of common algorithms, as well as to lead to I/O optimised algorithms. One can wonder whether a model of two memory layers is sufficient to model real hardware's memory hierarchy. However, the two layers are usually thought of being the highest level of cache hierarchy with I/O operations induced. Since memory accesses have a much higher latency the higher they appear in the hierarchy, I/O operations in deeper layers are dominated in time by the highest level. By now, this model is accepted and the I/O complexity, i.e. the (asymptotic) number of I/O operations, of many tasks is well understood.

[2]the direct method with n^3 multiplications

1.2.2 The Ideal Cache-Oblivious Model

In the EM model, the programmer usually has to specify M and B as parameters before being able to run the program. This leads to algorithms that are fine tuned for a specific layer of the hierarchy of a specific machine. In order to make programs applicable to a broad range of computers, the Ideal Cache-Oblivious model was proposed in [FLPR99]. Therein, the memory and block sizes are unknown to the algorithm, and can only be used for runtime analyses. One can consider algorithms in the Ideal Cache-Oblivious model to be written for the RAM model, but analysed in the I/O-model. In contrast to the explicit description of I/O operations in a program, it is assumed that an optimal cache replacement strategy is provided. This corresponds to real settings and takes away one of the specifications in algorithmic design. Though an optimal cache replacement strategy seems somewhat unrealistic, LRU or FIFO yield an approximation by only a constant factor [FLPR99]. Note that programs that perform optimally in a cache-oblivious setting are optimised for all layers of the memory hierarchy, not only for the communication between two layers.

1.2.3 The Parallel External Memory Model (PEM)

Since the early 2000s, a change in processor design paradigm from the pure increase of speed towards providing multiple cores on a single chip has been observed. Moore's law is no longer applicable to predict CPU speed, and nowadays, the number of cores per die is increasing. This shifts interests in algorithm design more towards good parallelisation which also raises a need for good parallel models.

As one of the earliest models to describe parallel computation, the **Parallel Random Access Machine** (PRAM) was suggested as a parallel version of the RAM (see e.g. [KR90]). In this model, multiple processors (usually synchronised) communicate via a shared memory. This gives rise to the question how parallel accesses to memory cells are handled. Variants are *(i)* exclusive read exclusive write (EREW) where each memory cell can only be accessed by one single processor at a time, independent from whether it will be read or written, *(ii)* concurrent read exclusive write (CREW) where a cell can be read by multiple processors at a time but write access is restricted to one, and *(iii)* concurrent read concurrent write where multiple processors can both read and write a single memory cell. In the last case, one further has to specify how write conflicts by multiple processors have to be handled, and what the result of the cell will be in that case.

The current multi-core architectures reflect this model in that a shared memory is commonly used by all cores. However, another common property is that private caches are attached to each core. The L1 cache is generally private to a single core. L2 cache in contrast is either assigned to a single, or only a few cores, depending on the architecture, and L3 cache is shared by all cores on the chip.

Following current multi-core memory design, the **Parallel External Memory model** (PEM) was proposed by Arge et al. [AGNS08] replacing the single CPU-cache in the EM model by P parallel caches with a CPU operating on each of them (cf. Figure 1.2). External memory is treated as shared memory, and within one parallel I/O, each processor can perform an input or an output of one block between its internal memory and the disk. Similar to the PRAM model, one has to define how concurrent access to the same cell is handled. Since concurrent access is always critical, a minimised access to shared data is desired. This corresponds to a maximisation of the local usage of data, i.e. an optimal use of private cache data. To handle multiple accesses to shared memory, we allow concurrent read exclusive write (CREW) in this work. However, the results can be easily modified for CRCW or EREW.

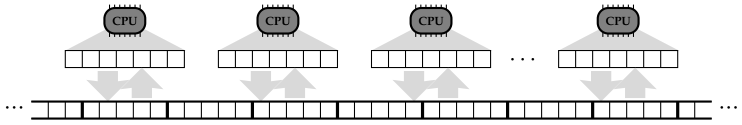


Figure 1.2: The Parallel External Memory model (PEM).

In this work, we follow [BBF$^+$07] where the idea of independent evaluations from [HK81] is combined with the EM model described in [AV88]. Referred to as the **Semiring I/O-model**, this model was used by Bender, Brodal, Fagerberg, Jacob and Vicari to obtain upper and lower bounds for the product of a sparse $N \times N$ matrix with a dense vector. The restriction to independent evaluations is realised by considering computation over a **commutative semiring**. We extend this notion to the PEM model, considering computation over an arbitrary semiring in this thesis. Corresponding to the definition of the PEM model, calculations can only be performed with records residing in internal memory, whereas the programs input and output have to be stored in external memory.

A commutative semiring $\mathbb{S} = (\mathbb{R}, +, \cdot)$ is an algebraic structure consisting of a set $\mathbb{R}$ with binary operations addition $(+)$ and multiplication $(\cdot)$ that are both associative and commutative. Multiplication is distributive over addition. Furthermore, there is a neutral element 0 for addition, 1 for multiplication and 0 is annihilating with respect to multiplication. In contrast to rings and fields, inverse elements are neither guaranteed for addition nor for multiplication, i.e. programs are not allowed to use subtraction and division.

For all algorithmic considerations in this work, if not otherwise noted, we consider the number of I/Os induced by a program in the **Semiring PEM model**. In analogy to the memory hierarchy, we sometimes refer to external memory as disk, and to internal memory as cache. In the following, we give a detailed description of the capabilities of this model as it will be used throughout this thesis.

Definition 1.1 (Semiring PEM Model, cf. [BBF+07, AGNS08]). *The Semiring PEM machine consists of P parallel processors, each connected to an internal memory which can hold up to M records of a commutative semiring $\mathbb{S} = (\mathbb{R}, +, \cdot)$, and a shared external memory of infinite size which is organised in blocks of B consecutive positions. The current* ***configuration*** *of a machine is described by the content of internal memories $\mathcal{M}_1, \ldots \mathcal{M}_P$ of the processors where $\mathcal{M}_p = (m_{p,1}, \ldots, m_{p,M})$ for $m_{p,j} \in \mathbb{R}$, $1 \leq p \leq P$, $1 \leq j \leq M$, and an infinite sequence of blocks $t_i \in \mathbb{R}^B$, $i \in \mathbb{N}$ of external memory. The positions in internal and external memory will be denoted as* ***cells*** *and their content as* ***record****. An* ***operation*** *is a transformation of one configuration into another, which can be one of the following types*

- ***Computation****: perform either one of the algebraic operations $m_i := m_j + m_k$ or $m_i := m_j \cdot m_k$, set $m_i := 1$, set $m_i := 0$ (deletion), or assign $m_j := m_i$ (copying) within internal memory of one processor.*
- ***Input****: copy the records from block t_i in external memory for some $i \in \mathbb{N}$ to records $m_{p,j_1}, \ldots, m_{p,j_B}$ in $\mathcal{M}_p$ for arbitrary $j_1, \ldots, j_B \in \{1, \ldots, M\}$, and a processor $p \in \{1, \ldots, P\}$.*
- ***Output****: copy the records $m_{p,j_1}, \ldots, m_{p,j_B}$ from $\mathcal{M}_p$ to block t_i for arbitrary $i \in \mathbb{N}$, $j_1, \ldots, j_B \in \{1, \ldots, M\}$, and $p \in \{1, \ldots, P\}$.*

We also refer to the latter two as ***I/O operations*** *or simply* ***I/Os****. Within one* ***parallel I/O****, each processor can perform either an input or an output operation. I.e. one parallel I/O refers to a sequence of I/O operations involving each processor once at the most.*

Using this, we define a **program** $\mathcal{P}$ as a finite sequence of operations. The number of parallel I/O operations describes the I/O costs of $\mathcal{P}$. If not otherwise noted, when referring to the number of I/Os as the costs of a program, we always mean *parallel* I/Os. An **algorithm** is a family of programs where the program can be chosen according to input size, and other parameters such as sparsity and the **conformation** of a matrix, i.e. the position of the non-zero entries. We use the notion of **uniform** and **non-uniform** similar to the use in the theory of circuits. A uniform algorithm depends only on the parameters B, M, and P and is applicable for any input size, i.e. matrix dimensions and number of non-zero entries. In a uniform algorithm, we allow branching based on the comparison of indices (not involving the semiring elements) to decide on the program progressively during runtime. A non-uniform algorithm in contrast is specifically adapted to the input size, and more importantly, to the conformation(s) of the given problem instance. Hence, the semiring elements specifying the input represent the only degree of freedom for a non-uniform algorithm. Throughout this thesis, we investigate the asymptotical number of I/O operations induced by a program, an algorithm, or by any algorithm for a specific problem. This is refer to as the **I/O complexity** of the program, algorithm, or problem. We aim to provide good upper and lower bounds on the I/O complexity in this thesis.

The **input-records** are initially given as contiguous records in external memory. Similarly, the **output-records** have reside contiguously in external memory at the end of the program. Depending on the task, the input and output describe vectors and matrices. For the non-zero entries of a sparse matrix, we usually think of a record to consist of an element of the semiring, annotated with the natural numbers defining its position in the matrix. This will be described more detailed in Section 1.3.

If a record r is an operand of a computation operation with result p, we call r a **predecessor** of p, and p a **successor** of r. We extend this notation to the transitive closure so that any record r used to derive p after several operations is called a predecessor of p, and p is its successor. As mentioned above, we consider CREW environments if not otherwise noted.

1.2.4 A Comparison to Other Parallel Models

First observe that there is a close relation between the PRAM and the PEM model when letting M be a constant and $B = 1$. In this case, the only difference is that multiple arithmetic operations that involve the same register elements in PRAM do not induce costs in the PEM model. However, assuming that a record can only represent a constant number of elements, the complexities of PEM and PRAM differ only by a constant factor.

Another famous model in the design of parallel algorithms is the **Bulk-Synchronous Parallel model** (BSP) [Val90]. One of the main differences to PRAM is that communication between processors is done via a network structure – usually managed by a router. Hence, there is no external storage device and all data has to be stored locally at the processors / computation nodes. A program for the BSP model is a sequence of supersteps where a superstep consist of independent (local) computation and a communication phase. Each L time units, it is checked whether the computational part of the superstep is completed by all processors. If so, the processors are synchronised for communication where the communication is realised in a so-called h-relation: Each processor is allowed to send and to receive up to h messages from other processors. Note (especially for $h = 1$) that a processor does not need to send to the same processors it receives from. For each superstep, a latency / startup cost s is assumed. The total communication cost of an algorithm with T supersteps is then $T \cdot (hg + s)$ where g is the throughput of the router – sometimes defined in ratio to the number of computations per time step to relate communication and computation times.

In contrast to the PRAM model where a shared memory allows for immediate communication throughout the computation, the BSP model models a rather coarsely synchronised environment where computation nodes operate independently within a superstep. The BSP model is therefore most applicable to describe environments of high bandwidth and high latency. It models a network structure such as internet communication quite adequately. The PRAM model on the contrary reflects a low bandwidth and low latency setting, and is thus more suitable for multi-core architectures. Also the PEM model was intended for such a setting [Sit09].

However, replacing shared memory in the PEM model by a direct communication network and setting $B = 1$ basically yields a BSP model. It is often assumed that processors in the BSP model have a restricted memory, e.g. $M = N/P$ for input size N. Note here that $M \geq N/P$ is required for the BSP model in order to store the input. If memory is unrestricted in the BSP model, one can set $M = N$, or even larger, in the PEM model to consider only costs that are induced by communication instead of costs induced by the restricted internal memory size. A special variant of the BSP model, which is described in the following, reveals many similarities to the PEM model.

The BSP* Model

As a significant change compared to [Val90] one can define the latency s in relation to the number of connections each processor has to establish. This is justified not only for hand shake protocols but also for the encapsulation process of messages performed by the network layers in the OSI model, i.e. todays network protocols. Hence, an incentive is given to send a number of records in a magnitude comparable to the connection-latency to the same processor. Another way to express this is the BSP* model in [BDH95] which encourages block-wise communication by defining a cost model per superstep of $\max \{gh \lceil m/B \rceil, L\}$ for the maximum message length m, and L time steps for computation within a superstep. If communication is more expensive than computation, as is assumed in the PEM model, the cost reduces to $gh\lceil m/B\rceil$. In this case, we can assume that $m \leq B$. Otherwise the communication can be split into $\lceil m/B\rceil$ multiple communication steps without changing the induced costs.

Any BSP* algorithm with ℓ supersteps on input size N can be simulated in the EREW PEM with $2h\ell + \lceil \frac{n}{PB} \rceil + \lceil \frac{o}{PB} \rceil$ parallel I/Os where n is the input size and o the output size, and internal memories are as large as memories in the BSP*-model. If there is no restriction on memory in the considered BSP* model, we set internal memory size M to the maximum number of records concurrently in memory of a computation node. For each superstep the h blocks that are sent via h-relation are written to external memory, causing h parallel I/Os. In h subsequent parallel inputs, these blocks are input by their destined processor. Additionally, input- and output-records may need to be read and written in the PEM model whereas a BSP model assumes input and output to be spread over processors / computation nodes. Altogether, all our lower bounds larger than $\lceil \frac{n+o}{PB} \rceil$ asymptotically hold for the BSP* model by a factor $1/h$. Note that the simulation can be non-uniform which, however, does not change the statement.

Similarly, a 1-BSP* algorithm can be derived from an EREW PEM algorithm that fulfils a certain communication balancing property. For each output that is made by the PEM algorithm, the corresponding block is sent to the processor that will read the block nearest in the future. In general, this implies that multiple blocks can be sent to the same processor violating the 1-relation. To tackle this problem, we annotate each block on disk during the PEM program by a processor id where it belongs to and demand that with one I/O, each processor may obtain at most one new block. Our algorithms can be observed to fulfil this condition in that they operate in phases where data is evenly (re)divided upon the processors. Furthermore, the assignment of input and output is not necessary in the BSP model, and the parallel sorting algorithm, which we use frequently, is even derived from a BSP algorithm (see [AGNS08]). However, since we consider CREW, additional scatter tasks may be required to obtain an algorithm for EREW. We omit a detailed transformation here since it is beyond the scope of this work.

1.2.5 The MapReduce Programming Paradigm

The MapReduce framework has been introduced in [DG04] by Dean and Ghemawat to provide a simple parallel model for the design of algorithms on huge data sets. It allows an easy design of parallel programs that scale to large clusters of hundreds or thousands of PCs. A MapReduce algorithm consists of a number of rounds that interleave serial functions, run in parallel, with parallel communication phases. A MapReduce round starts with splitting the input data among several workers such that a serial Map function can be executed in parallel for each part of the input. The intermediate results generated by these Map functions are then redistributed among the workers in the shuffle step. This is followed by a serial Reduce function which is executed on each worker to transform the intermediate results sent to this worker into a final output.

Since its introduction in 2004, apart from its intensive use by Google for tasks involving petabytes of data each day [DG10], the open source implementation Hadoop [Whi09] has found many applications including regular use by companies like Yahoo!, eBay, Facebook, Twitter and IBM. This success can be traced back to both the short development time of programs, even for programmers without experience in parallel and distributed programs, and the fast and fault tolerant execution of many tasks. However, there is also criticism passed on current progression towards MapReduce [DS, PPR+09, SAD+10]. This includes criticism on the applicability of MapReduce in all its simplicity to tasks where more evolved techniques have been examined already. Hence, it is of high importance to gain an understanding when and when not the MapReduce model can lead to implementations that are efficient in practice. In this spirit, MapReduce has been compared to PRAM [KSV10] and BSP [GSZ11] by presenting simulations in MapReduce. But theoretical foundations are still evolving.

MapReduce is usually applied in high performance computing on large data sets where fast memory access is especially important in order to provide fast computation. In clusters consisting of a large number of machines, the communication introduces a bottleneck of a kind similar to memory accesses. Therefore, we believe that the PEM model provides an important context in which MapReduce should be examined.

In Chapter 7, we provide further insights, contributing to the discussion on the applicability of MapReduce, in that we shed light on the I/O-efficiency loss when expressing an algorithm in MapReduce. Our investigation bounds the complexity that can be "hidden" in the framework of MapReduce in comparison to the parallel external memory (PEM) model. To this end, the I/O complexity of the *shuffle step* is considered. This step is the single communication phase between processors / workers during a MapReduce round. The redistribution of data is a reordering of intermediate results in shared or distributed memory. Since the communication is in general sparse, i.e. not all workers send to all other workers, we can relate the shuffle step to a sparse matrix transposition, considering the intermediate results as non-zero entries of a matrix. In this context, we consider several layouts, i.e. orderings of intermediate results, that seem natural in the context of MapReduce.

1.3 Sparse Matrix Computations

We use the notation $[N] = \{1, \ldots, N\}$ in order to classify indices in the following. Furthermore, throughout this thesis, matrices and vectors are bold symbols to distinguish them from scalars. In all the tasks regarding matrix multiplication in this thesis, we consider a sparse $N_{\mathrm{y}} \times N_{\mathrm{x}}$ matrix $\mathbf{A}$ with $\max\{N_{\mathrm{x}}, N_{\mathrm{y}}\} \le H \le N_{\mathrm{x}} N_{\mathrm{y}}$ non-zero entries. We assume that $\mathbf{A}$ contains no empty rows or columns. When referring to a matrix as **dense**, we usually mean that all entries are non-zero. Nevertheless, for our asymptotic results, $H = \Theta(N_{\mathrm{y}} N_{\mathrm{x}})$ is sufficient to call $\mathbf{A}$ dense. A matrix $\mathbf{A}$ is considered **sparse** if the number of non-zero entries is $\mathcal{O}(N_{\mathrm{y}} N_{\mathrm{x}})$. Most important are the results for $H = o(N_{\mathrm{x}} N_{\mathrm{y}})$. However, using this definition for sparsity, our complexity results for sparse matrix tasks carry over to dense matrices as well.

To the creation of the product $\mathbf{Ax}$ where $\mathbf{x}$ is a dense vector of dimension N_{x}, we refer to as **SpMV**. This task is considered for the product of $\mathbf{A} = (a_{ij})_{i \in [N_{\mathrm{y}}], j \in [N_{\mathrm{x}}]}$ with $w < B$ vectors $\mathbf{x}^{(1)}, \ldots, \mathbf{x}^{(w)}$ simultaneously where $\mathbf{x}^{(i)} = (x_j^{(i)})_{j \in [N_{\mathrm{x}}]}$. The output is denoted by $\mathbf{y}^{(i)} = \mathbf{Ax}^{(i)}$, $1 \le w \le B$, and the matrices given by the input and output vectors are $\mathbf{X} = \left[\mathbf{x}^{(1)} \ldots \mathbf{x}^{(w)}\right]$ and $\mathbf{Y} = \left[\mathbf{y}^{(1)} \ldots \mathbf{y}^{(w)}\right]$. Note that a record of the output vectors is given by the polynomial $y_j^{(i)} = \sum_{k \in A_j} a_{jk} x_k^{(i)}$ where $A_j \subseteq [N_{\mathrm{x}}]$ is the set of indices of the non-zero entries in the jth row of $\mathbf{A}$.

Multiplying $\mathbf{A}$ with a dense $N_{\mathrm{x}} \times N_{\mathrm{z}}$ matrix $\mathbf{B}$ for $N_{\mathrm{z}} \ge B$ will be called **SDM** throughout this thesis. To distinguish this case from SpMV where the dense matrix consists of less than B columns, we denote the output matrix by $\mathbf{C} = \mathbf{A} \cdot \mathbf{B}$. The value of an output record $\mathbf{c}_{ji}$ of the result matrix $\mathbf{C}$ is given by $\sum_{k \in A_j} a_{jk} \cdot b_{ki}$ where $A_j \subseteq [N_{\mathrm{x}}]$ describes again the positions of non-zero entries in the jth row of $\mathbf{A}$.

The same terminology is used when considering the product of $\mathbf{A}$ with another sparse matrix $\mathbf{B}$ which we refer to as **SSM**. The number of non-zero entries in the result matrix $\mathbf{C}$ depends crucially on the conformation of the input matrices. For the ith column of $\mathbf{A}$, $\mathbf{a}_i$, and the jth row of $\mathbf{B}$, $\mathbf{b}_j$, the result can be calculated $\mathbf{C} = \sum_i \mathbf{a}_i \mathbf{b}_i$. Hence, the number of multiplications over a semiring can be computed $\sum_i \alpha_i \beta_i$ where α_i, and β_i are the number of non-zero entries in $\mathbf{a}_i$, $\mathbf{b}_i$ respectively (note that $\mathbf{a}_i \mathbf{b}_i$ is a matrix). In a similar fashion, the number of non-zero entries in the jth row of $\mathbf{C}$ can be bounded above by $\sum_{i \in A_j} \beta_i$. It is common to describe the structure of the matrix product as a tripartite graph where the input matrices are considered adjacency matrices of bipartite graphs, connected in one dimension (here N_{y}) [Coh98]. A more detailed description is given in Chapter 6.

Additionally, we consider the computation of bilinear forms, denoted BIL. A bilinear form on a vector space is a linear function that maps two vectors $\mathbf{x}$ and $\mathbf{y}$ to a scalar z based on a matrix $\mathbf{A}$ by creating the product $z = \mathbf{y}^T\mathbf{A}\mathbf{x}$. The scalar product $\mathbf{y}^T\mathbf{x}$ of two vectors of the same dimension is a special case of bilinear forms where the identity serves as the matrix $\mathbf{A}$. Note that distributive law can be used for bilinear forms to reduce the number of multiplications in the semiring from $2H$ to $H + \min\{N_{\mathrm{x}}, N_{\mathrm{y}}\}$. Similar to SPMV, we consider BIL for $w < B$ vector pairs for the same matrix $\mathbf{A}$, performed simultaneously. The results are denoted by $z^{(i)}$ and are described by the polynomial $z^{(i)} = \sum_{1\le j\le N_{\mathrm{y}}} \sum_{k\in A_j} y_j^{(i)} a_{jk} x_k^{(i)}$ for $1 \le i \le w$.

1.3.1 Memory Layouts

By the **memory layout**, we denote the ordering of (input-)records in external memory. Considering sparse matrices, we assume that only non-zero entries are written in external memory. Here, we assume that a record describes the triple (value, column index, row index) where column and row indices are natural numbers. For a (sparse or dense) matrix $\mathbf{A}$, we distinguish the following layouts where the non-zero entries are given as contiguous records (cf. Figure 1.3).

- **Column major layout**: The records of $\mathbf{A}$ are ordered by column first, and by row index within a column.
- **Row major layout**: The records of $\mathbf{A}$ are order row-wise first, and column-wise within a row. Observe that row major layout can be obtained from column major layout by transposing $\mathbf{A}$, and vice versa.
- **Recursive layouts**: For completeness, recursive layouts shall be mentioned here as well. Such layouts are often helpful to construct efficient cache-oblivious algorithm. Many applications involve (recursive) space filling curves to define the ordering of records. However, we do not consider such layouts here.
- **Best-case layout**: When talking about the best-case layout, we consider the layout that induces asymptotically the fewest I/Os for given parameters and matrix conformation. It can be thought of that an algorithm may decide in which layout the matrix shall be provided. In terms of I/O complexity, we consider the worst-case over all conformations given the best-case layout.

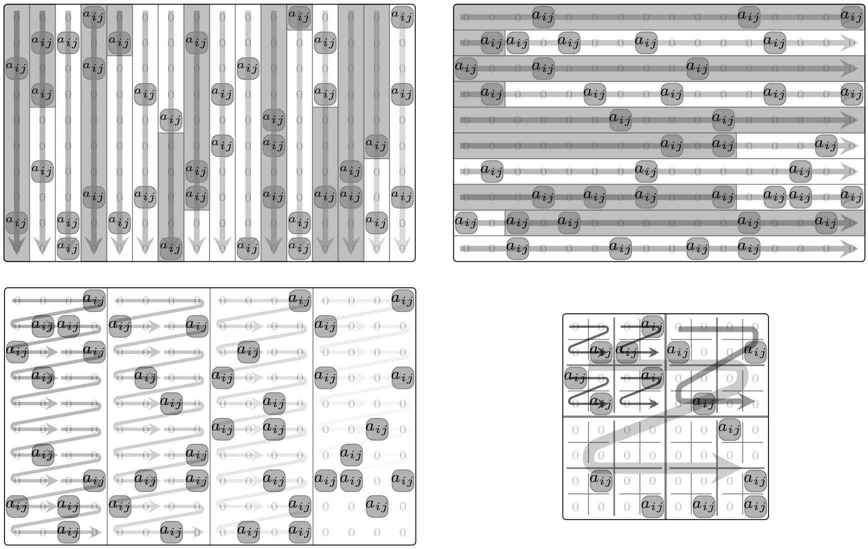


Figure 1.3: *Upper left:* A sparse matrix in column major layout with its block structure. *Upper right:* Row layout with its block structure of the same matrix. *Lower left:* Ordering of records in a layout with meta-columns where meta-columns are internally ordered row-wise. *Lower right:* A recursive Z-layout.

- **Worst-case layout**: Analogously, the worst-case layout can be defined to be the layout inducing the most I/Os.

The I/O complexity not only depends upon the layout of the input matrices, but also on the layout of the input vectors. However, a permutation on the records of a single vector can be shifted towards the matrix layout. Instead of considering a given ordering of vector records, these records can be considered contiguous while the matrix columns are considered to be permuted, leading to a different matrix layout.

1.3.2 Intermediate Results

The input of a program is specified by the **input-records**. These can be, depending on the considered task, matrix entries a_{ij}, b_{ij}, or vector entries $x_j^{(i)}$, $y_j^{(i)}$. For matrix vector multiplication and bilinear forms, we introduce the following terminology. Products of the form $a_{jk}x_k^{(i)}$, $y_j^{(i)}a_{jk}$, $x_k^{(i)}y_j^{(i)}$ and $y_j^{(i)}a_{jk}x_k^{(i)}$, for $1 \leq i \leq w$, $j \in [N_\mathsf{x}]$, $k \in [N_\mathsf{y}]$, are referred to as **elementary products**. In the context of bilinear forms, elementary products of the last form will be denoted **complete elementary product**. Sums of the form $\sum_{k\in S_x} a_{jk}x_k^{(i)}$, $\sum_{j\in S_y} y_j^{(i)}a_{jk}$, and $\sum_{j\in S_y}\sum_{k\in S_x} y_j^{(i)}a_{jk}x_k^{(i)}$, with $1 \le i \le w$, $S_x \subseteq [N_{\mathsf{x}}]$ and $S_y \subseteq [N_{\mathsf{y}}]$, are generally called **partial sums**. Altogether, the term **partial result** refers to any of these forms. Since our model is based on a semiring, the computation of $\mathbf{y}^{(1)}, \ldots, \mathbf{y}^{(w)}$ in SPMV includes the calculation of exactly wH elementary products $a_{jk} \cdot x_k^{(i)}$.

For the matrix matrix multiplication tasks SDM and SSM, we use the same terminology, by simply replacing $x_j^{(i)}$ with b_{ji}. No type of bilinear forms is considered therein so that all forms including some $y_j^{(i)}$ can be omitted.

It is easy to see that in the case of a matrix vector multiplication only elementary products and partial sums of the form $a_{jk}x_k^{(i)}$, and $\sum a_{jk}x_k^{(i)}$ respectively, can arise as detailed in [BBF+10]. For the case of evaluating a bilinear form, the distributive law can be useful, and some additional arguments are necessary to specify which intermediate results can be generated. Firstly, observe that no intermediate result that uses input-records from different pairs of vectors can be useful. Moreover, monomials with coefficient > 1 or with more than one $x_k^{(i)}$, $y_j^{(i)}$ or a_{jk} are useless. The same holds true if there is a mismatch in row or column between matrix coefficient and vector record. Hence, all monomials must be elementary products. Furthermore, any polynomial with at least two monomials that are elementary products of different types will continue to have this property and is hence not a predecessor of an output-record. Also a (non-trivial) sum of at least two monomials of type $x_k^{(i)}y_j^{(i)}$ is useless because it eventually has to be multiplied with some a_{jk} which would lead to mismatching monomials. Hence, the elementary products and partial sums listed above are sufficient to describe all intermediate results that arise during a program for BIL.

1.4 Previous Work

1.4.1 Matrix Multiplication

The complexity of computing the product of two $n \times n$ matrices over an arbitrary field is a long standing open problem. Until 1969, it was believed that $\mathcal{O}\left(n^3\right)$ arithmetic operations is essentially optimal when Strassen came up with the first subcubic algorithm. The widely known Strassen algorithm exploits distributivity and inverse elements of a field to reduce the number of multiplications. The Strassen algorithm [Str69], and its variant by Winograd [Win71] (which induces 15 instead of 18 additions) obtain the result by dividing each matrix into four quarters, and performing 7 instead of 8 matrix multiplications which leads to only $\mathcal{O}(n^{\omega})$ arithmetic operations for $\omega = \log_2 7 \leq 2.808$. It took another nine years for the exponent ω to be decreased further. First by Pan [Pan78], showing how to multiply two 70×70 matrices with 143640 arithmetic operations, and thus, $\omega \leq \log_7 0143640 < 2.796$, followed by Bini et al. [BCRL79] with $\omega < 2.78$ introducing approximate algorithms, and then in 1981 by Schönhage [Sch81] to $\omega < 2.522$. Schönhage introduced a technique which is also used to obtain the exponents ω in following works. In 1986, Strassen [Str86] presented a new attack on the matrix multiplication problem yielding $\omega < 2.479$. Coppersmith and Winograd [CW90] combined Strassen's technique with a novel analysis and obtained $\omega < 2.376$ which remained the best known upper bound for more than two decades. Recently, their technique was explored further by Stothers [Sto10], and then Williams [Vas11] who finally gave the bound $\omega < 2.373$. However, note that the asymptotically fastest matrix multiplication algorithms are more of a theoretical nature, and Winograd's variant of the Strassen algorithm [Win71] is still the most widely used fast matrix multiplication algorithm.

It is widely believed that $\omega = 2$. Nevertheless, since two decades, no significant progress was made in this direction. Instead, several conjectures implying $\omega = 2$ have been made in [CW90] and in [CKSU05], the latter one relying on a new group-theoretic approach. Unfortunately, both conjectures from [CW90], and one of the two conjectures in [CKSU05] have been found in [ASU11] to contradict another widely believed conjecture (the sunflower conjecture by Erdös and Rado [ER60]), leaving a focus on the remaining one by Cohn et al. [CKSU05].

Since all of these fast matrix multiplication algorithms rely on a recursive structure where parts of the matrix get summed, the number of non-zeros entries increases in each recursion. Thus, in general, sparsity does not lead to an asymptotic speed up. As described in Section 1.3, the number of multiplications, and the number of non-zero entries in the result matrix $\mathbf{C}$ depends strongly on the structure of the input matrices. For uniformly chosen input matrices (for a fixed number of non-zero entries), the expected running time of a hybrid version of the Strassen algorithm is examined in [MS04]. There, the recursion is stopped at a certain level, and the direct algorithm is used. By estimating the density of the matrices in recursion depth k of the Strassen algorithm, a formula is derived yielding the expected number of multiplications of this hybrid approach given the number of recursions and the densities of the input matrices.

Addressing upper bounds on the worst-case complexity, in [YZ04] Yuster and Zwick present an algorithm that applies a fast matrix multiplication algorithm like Coppersmith-Winograd to a denser set of columns while the product of the remaining columns is created with the classical direct approach. In contrast to [MS04], their algorithm has guaranteed running times even for products that become dense already after few recursion steps in Strassen's algorithm. For the numbers of non-zero entries m_1 and m_2 in the two input matrices, Yuster and Zwick obtain a running time bounded by $\mathcal{O}\left((m_1 m_2)^{0.35} n^{1.2} + n^{2+o(1)}\right)$. Hence, for multiplying a sparse with a dense matrix, the complexity of their algorithm is $\mathcal{O}\left(m_1^{0.35} n^{1.9} + n^{2+o(1)}\right)$. The direct approach instead requires $\mathcal{O}\left(m_1 n\right)$ multiplications which is still better for $m_1 < n^{1.38}$. For two sparse matrices that are both column-regular, i.e. each column contains m_1/n, m_2/n respectively, non-zero entries, $\mathcal{O}\left(m_1 m_2/n\right)$ multiplications are sufficient in a direct approach. A direct approach thus outperforms the algorithm in [YZ04] for $m_1 m_2 < n^{3.38}$. However, note that reducing the exponent ω of fast matrix multiplication also improves on the complexity of their algorithm.

1.4.2 I/O-model and PEM model

In their seminal paper [HK81], Hong and Kung present lower bounds on the I/O complexity and asymptotically optimal algorithms for several problems, including dense matrix multiplication and FFT. Applying their technique of partitioning the computation process (see Section 2.1), they show that any program for FFT requires $\Omega\left(N \log_M N\right)$ I/Os. For multiplying two $N \times N$ matrices with independent evaluations, they give a lower bound of $\Omega\left(\frac{N^3}{\sqrt{M}}\right)$ I/Os. They also present an algorithm for matrix multiplication where the matrices are partitioned into $\sqrt{M} \times \sqrt{M}$ tiles. Two tiles can be multiplied with $\mathcal{O}\left(M\right)$ I/Os, evaluating $M^{3/2}$ elementary products, thus inducing a total of $\mathcal{O}\left(\frac{N^3}{\sqrt{M}}\right)$ I/Os. Though not directly stated for block sizes $B > 1$, the algorithm can be extended straightforward to arbitrary block sizes inducing $\mathcal{O}\left(\frac{N^3}{B\sqrt{M}}\right)$ I/Os, given an appropriate layout. For column or row major layout, the internal memory size has to fulfil $M \geq B^2$ in order to load a tile with $\mathcal{O}\left(M/B\right)$ I/Os.

Aggarwal and Vitter [AV88], who introduced the classical I/O-model with block size $B \geq 1$, study permuting N records in external memory, as well as sorting and FFT. Using an argument for comparison-based sorting which is similar to lower bounds in other computational models based on comparison, they derive a bound of $\Omega\left(\frac{N}{B} \log_{M/B} \frac{N}{B}\right)$ on the number of I/Os. This complexity is achieved up to constant factors by an M/B-way merge sort described in [AV88] as well. Within an iteration of this merge sort, in each block of internal memory an input stream is buffered to generate a single merged output stream. Hence, each iteration reduces the number of runs

– initially N in an unsorted array – with one scan by a factor M/B. It is worth noting that corresponding to the concept of merge sort algorithms, their merge sort can be started with any number of pre-sorted runs (even of arbitrarily differing size), and can be stopped at any time to obtain a certain number of sorted runs. For FFT, they obtain a similar complexity as for sorting by using an insight from [WF81] that three FFT graphs can be used to construct a permutation network.

Aggarwal and Vitter further study the task of permuting N records. Note that scanning N contiguous records takes $\lceil N/B \rceil$ I/Os while a random access, like a direct permuting, induces N I/Os in the worst-case. They prove a complexity $\Theta\left(\min\left\{N, \frac{N}{B}\log_{M/B}\frac{N}{B}\right\}\right)$ for permuting, using a counting argument that we describe for the PEM model in Section 2.2. Given the simplicity of the task and its trivial complexity in RAM, it seems somewhat surprising that permuting in external memory is nearly as hard as sorting. Using a potential – which we consider in Section 2.3 – it is shown that dense matrix transposition in row or column major layout is a rather easy permutation task. The transposition of a dense $N_{\mathsf{y}} \times N_{\mathsf{x}}$ matrix has complexity $\Theta\left(\frac{N_{\mathsf{x}}N_{\mathsf{y}}}{B}\log_{M/B}\min\left\{B, N_{\mathsf{x}}, N_{\mathsf{y}}, N_{\mathsf{x}}N_{\mathsf{y}}/B\right\}\right)$. Recall that transposing corresponds to a layout change from column to row major layout, or vice versa. Given a dense matrix in column major layout, the columns can be used as pre-sorted runs and sets of basically B columns are merged together with the M/B-way merge sort, to obtain a row major layout. Under the so called **tall-cache assumption**, i.e. $M \geq B^{1+\varepsilon}$ for constant $\varepsilon > 0$, the I/O complexity reduces to $\Theta\left(N_{\mathsf{x}}N_{\mathsf{y}}/B\right)$.

Another class of easy permutations is considered in [CW93]. There, the I/O complexity of bit-matrix-multiply/complement (BMMC) permutations is studied. A permutation is called BMMC if the source indices are mapped to the target indices by an affine transformation of the bit vectors of source and target indices. This involves dense matrix transposition, and has the same worst-case complexity. In Chapter 8, we present a simpler proof of the upper bound leading to a much simpler algorithm.

Evaluating the matrix vector product $\mathbf{Ax}$ for an $N \times N$ matrix $\mathbf{A}$ with kN non-zero entries with a single vector has been investigated in [BBF⁺07, BBF⁺10]. They show that the I/O complexity of this task for matrices in column major layout is $\Theta\left(\min\left\{\frac{kN}{B}\overline{\log}_{\frac{M}{B}}\frac{N}{\max\{M,k\}}, kN\right\}\right)$[3] and for a layout chosen by the program (best-case layout) it is $\Theta\left(\min\left\{\frac{kN}{B}\overline{\log}_{\frac{M}{B}}\frac{N}{Mk}, kN\right\}\right)$, as long as $k \leq N^{\gamma}$ for some constant $\gamma > 0$ depending on the layout. In [BBF⁺10], the results are extended to the worst-case layout yielding a bound on the

[3] $\overline{\log}_b(x) := \max\{\log_b(x), 1\}$

number of I/Os of $\Theta\left(\min\left\{\frac{kN}{B}\,\overline{\log}_{\frac{M}{B}}\frac{N}{M}, kN\right\}\right)$.

The evaluation of bilinear forms in the I/O-model was considered as an optimisation problem in [Lie09]. There it is shown that an optimal program for the evaluation of matrix vector products is $\mathcal{NP}$-hard to find, even for $B = 1$, given that M is part of the input. Recently, they showed that the problem is even $\mathcal{NP}$-hard for fixed M [LJ12]. In [RJG+07], the I/O complexity of evaluating the bilinear form for an implicitly given (non-square) $N_\text{y} \times N_\text{x}$ matrix with H entries that form a diagonal band, i.e. entries are only placed near the diagonal, is determined to be $\Theta\left(\frac{H}{BM} + \frac{N_\text{x}+N_\text{y}}{B}\right)$.

In the context of the Ideal Cache-Oblivious model, several works consider restrictions on the parameter space in order to obtain optimal cache-oblivious algorithms. Most famous is the tall-cache assumption $M \geq B^{1+\varepsilon}$ for constant $\varepsilon > 0$ under which an optimal sorting algorithm is presented in [FLPR99]. Note that any lower bound for the complexity in the I/O-model carries over to the Ideal Cache-Oblivious model. However, since algorithms are more restricted, not all upper bounds in the I/O-model can be achieved in a cache-oblivious setting. Brodal and Fagerberg [BF03] prove that without the tall-cache assumption, sorting cannot be performed optimally. Additionally, they show that permuting is not possible cache-obliviously, not even under the tall-cache assumption. Intuitively speaking, the minimum $\min\left\{N, \frac{N}{B}\log_{M/B}\frac{N}{B}\right\}$ cannot be decided without knowledge of M and B in order to apply either a sorting algorithm or perform the direct approach. Using a similar argument, for the subclass of permutations that describe dense matrix transposition, optimality is proven to exist only under the tall-cache assumption [Sil07]. Hence, this statement also applies to BMMC permutations.

The PEM model was first introduced by Arge, Goodrich, Nelson and Sitchinava in [AGNS08] where a parallel merge sort is presented. This sorting algorithm is shown to perform optimal for the number of processors $P \leq N/B^2$ and $M = B^{\mathcal{O}(1)}$. In Chapter 2, we make a slight modification to this algorithm, and show its optimality if $P \leq \frac{N}{B\log^{\varepsilon}(N/B)}$ for constant $\varepsilon > 0$, or $B \geq \log_{M/B} N$ holds. Other tasks are considered in the PEM model in [Sit09], and [AGS10].

Using a different model of parallel caches, Irony et al. [ITT04] extended the results of Hong and Kung [HK81] to the parallel case. Similarly to [HK81], they require the independent evaluation of each monomial (elementary product). Using block size $B = 1$, they consider communication between P parallel processors with bounded memory each of size M. It is not surprising, they obtain a lower bound of $\Omega\left(\frac{W}{P\sqrt{M}}\right)$ where W is the number of arithmetic operations required. For dense matrix multiplication, this bound can be obtained algorithmically, especially with algorithms that rely on the replication of the input. In the same setting, lower bounds for several other algebraic matrix computations are studied by Ballard et al. [BDHS10, BDHS11b]. This involves the complexity of LU factorisation, Cholesky factorisation, LDL^T factorisation, QR factorisation, and eigenvalues and singular values algorithms. Recently, the I/O complexity of the Strassen algorithm was analysed in [BDHS11a]. For their proof, which applies a method closely related to Hong and Kung rounds but with a focus on graph expansion properties, they have to disallow the recomputation of intermediate results.

In practice, there often exist well-understood structures in the non-zero entries of sparse matrices which can be exploited. See [DDE+05, Vud03] for a survey on the vast amount of practical research on this topic. Several libraries supply techniques to analyse the structure and allow for fast computation, see [FC00, jIY00, RP96, VDY05, VDY06] among others. However, the focus of this thesis lies on the consideration of worst-case I/O-complexities and average case I/O-complexities over uniformly chosen matrices (generally without any restrictions on the conformation).

1.5 Structure and Contribution of this Thesis

In Chapter 2, we describe several techniques to obtain lower bounds in the I/O-model. These techniques are extended to the PEM model in this thesis, allowing for the first work with a focus on lower bounds in the PEM model. Using these techniques, we prove lower bounds on sorting and permuting in the PEM model. As explained in Chapter 2 it is in general not possible to adapt a given lower bound for the I/O-model to the PEM model simply by dividing by P. It is in fact possible to obtain a speed-up larger then P, when using P parallel processors since the overall internal memory becomes $M \cdot P$.

Succeeding the lower bound techniques, we describe our modifications to the PEM merge sort [AGNS08] which widens the parameter range of an efficient performance for a number of processor $P \leq N/B$ (previously $P \leq N/B^2$). The lower bound techniques show that this algorithm is optimal for sorting and permuting unless a direct strategy is superior for any $P \leq \frac{N}{B \log^{\varepsilon}(N/B)}$. Concluding this chapter, some basic algorithmic building blocks which are required for our parallel algorithms are described.

In the remainder of the thesis, we consider the parallel I/O complexities of SpMV, Bil, SDM, SSM, the MapReduce shuffle step, and two subclasses of permutations. In Chapter 3, we present a reduction of Bil to SpMV and vice versa. Thus, all complexities shown for SpMV carry over to Bil up to an additive factor $\mathcal{O}(\log P)$, hence, reducing the set of problems that have to be considered. We also use the reduction in that some algorithms in Chapter 4 are expressed for BIL, hence, yielding algorithms for SPMV by the transformation.

Previous work by Bender et al. [BBF+07] on SPMV is extended in Chapter 4 in several ways. Most straightforward to name are the extensions to non-square matrices and from the I/O- to the PEM model. On the algorithmic side, a different variant of the direct algorithm is presented, involving a look-up table generated from the input. This allows dropping a previous condition on the sparsity (the average number of non-zero entries per column was previously polynomially bounded). Additionally, we study the product of a fixed matrix with several vectors simultaneously. There, the number of vectors (or vector pairs for BIL) is restricted to $w < B$.

Multiplying a sparse matrix $\mathbf{A}$ with a dense matrix consisting of at least B columns is analysed in Chapter 5. The complexity of this task reveals a completely different character in that the I/O-performance depends crucially on the existence of denser than average submatrices of $\mathbf{A}$. For a low density, computing each row of the result matrix $\mathbf{C}$ directly is optimal. In higher densities, a modification of the tile-based algorithm in [HK81] performs optimally. For an intermediate density range, above average dense submatrices can be exploited. More precisely, a submatrix does not have to consist of a consecutive part of the matrix, but is defined by the entries a_{ij} of $\mathbf{A}$ that belong to the intersection of a set of rows S_r and a set of columns S_c. It will be proven that such submatrices exist with a certain density greater or equal to a threshold Δ, and that there are matrices without any submatrices of higher density than Δ. Hence, we obtain upper and lower bounds that match up to constant factors. Nevertheless, the upper bounds depending on denser submatrices are theoretical in nature, and are stated more to complement the lower bounds than to be implemented. In a preprocessing step, such denser submatrices have to be identified first. To show that this is possible in polynomial time, we present a derandomisation argument to find a denser than average submatrix.

In Chapter 6, we extend the techniques used in the previous results to study the I/O complexity of multiplying two sparse matrices. For simplicity, we restrict ourselves in this chapter to square matrices that allow for a direct estimation of the number of elementary products which have to be created. These are matrices that have the same number of non-zero entries in each column, or in each row. Unfortunately, it turns out that the used techniques are not strong enough to obtain asymptotically matching upper and lower bounds for most parameter ranges. By a reduction from SPMV, we derive lower bounds that can be matched algorithmically only for a number of non-zero entries per column/row less than $\min\{B, N/B\}$ for matrix dimension N. Techniques that rely on denser than average parts of the matrices, similar to Chapter 5, can only be exploited for a lower bound on the I/O complexity of a subclass of algorithms.

Complementing the considerations of sparse matrix tasks, we investigate the parallel I/O complexity of the shuffle step in MapReduce. This step is provided by the framework itself. In Chapter 7, we present lower bounds for the complexity of this step for several variants of MapReduce rounds. This bounds the complexity that can be hidden in the framework, when expressing an algorithm in MapReduce. It also bounds the worst-case efficiency loss when analysing a MapReduce algorithm in the PEM model. To match our lower bounds, we provide algorithms for all the considered variants. These are closely related to the algorithms in Chapter 4. However, we also extend them to sparse matrix transposition.

Finally, we consider two classes of permutations in Chapter 8. The one class, BMMC permutations, is known to be easy, i.e. can be processed with $\mathcal{O}\left(\frac{N}{B}\log_{M/B} B\right)$ I/Os [CW93]. We introduce the block graph; a new notion to analyse the structure of permutations in external memory with block size B. This allows for a much simpler proof of the complexity of BMMC-permutations, and a simple and parallelisable algorithm. Secondly, in an attempt to determine what makes a permutation difficult, we refute a conjecture that arises naturally. We construct a class of permutations whose block graphs have good expansion properties, but which are still easy to perform.

Concluding this thesis, we carve out some problems that still remain open. In particular, all our lower bounds rely on counting arguments and are not constructive. Thus, it is not completely understood what in particular makes an instance hard.

Several results in this thesis where previously published as conference proceedings or technical reports [GJ10a, GJ10b, GJ10c, GJ11, GJ12]. We will explain in the respective chapters if the results where published before, and if so, how they are extended here. For all our bounds, we assume that a program requires at least one I/O, and, hence, mean at least 1 when writing complexities using $\mathcal{O}$, Θ or Ω.

2

Techniques

Throughout this work, we make use of several common techniques to obtain upper and lower bounds on I/O complexities. These are described in the following sections. The first five sections are dedicated to standard techniques for lower bounds. All these techniques are extended here to the PEM model. The remaining sections cover some basic tasks that are required frequently for the PEM algorithms presented in the next chapters, most important a minor modification of the PEM merge sort from [AGNS08].

Concerning lower bounds, observe that the following reason makes it necessary to revise all the methods for the PEM model. For other parallel models, such as PRAM, the speed-up of a parallel environment compared to the single processor case is bounded by the number of parallel processors P. This is the case since any parallel program can be transformed into a single

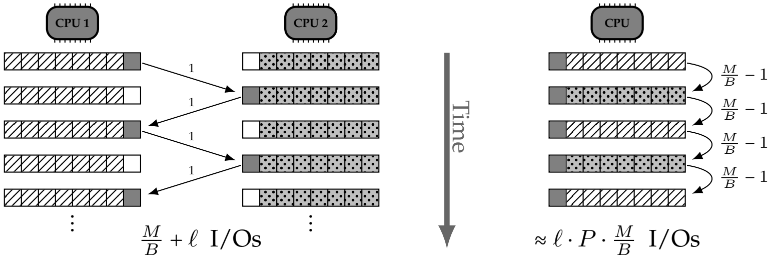


Figure 2.1: Simulating a PEM program with permanent communication in the I/O-model.

processor program by simulating each of the P processors in a round robin fashion. Hence, given a lower bound for a task in the serial RAM model, a lower bound for PRAM can be obtained by multiplying a factor $1/P$. Such a statement is however not given for the PEM model. A round-robin simulation of the P processors leads in worst-case to a factor $\Theta\left(P\frac{M}{B}\right)$ in the number of I/Os because the entire memory of each simulated processor has to be loaded for every simulation step (cf. Figure 2.1).

On the other hand, we extend the lower bound techniques to cases where the total number of records throughout the computation is smaller than the overall memory $P \cdot M$, i.e. internal memory is larger than necessary. Such a situation leads to a trivial setting for any constant number of processors. In contrast to the classical I/O-model, one can still obtain non-trivial lower bounds in the PEM model for non-constant P. This sheds a different light on the capabilities of the PEM model, in that I/Os are not only required to cope with the limited size of internal memory, but for the communication between parallel processors/machines. This reveals some of the similarities to the BSP model described in Section 1.2.4.

In many proofs, the binomial coefficients are estimates by the following well-known inequalities (see for instance [DK03]).

Observation 2.1. *For $x \geq y \geq 1$ it holds*

$$\left(\frac{x}{y}\right)^y \overset{(a)}{\leq} \binom{x}{y} \overset{(b)}{\leq} \left(\frac{ex}{y}\right)^y .$$

Proof. Inequality (a) is obvious since $\frac{x-i}{y-i} \geq \frac{x}{y}$ holds for all $i \geq 0$. Inequality (b) is given by the well-known inequality $y! \geq \left(\frac{y}{e}\right)^y$ (cf. [DK03]) and $\frac{x!}{(x-y)!} \leq x^y$. □

Normalisations Throughout this thesis, we usually consider programs that are normalised as follows. In a **normalised program**, all intermediate results are predecessors of an output-record. This includes any record in internal memory that is not used for any further computation leading to a predecessor of an output-record. Since I/Os can copy records, records are removed immediately from internal memory after an I/O if they are not involved in any further computation. Similarly, for any computation operation, records are removed immediately after the operation if not required further. Observe that every program for a task can be transformed into a normalised program for the same task without increasing the number of I/Os. If an intermediate result is not a predecessor of an output-record, it can be removed from the calculations along with all its successors.

2.1 Hong Kung Rounds

In [HK81], a method to obtain lower bounds on the I/O complexity of evaluating an algebraic function f is presented which is based on the **computation graph** describing the function.

Definition 2.2 (Computation Graph)**.** *The **computation graph** is a directed acyclic graph (DAG). Each node v corresponds to either an input (if v does not have any ingoing edges), or to an algebraic/computation operation, and its result. An ingoing edge $e = (u, v)$ of a vertex v means that v involves the result of u as an operand.*

They describe the red-blue pebble game which is played on this graph, implementing an I/O-model with block size $B = 1$. A red pebble located on a node defines that the record described by the node is situated in internal memory, a blue pebble symbolises that it exists in external memory. A blue pebble can be placed on a node having a red pebble on it, and a red pebble can be placed if a blue pebble is existent (corresponding to I/O operations). A computation operation can be performed if all ingoing edges have red pebbles on them, i.e. all operands are in internal memory. Then, the node corresponding to the computation and its result is pebbled red. The number of red pebbles during the game must never exceed M. Additionally, pebbles can be removed at any time throughout the game. The goal is to find a pebbling strategy to pebble a defined set of output nodes blue, given a set of input nodes with blue pebbles at the start of the game. A pebbling strategy fulfilling this goal corresponds to a valid I/O program.

In their main theorem in [HK81], Hong and Kung describe that a partitioning of the computation graph according to the following rules yields a lower bound on the number of I/O operations. Any partition set S may have a dominating set of size at most $2M$ where a dominating set D is a minimal set of nodes such that each path from an input node to S includes a node from D. Furthermore, a partition set S may include no more than $2M$ nodes without a child in S. They prove that given a lower bound on the number of partition sets $p(2M)$ in any such partitioning, the minimal number of I/Os of any program is $M \cdot (p(2M) - 1)$.

A maybe more intuitive interpretation of this result can be phrased as follows. We describe the single processor case first by using a reduction argument.

Lemma 2.3. *Assume there is an I/O program $\mathcal{A}$ performing ℓ I/Os for parameters M and B. Then there is an I/O program $\mathcal{B}$ computing the same function performing at most $3\ell + M/B$ I/Os for parameters $2M$ and B, that works in rounds: Each round* *consists of $2M/B$ input operations, an arbitrary number of computation operations followed by $2M/B$ output operations such that after each round internal memory is empty.*

Proof. A program $\mathcal{B}$ can be created by splitting the computation of $\mathcal{A}$ into rounds of M/B consecutive I/Os. With the additional memory, input and output can be serialised/buffered as claimed. The final content of internal memory in each round can be transferred to the next round with $2M/B$ I/Os. □

Upper bounding the progress that can be performed within one round, yields a lower bound on the number of rounds in a round-based program which in turn yields a general lower bound on the number of I/Os for any program. However, for a PEM program where processors can interchange records within a round, I/Os cannot be serialised like in Lemma 2.3. Thus, we describe a different view which allows for a similar result.

Consider a processor p, and the maximum number of records in its internal memory $M_{\max} \leq M$ during a program. During a sequence σ of operations performed by p involving at most $M_{\max}/B$ I/Os, there are at most $2M_{\max}$ records that can be predecessors of the computations performed during σ: At most $M_{\max}$ records that stem from input operations within σ, and at most $M_{\max}$ records that where present in internal memory at the beginning of σ. Similarly, at most $2M_{\max}$ records created during σ can be involved in succeeding computations. To measure the progress, a potential Φ is used, e.g. to describe the number of computation operations that are already performed. Note that there must be one processor in any program to evaluate f that causes a total potential change of at least $(\Phi(\ell) - \Phi(0))/P$ where $\Phi(\ell)$ is the final and $\Phi(0)$ the initial potential. These considerations lead to the following lemma.

Lemma 2.4. *Given a potential Φ describing the evaluation of a function f where $\Phi(0)$ denotes the initial potential and $\Phi(\ell)$ the potential after evaluating f. Let $\Delta(2M_{\max})$ be an upper bound on the change of the potential by computation involving at most $2M_{\max}$ predecessors, and having at most $2M_{\max}$ successors. Every (parallel) program to evaluate f requires at least $\left(\left\lceil \frac{\Phi(\ell)-\Phi(0)}{P\cdot\Delta(2M_{\max})} \right\rceil - 1\right) \frac{M_{\max}}{B}$ (parallel) I/Os.*

For all the tasks considered in this thesis, the computations are quite simple. Computing over an arbitrary semiring, for each monomial in the function describing an output record, there is at least one (separate) multiplication operation creating a predecessor of this monomial. Hence, the number of elementary products created can be used for the potential Φ. For our lower bounds obtained with this method, we usually upper bound the number of elementary products that can be created within one sequence, involving at most $2M_{\max}$ predecessors and creating at most $2M_{\max}$ records to leave the sequence.

2.2 Lower Bounds by Counting Arguments

2.2.1 Time Forward Analysis

In [AV88], a lower bound for permuting is presented which is based on a counting argument. There, the number of different tasks (output permutations) for a given array is compared to the number of different configurations an I/O-machine can reach after ℓ I/Os. For a family of programs that can create every permutation of an array of N records, there have to be at least $N!$ different configurations reached by the family. Considering the degrees of freedom of a parallel I/O, the number of distinct configurations that can be reached by programs with ℓ parallel I/Os can be bounded above. Relating the number of permutations $N!$, and thus the number of different output configurations, with the number of configurations reached by programs with ℓ I/Os yields a bound on ℓ.

This counting argument is also a main piece of the lower bounds for square sparse matrix dense vector multiplication in [BBF+10]. In this section, we bound the number of distinct configurations that can be reached in the PEM model after ℓ parallel I/Os, thus, the number of different programs with maximum number of I/Os ℓ. The main difference to the considerations in [AV88, BBF+10] is that parallel I/O operations have to be considered here. The concurrent read policy, for instance, allows for implicit copy operations. Additionally, we distinguish the case that internal memory of processors is not entirely filled. This leads to different bounds for cases where the input size falls below the total size of internal memories PM.

In the following, we assume a fixed input of total size n and abstract in a similar way to [BBF+10]. In an **abstract configuration**, empty blocks are ignored completely. Only the ordering of the non-empty blocks and their content is considered. Since we require the output to be written contiguously on disk, this is not a restriction. It only abstracts from programs that produce the same output, but have intermediate results written with a different number of empty blocks in between. Furthermore, we abstract from the ordering and multiplicity of records in blocks and in internal memory. Note that this may reduce the number of different tasks. For the case of permuting N records, we consider thus only $N!/B!^{\lceil N/B \rceil}$ different abstract output configurations.

For permuting records there is no computation required and it suffices to consider I/O operations only. To derive lower bounds for matrix computation tasks, in the contrary, we have to keep track of computation operations in order to determine the number of (abstract) configurations that can be reached. Since we abstracted from the multiplicity of records in blocks and internal memory, we do not have to consider copy operations. Note that in a normalised programs, deletion operations appear only immediately after an I/O, a sum operation, or a multiplication operation. Hence, we consider deletion operations always together with their preceding I/O, sum, or multiplication operation – and each of these operations is always considered together with possible succeeding deletion operations. For the following lemma, we consider the I/O operations in a program separately from the computation operations. To this end, we define the **I/O trace** and the **computation trace**. Both are required to describe a program. However, for a fixed computation trace there can be many valid programs with different I/O traces, and for a fixed I/O-trace there can be many valid programs with different computation trace. The computation trace fixes the sum and multiplication operations in a program, i.e. their position in the ordering of operations. Furthermore, for each computation operation, the cells (positions in the abstraction set) of internal memory that serve as operands and the cell that will contain the result are specified in the computation trace. Finally, the operands that are deleted afterwards are identified. The I/O trace defines the sequence of (parallel) I/O operations in a program by specifying which records are read or written and which positions on disk are accessed. Given the I/O trace of a program and its computation trace, the abstract configuration after the program is determined uniquely.

Lemma 2.5. *Given a family $\mathcal{F}$ of normalised programs with at most ℓ (parallel) I/Os and fixed computation trace, the number of abstract configurations that can be reached by $\mathcal{F}$ for a fixed input of size n is at most*

$$\prod_{l=1}^{\ell} 3^P \prod_{p=1}^{P} \binom{M_{p,l}+B}{B} 2^B \cdot 2(\lceil n/B \rceil + P\ell) \tag{2.1}$$

where $M_{p,l}$ is any upper bound on the number of records in internal memory of processor p before the lth I/O. We use $\lceil n/B \rceil + P\ell$ to upper bound the number of non-empty blocks in external memory at a time.

Proof. For a fixed input, the initial configuration is unique for all programs. Given an abstract configuration, we examine the number of possible succeeding abstract configurations that can be obtained by an arbitrary (parallel) I/O.

To this end, first assume that during the ℓth I/O, processor p performs an input while all the other processors stay idle. Let n be the total input size. After ℓ (parallel) I/Os, there can be at most $\lceil n/B \rceil + P\ell$ non-empty blocks in external memory, which can be read by the input operation. Afterwards, up to B new records are added to internal memory of processor p. In a normalised program, unneeded records that are copied into internal memory are removed immediately after an I/O, which leads to at most another 2^B distinct abstract configurations. Altogether, there can be at most $2^B(\lceil n/P \rceil + P\ell)$ succeeding abstract configurations.

Now consider the case processor p performs an output. There are less than $2(\lceil n/B \rceil + P\ell)$ positions available relative to the non-empty blocks to perform the output to: Considering the ℓth I/O to be the output, there are at most $\lceil n/B \rceil + P(\ell - 1)$ non-empty blocks that can be overwritten and another $\lceil n/B \rceil + P(\ell - 1) + 1$ empty positions relative to the non-empty blocks. Additionally, the content of the output block consists of up to B out of the $M_{p,l}$ records currently in internal memory of processor p. Records that shall not be copied and are not needed further in internal memory are removed right after the output which constitutes another 2^B different possible abstract configurations.

Each of the P processors can perform either an input, an output, or be idle during the lth parallel I/O. Thus, there are up to

$$3^P \prod_{p=1}^{P} \binom{M_{p,l} + B}{B} \cdot 2^B \cdot 2(\lceil n/B \rceil + P\ell)$$

possible configurations succeeding a given configuration caused by a (parallel) I/O operation. Creating the product over all ℓ (parallel) I/Os, this yields the lemma. Note that we also cover programs with $l < \ell$ I/Os since for each such program there is a corresponding program with ℓ I/Os and $\ell - l$ idle operations. □

2.2.2 Time Backward Analysis

Additionally to considering how many configurations can be reached from a single configuration over time, the proofs in [BBF+10] rely on the analysis of how many initial configuration can lead to a single final configuration. Thus, the change of configurations is analysed backwards in time, starting with a unique final abstract configuration. While the time forward analysis usually serves here and in [BBF+10] to analyse a distribution behaviour involving copy operations, the time backward analysis is used to consider a reducing character, like creating sums of records. In contrast to [BBF+10], we consider the changes of configurations directly here, instead of giving a transformation to inverse time in a program.

In the model in [BBF+10], I/O operations move records between memory layers whereas in our model, the records are copied by an I/O to enable concurrent reads. To consider a unique final configuration, independent from any intermediate results on disk, we ignore records in external memory that do not have a successor. Hence, we assume in an abstract configuration that records in external memory that do not belong to the final output disintegrate when read for the last time. This has especially implications for output operations that replace records. The replaced records cannot be read any more and hence disintegrated in an abstract configuration before. Thus, outputs are only performed to empty blocks in an abstract configuration.

The following lemma shows that a similar number of configurations is reached in both, the time forward analysis and the time backward analyses (cf. Lemma 2.5). This fact is caused by the simplicity of our model in which an input is very similar to an output considered time backwards and vice versa.

Lemma 2.6. *Given a family $\mathcal{F}$ of normalised programs with at most ℓ (parallel) I/Os and fixed computation trace, the number of initial abstract configurations for $\mathcal{F}$ that reach the same abstract output of size n is at most*

$$\prod_{l=1}^{\ell} 3^P \prod_{p=1}^{P} \binom{M_{p,l}+B}{B} 2^B 2(\lceil n/B \rceil + P\ell) \tag{2.2}$$

where $M_{p,l}$ is any upper bound on the number of records in internal memory of processor p after the lth I/O. Again, $\lceil n/B \rceil + P\ell$ is an upper bound for the number of non-empty blocks in external memory.

Proof. Considering only abstract configurations and given the computation trace, it remains to consider the I/O operations and their possible succeeding delete operations. Given a certain configuration after l I/Os, we now need to count the number of possible *preceding* abstract configurations. We start with the final configuration after ℓ I/Os. Since all blocks that do not belong to the output disintegrated, the final abstract configuration is unique for all programs that compute the same (abstract) output. Note that within ℓ (parallel) I/Os, there can be at most ℓP blocks that disintegrated completely. Hence, the total number of non-empty blocks in external memory throughout a normalised program is upper bounded by $\lceil n/B \rceil + \ell P$. During the description of our upper bound, we sometimes overestimate and ignore the possibility of considering inconsistent I/O traces.

For the lth (parallel) I/O, consider the case processor p performs an input. We upper bound the number of abstract configurations preceding the lth configuration. There are at most $\lceil n/B \rceil + P\ell$ non-empty positions on disk from which the input block could have been read. If the block disintegrated after the I/O, also one of the $\lceil n/B \rceil + P\ell$ empty positions relative to $\lceil n/B \rceil + P\ell - 1$ non-empty blocks could have been read. Furthermore, there are at most B records – which are not deleted right away – that stem from the current input. For $M_{p,l}$ being the number of distinct records in internal memory after the lth I/O, there are at most $\binom{M_{p,l}+B}{B}$ possibilities which records have been read. However, in an abstract configuration, the records (i.e. a copy) could have been present in internal memory before. Hence, for each set of accessed records, there are up to 2^B possible preceding abstract configurations, depending on whether the records were present before or appear as a new record. Since we assume that records that are never read afterwards disintegrated immediately after the input, the configuration on disk before the input can differ from the situation afterwards. However, for a defined set of records that have been input, and since the multiplicity of records in a block is ignored in an abstract configuration, the abstract configuration before the I/O is determined. Altogether, there are up to $\binom{M_{p,l}+B}{B}2^B 2P\ell$ preceding configurations if processor p performs an input, while all other processors are idle.

Now, consider the case that processor p performs an output. This alters one block on disk that was empty before. Hence, there are up to $\lceil n/B \rceil + P\ell$ possible preceding configurations. In the preceding configuration, the records of the output block are in internal memory. However, when fixing the block position where the output is performed to, these records are determined by the (known) abstract configuration after the output.

Each of the P processors can perform either an input, an output or be idle during the lth I/O. Thus, there are up to $3^P \prod_{p=1}^{P} \binom{M_{p,l}+B}{B} 2^B 2(\lceil n/B \rceil + P\ell)$ possible preceding abstract configurations for any given abstract configuration if a (parallel) I/O is performed. A family of programs with ℓ (parallel) I/Os and fixed computation trace can hence create the matrix vector product from at most

$$\prod_{l=1}^{\ell} 3^P \prod_{p=1}^{P} \binom{M_{p,l}+B}{B} 2^B 2(\lceil n/B \rceil + P\ell)$$

initial abstract configurations. □

Theorem 2.7 (Lower Bound for Permuting)**.** *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*

$$\Omega\left(\min\left\{\frac{N}{P}, \frac{N}{PB}\log_{\mathrm{d}}\frac{N}{B}\right\}\right)$$

where $\mathrm{d} = \max\left\{2, \min\left\{\frac{M}{B}, \frac{N}{PB}\right\}\right\}$.

Proof. First, observe that copying or deleting a record does not contribute to the number of possible permutations since only one version of the record appears in the output. The same holds for any computational operation. Hence, the computation trace is empty for all programs. In using Lemma 2.5, we also consider abstract configurations where a record disintegrates if it is not a predecessor of an output record. Records that are not deleted in internal memory after an input, automatically disintegrate on disk. Hence, the upper bound on the number of preceding configurations in (2.1) is not changed.

The number of abstract configurations described by all the $N!$ permutations is $N!/B!^{N/B}$ since the ordering of records in a block is ignored. By Lemma 2.5, we thus require

$$\prod_{l=1}^{\ell} 3^P \prod_{p=1}^{P} \binom{M_{p,l}+B}{B} 2^B \cdot 2(\lceil n/B \rceil + P\ell) \geq \frac{N!}{B!^{N/B}}$$

for the worst-case number of I/Os.

Since the total number of records (that did not disintegrate in the abstraction) at any time during a program is at most N, there can be no more than N non-empty blocks at a time. Hence, only N non-empty blocks can be used for an input. Additionally, when performing an output, one of the positions relative to the at most $N-1$ non-empty blocks on disk can be accessed (at least one record is in internal memory of a processor to perform an output). Observe furthermore that the product of the binomial coefficients can be bounded above by the single binomial coefficient given by drawing all at once from the union of all sets. The product of binomial coefficients can hence be bounded by

$$\prod_{p=1}^{P} \binom{M_{p,l}+B}{B} \leq \binom{\sum_{p=1}^{P}(M_{p,l}+B)}{PB} \leq \binom{\min\{MP, N\}+PB}{PB}.$$

Altogether, by replacing the upper bound on the number of non-empty blocks in Lemma 2.5 and bounding the product of binomial coefficients, we obtain

$$\left(3^P \binom{\min\{MP, N\}+PB}{PB} 2^{PB} \cdot N^P\right)^{\ell} \geq \frac{N!}{B!^{N/B}}. \tag{2.3}$$

Taking logarithms and estimating binomial coefficients according to Observation 2.1, we get

$$\ell P\left(\log 3N + B + B\log\frac{e(\min\{MP, N\} + PB)}{PB}\right) \geq N\log\frac{N}{eB}\,.$$

Assuming $M \geq B$, and considering the cases $N < PB$, and vice versa, yields

$$\ell P\left(\log 3N + B + B\log 2e\max\left\{1, \min\left\{\frac{M}{B}, \frac{N}{PB}\right\}\right\}\right) \geq N\log\frac{N}{eB}\,.$$

Hence, we have a bound of

$$\ell \geq \frac{N}{P}\frac{\log\frac{N}{eB}}{\log 3N + B\log 4e\max\left\{1, \min\left\{\frac{M}{B}, \frac{N}{PB}\right\}\right\}}\,.$$

Finally, we distinguish according to which term in the denominator is dominating. In the case of $\log 3N > B\log 4e\max\left\{1, \min\left\{\frac{M}{B}, \frac{N}{PB}\right\}\right\}$, the block size is bounded by $B < \log 3N$ so that $\log(N/B) \geq \log(N/\log 3N) = \Omega(\log N)$ holds. Together with the contrary case, the theorem is obtained.

A result for the average-case can be obtained by considering the worst-case complexity of the $N!/2$ permutations with the the least I/Os required. This changes the calculations for the worst-case bound by at most a constant factor. □

2.3 Lower Bounds with a Potential Function

Another method to obtain a lower bound for the I/O complexity is presented in [AV88]. They use a potential function to lower bound the complexity of dense matrix transposition. The potential describes only the movement of records, and is not capable of computational tasks. Lower bounds obtained with this method never reach beyond $\Omega\left(\frac{n}{B}\log_{M/B} B\right)$ for a number of blocks $\lceil n/B\rceil$ in the single processor case. Hence, they match only sorting algorithms for rather simplistic permutation classes such as bit-matrix-multiply/complement (BMMC) permutations (cf. Chapter 8). In the following, we extend the potential from [AV88] to the PEM model for a given input of n records.

For each block j present on disk at time t, its togetherness rating is defined by

$$\beta_j(t) = \sum_{i=1}^{\lceil n/B\rceil} \varphi(x_{ij}(t))$$

where $x_{ij}(t)$ is the number of records present in block j at time t that belong to the ith output block, and

$$\varphi(x) = \begin{cases} x \log x & \text{for } x > 0 \\ 0 & \text{otw.} \end{cases}$$

Similarly, for the internal memory of processor p, a togetherness rating of

$$\mu_p(t) = \sum_{i=1}^{\lceil n/B \rceil} \varphi(y_{ip}(t))$$

is assigned with $y_{ik}(t)$ being the number of records that belong to output block i and reside at time t in internal memory of processor k. The potential is then defined by

$$\Phi(t) = \sum_{p=1}^{P} \mu_p(t) + \sum_{j=1}^{\infty} \beta_j(t)\,. \tag{2.4}$$

In [AV88], the change of the potential induced by an I/O is bounded by $\Delta\Phi(t) < 2B \log \frac{M}{B}$. To extend the bound for the PEM model, we have to restate this argument for parallel I/Os. While the argument in [AV88] goes a little short (a "simple convexity argument" is mentioned), we state a complete proof here which makes use of a well-known property of the Kullback–Leibler divergence [KL51]. Our proof naturally applies to the single processor case as well.

Lemma 2.8. *The increase of the potential during one parallel I/O is bounded by*

$$\Delta\Phi(t+1) \le PB \log\left(2e \cdot \max\left\{1, \min\left\{\frac{M}{B}, \frac{n}{PB}\right\}\right\}\right).$$

Proof. Obviously, an output can never increase the potential. An input of block j by processor p, in contrast, induces the following change in the potential

$$\Delta\Phi(t+1) = \sum_{i=1}^{\lceil n/B \rceil} \varphi(y_{ip}(t) + x_{ij}(t)) - \varphi(y_{ip}(t)) - \varphi(x_{ij}(t))\,.$$

Hence, since in the worst-case all processors perform an input, the maximal increase of the potential function is bounded by

$$\Delta\Phi(t+1) \le \sum_{p=1}^{P} \sum_{i=1}^{\lceil n/B \rceil} \varphi(y_{ip}(t) + x_{ij_p}(t)) - \varphi(y_{ip}(t)) - \varphi(x_{ij_p}(t))$$

where j_p is the index of the block read by processor p.

Let $I_p(t)$ be the set of indices i such that $x_{ij_p}(t) \geq 1$ and $y_{ip}(t) \geq 1$. Substituting $\varphi(x)$, we obtain

$$\Delta\Phi(t+1) \leq \sum_{p=1}^{P} \sum_{i \in I_p(t)} \left(y_{ip}(t) \log \frac{y_{ip}(t) + x_{ij_p}(t)}{y_{ip}(t)} + x_{ij_p}(t) \log \frac{y_{ip}(t) + x_{ij_p}(t)}{x_{ij_p}(t)} \right).$$

We distinguish in the following between indices $i \in I_p(t)$ with $x_{ij_p}(t) > y_{ip}(t)$ and otherwise. To this end, we partition $I_p(t)$ into sets $I_p^{\mathsf{x}}(t) := \{i \in I_p(t) \mid x_{ij_p}(t) > y_{ip}(t)\}$ and $I_p^{\mathsf{y}}(t) := \{i \in I_p(t) \mid x_{ij_p}(t) \leq y_{ip}(t)\}$. Using this, we can upper bound

$$\Delta\Phi(t+1) \leq \chi(t+1) + \psi(t+1)$$

with

$$\chi(t+1) \leq \sum_{p=1}^{P} \sum_{i \in I_p^{\mathsf{x}}} \left(y_{ip}(t) \log \frac{2x_{ij_p}(t)}{y_{ip}(t)} + x_{ij_p}(t) \log \left(1 + \frac{y_{ip}(t)}{x_{ij_p}(t)} \right) \right)$$

and

$$\psi(t+1) \leq \sum_{p=1}^{P} \sum_{i \in I_p^{\mathsf{y}}} \left(y_{ip}(t) \log \left(1 + \frac{x_{ij_p}(t)}{y_{ip}(t)} \right) + x_{ij_p}(t) \log \frac{2y_{ip}(t)}{x_{ij_p}(t)} \right).$$

Observe that $\log(1+a) \leq a \log e$ for $a \geq 0$: Equality holds for $a = 0$. The derivative of $f(a) := \log(1+a)$ is $f'(a) = \log e/(1+a)$. Because $\log a$ is a concave function, the gradient $f'(0) = \log e$ is a sufficient condition. Using $x \log(1 + \frac{y}{x}) \leq y \log e$ for $x, y \geq 0$, yields

$$\chi(t+1) \leq \sum_{p=1}^{P} \sum_{i \in I_p^{\mathsf{x}}} \left(y_{ip}(t) \log \frac{2e x_{ij_p}(t)}{y_{ip}(t)} \right)$$

and

$$\psi(t+1) \leq \sum_{p=1}^{P} \sum_{i \in I_p^{\mathsf{y}}} \left(x_{ij_p}(t) \log \frac{2e y_{ip}(t)}{x_{ij_p}(t)} \right).$$

Now, let $X_a(t) := \sum_{p=1}^{P} \sum_{i \in I_p^a(t)} x_{ij_p}(t)$, and $Y_a(t) := \sum_{p=1}^{P} \sum_{i \in I_p^a(t)} y_{ij}(t)$ for $a \in \{\mathsf{x}, \mathsf{y}\}$. Furthermore, let $X(t) = X_{\mathsf{x}}(t) + X_{\mathsf{y}}(t)$ and $Y(t) = Y_{\mathsf{x}}(t) + Y_{\mathsf{y}}(t)$ and note that $X(t) \leq \min\{PB, n\}$ and $Y(t) \leq \min\{PM, n\}$. We can then substitute $\hat{x}_{ij_p}(t) = x_{ij_p}(t)/X_{\mathsf{y}}(t)$ and $\hat{y}_{ip}(t) = y_{ip}(t)/Y_{\mathsf{y}}(t)$ in $\psi(t+1)$, which leads to

$$\begin{aligned}\psi(t+1) &\leq X_{\mathsf{y}}(t)\log 2e + \sum_{p=1}^{P}\sum_{i\in I_p^{\mathsf{y}}(t)} X_{\mathsf{y}}(t)\hat{x}_{ij_p}(t)\left[\log\frac{\hat{y}_{ip}(t)}{\hat{x}_{ij_p}(t)} + \log\frac{Y_{\mathsf{y}}(t)}{X_{\mathsf{y}}(t)}\right]\\ &= X_{\mathsf{y}}(t)\log 2e + X_{\mathsf{y}}(t)\log\frac{Y_{\mathsf{y}}(t)}{X_{\mathsf{y}}(t)} + X_{\mathsf{y}}(t)\sum_{p=1}^{P}\sum_{i\in I_p^{\mathsf{y}}(t)}\hat{x}_{ij_p}(t)\log\frac{\hat{y}_{ip}(t)}{\hat{x}_{ij_p}(t)}\,.\end{aligned}$$

The last sum can be understood as a negative Kullback–Leibler divergence where, for fixed t, $\hat{x}_{ij_p}(t)$ and $\hat{y}_{ip}(t)$ constitute a probability distribution over $\{1,\dots,P\}\times I_p^{\mathsf{y}}(t)$. The Kullback–Leibler divergence is minimised when both probability distributions equal, and positive otherwise [KL51]. Hence, since it appears in a negative form, the last term is upper bounded by 0.

It remains to bound $\chi(t+1)$. By substituting $\hat{x}_{ij_p}(t) = x_{ij_p}(t)/X_{\mathsf{x}}(t)$ and $\hat{y}_{ip}(t) = y_{ip}(t)/Y_{\mathsf{x}}(t)$, and bounding the Kullback-Leibler divergence, we have

$$\begin{aligned}\chi(t+1) &\leq \sum_{p=1}^{P}\sum_{i\in I_p^{\mathsf{y}}(t)} Y_{\mathsf{x}}(t)\hat{y}_{ip}(t)\left[\log\frac{\hat{x}_{ij_p}(t)}{\hat{y}_{ip}(t)} + \log\frac{2eX_{\mathsf{x}}(t)}{Y_{\mathsf{x}}(t)}\right]\\ &\leq Y_{\mathsf{x}}(t)\log\frac{2eX_{\mathsf{x}}(t)}{Y_{\mathsf{x}}(t)} \leq X_{\mathsf{x}}(t)\log 2e\end{aligned}$$

where the last inequality is obtained by maximising $y\log\frac{2ex}{y}$ under the condition $x > y$. Note that $f(y) := y\log\frac{2ex}{y}$ is positive for any $2ex > y$ and has its only extremum at $y = 2x$ which is a maximum: The derivative is $f'(y) = \log\frac{2ex}{y} - \log e$ which is monotonically falling in y and 0 for $y = 2x$. However, because we have $X_{\mathsf{x}}(t) > Y_{\mathsf{x}}(t)$, an upper bound is obtained when setting $Y_{\mathsf{x}}(t) = X_{\mathsf{x}}(t)$.

Altogether, a lower bound of

$$\Delta\Phi(t+1) \leq (X_{\mathsf{x}}(t) + X_{\mathsf{y}}(t))\log 2e + X_{\mathsf{y}}(t)\log\frac{Y_{\mathsf{y}}(t)}{X_{\mathsf{y}}(t)}$$

is derived. This term is maximised for $X_{\mathsf{y}}(t) = X(t)$ and $Y_{\mathsf{y}}(t) = Y(t)$, resulting in an upper bound of $X(t)\log\frac{2eY(t)}{X(t)} \leq X(t)\log\frac{2e\cdot\min\{PM,n\}}{X(t)}$. We remark again that $x\log\frac{2ey}{x}$ is maximised for $x = 2y$, and monotonically increasing for $x < 2y$. The term is hence maximised for $X(t) = \min\{PB, n\}$. This results in an overall upper bound of

$$\Delta\Phi(t+1) \leq PB\log\left(2e\cdot\max\left\{1,\min\left\{\frac{M}{B},\frac{n}{PB}\right\}\right\}\right)\,.$$

□

Given this bound on the increase of the potential per I/O, we can extend the lower bound for dense matrix transposition in [AV88] to the PEM model.

Theorem 2.9. *The number of parallel I/Os required to transpose a dense $N_y \times N_x$ matrix stored in column major layout in the PEM model with $P \leq N_x N_y / B$ processors is*

$$\Omega\left(\frac{N_x N_y}{PB} \log_{\mathrm{d}} \min\left\{B, N_x, N_y, \frac{N_x N_y}{B}\right\}\right)$$

where $\mathrm{d} = \max\left\{2, \min\left\{\frac{M}{B}, \frac{N}{PB}\right\}\right\}$.

Proof. The proof follows that of [AV88], but with the potential according to our extension to the multi-processor case in (2.4). Let ℓ be the number of I/Os of a program for transposing the matrix. Because by the end of the algorithm all records have to reside in their target blocks, each block has a togetherness rating of $B \log B$. This yields a final potential of $\Phi(\ell) = N_\mathrm{x} N_\mathrm{y} \log B$ since internal memories are empty. The initial togetherness ratings, and hence the initial potential $\Phi(0)$, depend on the dimensions of the matrix. If $B \leq \min\{N_\mathrm{x}, N_\mathrm{y}\}$, no input block intersects an output block by more than one record (cf. Figure 2.2). Hence, each target group $x_{ij}(t)$ is at most 1, and the initial potential is $\Phi(0) = 0$.

For the case $\min\{N_\mathrm{x}, N_\mathrm{y}\} \leq B < \max\{N_\mathrm{x}, N_\mathrm{y}\}$, assume w.l.o.g. that $N_\mathrm{x} > N_\mathrm{y}$. Now, one block of a column major layout spans several columns. More specifically, it contains at most $\lceil B/N_\mathrm{y} \rceil$ records from each row. A block of the desired row major layout instead contains from each column at most one record. Hence, each input block contains at most $\lceil B/N_\mathrm{y} \rceil$ records from the same output block and the initial potential is $\Phi(0) \leq N_\mathrm{x} N_\mathrm{y} \log B / \min\{N_\mathrm{x}, N_\mathrm{y}\}$.

Finally, if $B < \min\{N_\mathrm{x}, N_\mathrm{y}\}$, both, input and output blocks, cover several columns/rows. An input block covers at most $\lceil B/N_\mathrm{y} \rceil$ columns and an output block covers at most $\lceil B/N_\mathrm{x} \rceil$ rows. The initial potential is hence given by $\Phi(0) \leq N_\mathrm{x} N_\mathrm{y} \log \frac{B^2}{N_\mathrm{x} N_\mathrm{y}}$. Applying Lemma 2.8 to bound the number of I/Os

$$\ell \geq \frac{H \log B - \Phi(0)}{\Delta\Phi(t)}$$

proves the theorem. □

2.4 A Parallel Lower Bound for Simple Functions

The lower bound described here mostly matches the asymptotic complexity of the gather and scatter tasks described below in Section 2.7.1. These tasks are generally used in this thesis to create sums of partial results that are

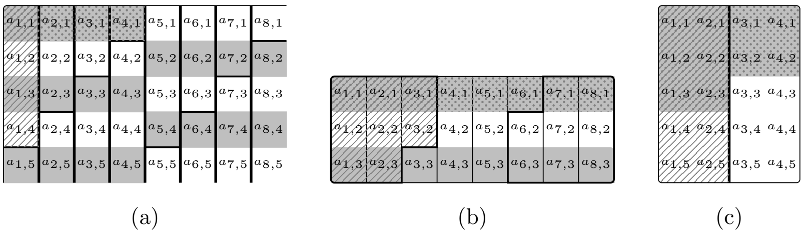

$a_{1,1}$ $a_{2,1}$ $a_{3,1}$ $a_{4,1}$ $a_{5,1}$ $a_{6,1}$ $a_{7,1}$ $a_{8,1}$
$a_{1,2}$ $a_{2,2}$ $a_{3,2}$ $a_{4,2}$ $a_{5,2}$ $a_{6,2}$ $a_{7,2}$ $a_{8,2}$
$a_{1,3}$ $a_{2,3}$ $a_{3,3}$ $a_{4,3}$ $a_{5,3}$ $a_{6,3}$ $a_{7,3}$ $a_{8,3}$
$a_{1,4}$ $a_{2,4}$ $a_{3,4}$ $a_{4,4}$ $a_{5,4}$ $a_{6,4}$ $a_{7,4}$ $a_{8,4}$
$a_{1,5}$ $a_{2,5}$ $a_{3,5}$ $a_{4,5}$ $a_{5,5}$ $a_{6,5}$ $a_{7,5}$ $a_{8,5}$
(a)
$a_{1,1}$ $a_{2,1}$ $a_{3,1}$ $a_{4,1}$ $a_{5,1}$ $a_{6,1}$ $a_{7,1}$ $a_{8,1}$
$a_{1,2}$ $a_{2,2}$ $a_{3,2}$ $a_{4,2}$ $a_{5,2}$ $a_{6,2}$ $a_{7,2}$ $a_{8,2}$
$a_{1,3}$ $a_{2,3}$ $a_{3,3}$ $a_{4,3}$ $a_{5,3}$ $a_{6,3}$ $a_{7,3}$ $a_{8,3}$
(b)
$a_{1,1}$ $a_{2,1}$ $a_{3,1}$ $a_{4,1}$
$a_{1,2}$ $a_{2,2}$ $a_{3,2}$ $a_{4,2}$
$a_{1,3}$ $a_{2,3}$ $a_{3,3}$ $a_{4,3}$
$a_{1,4}$ $a_{2,4}$ $a_{3,4}$ $a_{4,4}$
$a_{1,5}$ $a_{2,5}$ $a_{3,5}$ $a_{4,5}$
(c)

Figure 2.2: Block structures of dense matrices in column major layout (framed) and row major layout (white/gray background) for different matrix dimensions and block sizes. The togetherness rating of the hatched input block and the dotted output block is in (a) 1, in (b) $\lceil B/N_{\mathrm{y}} \rceil = 3$ and in (c) $\lceil B/N_{\mathrm{x}} \rceil \cdot \lceil B/N_{\mathrm{y}} \rceil = 6$.

spread among multiple processors, and to distribute records to processors that are not aware of the position on disk to read them.

Considering worst-case lower bounds, we can think of creating SPMV for a matrix $\mathbf{A}$ with a dense row. All the N_{x} records (the elementary products created with these records respectively) in this row have to be summed up. A lower bound for computing the logical "or" of N boolean values for the PRAM model was presented in [CDR86], and for the BSP model in [Goo99]. Arge et al. extend this bound to the PEM model in [AGNS08], proving a lower bound of $\Omega\left(\log \frac{N}{B}\right)$ parallel I/Os. This bound reflects the intuition that computing the result in a binary tree-like fashion is optimal.

The presented lower bound not only applies to the logical "OR", but to any function f on N bits that has an input I such that $f(I) \neq f(I(k))$ for all k where $I(k)$ is the input given by inverting the kth bit in I. This implies the same lower bound for other function such as the "XOR"-, and "AND"-function. Since we consider computations over an arbitrary semiring, the evaluation of an "XOR"-function on the entries of a row in $\mathbf{A}$ is implicitly required (e.g. for $GF(2)$). Hence, we obtain a lower bound of $\Omega\left(\log \frac{N_{\mathrm{x}}}{B}\right)$ in an EREW and CREW environment.

While the above lower bounds hold for any layout of $\mathbf{A}$, we can obtain a stronger lower bound if $\mathbf{A}$ is in column major layout. In this case, the records that belong to a certain row can be spread such that no two records of this row are in the same block. This is possible for $H/B \geq N_{\mathrm{x}}$. Otherwise, the records can be spread over the H/B blocks. Hence, there have to be $\min\{H/B, N_{\mathrm{x}}\}$ blocks considered in the worst-case which leads to a lower bound of $\Omega\left(\log\min\{H/B, N_{\mathrm{x}}\}\right)$.

Finally, observe that for for computing the bilinear form of two all-ones vectors with a sparse matrix $\mathbf{A}$, the logical "OR" over all non-zero entries in $\mathbf{A}$ has to be evaluated for the semiring $GF(2)$. Thus, BIL induces an I/O complexity of $\Omega(\log H/B)$.

2.5 A Lower Bound for Sorting in the Comparison Model

This section considers another method for lower bounds used in [AV88]. It is related to the counting argument from Section 2.2, but focuses on the number of comparisons required to identify a permutation. We restrict ourselves to the comparison model, where only comparisons of input-records are allowed but no computation on them, and apply a common method to lower bound sorting-complexity in the RAM model. An extension of Theorem 3.1 in [AV88] from the I/O-model to the PEM model is expressed in the following. Similar to the other bounds, we obtain a speed-up of P while the base of the logarithm changes for the PEM.

Theorem 2.10. *For $M \geq 2B$ and $P \leq \frac{N}{B \log^{\varepsilon}(N/B)}$ with constant $\varepsilon > 0$, the average-case and worst-case parallel I/O complexity of sorting N records is $\Omega\left(\frac{N}{PB} \log_{\mathrm{d}} \frac{N}{B}\right)$ with $\mathrm{d} = \max\left\{2, \min\left\{\frac{M}{B}, \frac{N}{PB}\right\}\right\}$.*

Proof of Theorem 2.10 for the Worst-Case. Each instance of a sorting problem describes a permutation. However, the permutation that has to be performed is not known to the algorithm and has to be determined by comparisons. Without restrictions on the input, an input of N records can lead to $N!$ permutations. After sorting the N records, the permutation is obviously determined uniquely. Hence, bounding the number of comparisons required to determine the permutation lower bounds the complexity of sorting N records.

We assume an adversary who decides the outcome of each comparison, consistently with his/her previous decisions. Note that the result of each comparison divides the space of possible permutations. After each comparison there is a set of permutations that are still potential output permutations, and a set of permutations that contradict one of the comparison results so far. In order to derive a lower bound for the worst-case, we assume that the adversary chooses the outcome of each comparison so that the set of potential output permutations is maximised. This guarantees for a large number of comparisons required to shrink the set to size 1.

We assume for our programs, that all comparisons are determined immediately after the input of a block. Consider processor p performing an input of block t_i. If t_i is accessed for the first time, there are at most $B!$ possible outcomes when determining the ordering of these up to B records. Otherwise, the ordering is already fixed. Furthermore, the ordering of the records of t_i within the records in internal memory of p has to be determined. Let $M_{p,l}$ be the number of records residing in internal memory of p before the input is performed. There are at most $\binom{M_{p,l}+B}{B}$ possible outcomes for this.

Hence, after an input of processor p the possible decisions of the adversary partition the space of potential output permutations into $\binom{M_{p,l}+B}{B}$ sets if t_i has been accessed before, and $B!\binom{M_{p,l}+B}{B}$ if not. By choosing the decisions such that the remaining set of potential output permutations is maximised reduces this set by a factor of at most $\binom{M_{p,l}+B}{B}^{-1}$, $\left(B!\binom{M_{p,l}+B}{B}\right)^{-1}$ respectively.

Note that there are only N/B input operations of input blocks that lead to $B!\binom{M_{p,l}+B}{B}$ outcomes. All other inputs partition the set of allowed orderings into $\binom{M_{p,l}+B}{B}$ sets. For the up to P inputs performed during one parallel I/O, the decisions of the adversary can be performed for each processor one after another in an arbitrary ordering. Hence, after ℓ parallel I/Os, the set of remaining potential output permutations is at most

$$N!\left(B!^{N/B}\prod_{l=1}^{\ell}\prod_{p=1}^{P}\binom{M_{p,l}+B}{B}\right)^{-1}. \tag{2.5}$$

We obtain a lower bound for the worst-case, by choosing ℓ such that (2.5) is at most 1 so that the output is uniquely determined. Taking logarithms and estimating binomial coefficients according to 2.1, we obtain the inequality

$$N\log N \le N\log eB + \sum_{l=1}^{\ell}\sum_{p=1}^{P} B\log\frac{e(M_{p,l}+B)}{B}.$$

Observe that each $M_{p,l}$ is bounded by $\min\{M, \ell\cdot B\}$. Hence, basic transformations yield

$$\ell \ge \frac{N\log\frac{N}{eB}}{PB\log\left(2e\cdot\min\left\{\frac{M}{B},\ell\right\}\right)} \ge \frac{N\log\frac{N}{eB}}{PB\log\left(2e\cdot\min\left\{\frac{M}{B},\frac{N}{PB}\log\frac{N}{B}\right\}\right)}.$$

Assuming $P \le \frac{N}{B\log^{\varepsilon}(N/B)}$ for constant $\varepsilon > 0$, implying $\log\left(\frac{N}{PB}\log\frac{N}{B}\right) = \mathcal{O}\left(\log\frac{N}{PB}\right)$, we obtain

$$\ell = \Omega\left(\frac{N}{PB}\log_{\mathrm{d}}\frac{N}{B}\right)$$

where $\mathrm{d} = \max\left\{2, \min\left\{\frac{M}{B}, \frac{N}{PB}\right\}\right\}$. □

For the average-case analysis, we use the following lemma.

Lemma 2.11. *Let δ, b, k, p be positive integers with $k \geq p$. Given a (comparison) tree T with each internal node v_i having an upper bound $\delta \cdot b_i$, $1 \leq b_i \leq b^p$ on the number of children, and for each path S to a leave, $\prod_{i \in S} b_i \leq b^k$ has to hold. Then, the number of internal nodes of depth t plus the number of leaves with depth at most t is bounded from above by $\delta^t b^{\min\{k, tp\}}$.*

Proof. We prove the lemma by upper bounding the number of leaves $l(t, k)$ of such a comparison tree with depth exactly t for given k. This bound extends to comparison trees of depth greater t by thinking of removing all nodes with higher depth, hence, turning internal nodes in depth t into leaves. The bound on $l(t, k)$ is proven by induction over t.

For $t = 1$, the number of leaves is at most δb^p. Thus, we have $l(1, k) \leq \delta b^p$ for any k (we required $p \leq k$). Now assume $l(t-1, k') \leq \delta^{t-1} b^{\min\{k', (t-1)p\}}$. A comparison tree T of depth t can be seen as the composition of multiple comparison trees $T_1, \ldots, T_n$ of depth at most $t-1$ connected by the root r of T. Let the degree of r be δb^x, $x \leq p$. Hence, the comparison trees $T_1, \ldots T_n$ have parameter $k' \leq k - x$. The number of leaves of T is therefore at most $\delta b^x \cdot l(t-1, k-x) \leq \delta^t b^{\min\{k, tp\}}$. □

Proof of Theorem 2.10 for the Average-Case. To prove the average-case, we consider the tree induced by the comparisons performed by an arbitrary algorithm for sorting N records (where we assume again that all comparisons are performed immediately after an input operation). In such a comparison tree, each node corresponds to a parallel input operation, and its outgoing edges correspond to the outcome of the comparisons performed afterwards. Hence, the degree of internal nodes is bounded by $d = \binom{M'+B}{B}^P$ where $M' = \min\{M, \ell \cdot B\}$, except for nodes that include the input of one or more blocks that have not been read before. Any such node can have degree up to $dB!^k$ where $k \leq P$ is the number of blocks that are input by the parallel input operation and have not been read before by any processor. However, on any path to a leave, the product of the multiplicative factors $B!^k$ for blocks not read before must not exceed $B!^{N/B}$.

Similar to the proof for the worst-case complexity, each node of the tree partitions the number of possible output permutations. Hence, the internal nodes in depth t together with all leaves with depth at most t constitute a partitioning of the $N!$ permutations. We want to upper bound the number of partitions in any comparison tree for a certain depth.

Let $t := \frac{\log N! - \frac{N}{B} logB!}{2P\log\binom{M'+B}{B}} \geq \frac{N}{2PB}\frac{\log(N/eB)}{\log(2e\cdot\min\{M/B,\ell\})}$ which is in $\Omega\left(\frac{N}{PB}\log_{\mathrm{d}}\frac{N}{B}\right)$ for $P \leq \frac{N}{B\log^{\varepsilon}(N/B)}$ with constant $\varepsilon > 0$. By Lemma 2.11 with $\delta = \mathrm{d}$, $b = B!$, $k = N/B$ and $p = P$, the number of partitions after t I/Os is bounded from above by

$$\binom{M'+B}{B}^{Pt} B!^{N/B} = \sqrt{B!^{N/B}N!}$$

because of the choice of t. Only permutations that are contained in a partition set of size 1 can be determined with t I/Os. In other words, at most $\sqrt{B!^{N/B}N!}$ permutations can be sorted in t I/Os. Since there are $N!$ permutations in total, using any comparison tree, $N! - \sqrt{B!^{N/B}N!}$ permutations remain after t comparisons that are not completely determined yet, and require hence more than t I/Os. Note that $B!^{N/B} = \prod_{i=1}^{B} i^{N/B} = \prod_{j=1}^{N}\left\lceil\frac{j}{N/B}\right\rceil$ which for $N \geq 2B$ and $N \geq 3$ is less than $N!/4$. Hence, there are at least $N!/2$ permutations that require more than t I/Os. This results in an average I/O complexity of $\Omega\left(\frac{N}{PB}\log_{\mathrm{d}}\frac{N}{B}\right)$ even when assuming that $N!/2$ permutations require no I/Os at all. $\square$

2.6 The PEM Merge Sort

We use the PEM merge sort by Arge et al. [AGNS08] within several algorithms in this work. Recall that a merge sort algorithm with degree d merges in each iteration d sorted lists, aka **runs**, to produce one combined (sorted) run. Starting with N single record lists, after $\log_d N$ iterations, a sorted list is obtained. The classical M/B-way merge sort for the I/O-model, which is described in [AV88], uses the M/B blocks that fit into internal memory to buffer M/B runs as input-streams. Hence, with one scan of the data, M/B runs can be merged which yields an optimal sorting algorithm for the single processor case.

While single processor implementations are straightforward, communication in parallel models make this algorithm more involved. A first Parallel merge sort for the PRAM model was presented by Cole in [Col88] (also note the correction of a flaw [Col93]). Based on this algorithm, Goodrich proposed a BSP version in [Goo99] with a merging degree $d = \max\left\{\sqrt{N/P}, 2\right\}$ to sort N records with P processors. The PEM merge sort in [AGNS08] is in turn based on this BSP merge sort algorithm, changing the merging degree to $d = \max\left\{2, \min\left\{\sqrt{N/P}, M/B\right\}\right\}$. We describe its principle coarsely in the following where we will not go into details of the quite complicated merging procedure.

The data is divided evenly upon the P processors throughout the algorithm such that each processor is permanently responsible for $\mathcal{O}(N/P)$ records. It starts with an optimal (single processor) external memory sorting algorithm such as the classical M/B-way merge sort [AV88] to create P pre-sorted runs of approximately even size. Then, the created runs are merged in a parallel way with a merging degree $\lceil \mathrm{d} \rceil$, where we use $\mathrm{d} = \max\left\{2, \min\left\{N/(PB), \sqrt{N/P}, M/B\right\}\right\}$ in this thesis. In contrast to [AGNS08], we slightly changed this merging degree in that we added the term $N/(PB)$ to the minimum. This maintains the I/O complexity within the original range in [AGNS08] while it improves on the number of I/Os for any larger number of processors. Additionally, we have matching lower bounds provided by Theorem 2.7 and Theorem 2.10 for a wider range of parameter settings. For asymptotic consideration, using

$$\mathrm{d}(N, M, B, P) = \max\left\{2, \min\left\{\frac{N}{PB}, \frac{M}{B}\right\}\right\}$$

is even sufficient as shown in the Theorem 2.12. There, it is show that sorting N records has a parallel I/O complexity of $\mathcal{O}\left(\frac{N}{PB}\,\overline{\log}_{\mathrm{d}(N,M,B,P)}\frac{N}{B}\right)$ where

$$\overline{\log}(x) = \begin{cases} \log(x) & \text{if } x > 2, \\ 1 & \text{otw.} \end{cases}$$

Usually, the parameters M, B and P are clear and fixed throughout a section. Then, we abstain from mentioning these parameters and write only $\mathrm{d}(N)$. Furthermore, if even the volume N that shall be sorted is obvious from the context, we simply use d for the sake of readability. In this case, we usually write out d once, like in the following theorem.

Theorem 2.12. *The I/O complexity of sorting N records with the PEM merge sort algorithm using $P \leq \frac{N}{B}$ processors is $\mathcal{O}\left(\frac{N}{PB}\,\overline{\log}_{\mathrm{d}}\frac{N}{B}\right)$ for $\mathrm{d} = \max\left\{2, \min\left\{\frac{N}{PB}, \frac{M}{B}\right\}\right\}$.*

Proof. Runs are merged with a merging degree $\lceil d' \rceil$ for

$$d' = \max\left\{2, \min\left\{\frac{N}{PB}, \sqrt{\frac{N}{P}}, \frac{M}{B}\right\}\right\}.$$

For $\max\left\{2, \min\left\{\frac{N}{PB}, \sqrt{\frac{N}{P}}, \frac{M}{B}\right\}\right\} = \max\left\{2, \min\left\{\sqrt{\frac{N}{P}}, \frac{M}{B}\right\}\right\}$, the original description of the algorithm and its analysis does not need to be modified. Otherwise, $\frac{N}{PB} < \sqrt{\frac{N}{P}}$ has to hold and the following can be observed. This case is equivalent to $P > \frac{N}{B^2}$ which is not considered in the analysis of the PEM merge sort in [AGNS08]. However, it is mentioned that the algorithm is still correct for larger P whereas their given I/O-bounds do not hold anymore (since they do not include the term $\frac{N}{PB}$ in the degree). The PEM merge sort is essentially the same as the BSP merge sort but takes care of restricted internal memory size and communication through shared memory in blocks. Reducing the merging degree does not violate any properties of the algorithm. It only increases the number of merging iterations to finish. In contrast, observe that the costs of one iteration in the PEM model is decreased when setting $d' = \frac{N}{PB}$ for the case $P > \frac{N}{B^2}$. By the analysis in [AGNS08], each iteration induces $\mathcal{O}\left(\frac{N}{PB} + \frac{2d'^2}{B} + d'\right)$ parallel I/Os which is still $\mathcal{O}\left(\frac{N}{PB}\right)$ for $P > \frac{N}{B^2}$ when setting $d' = \frac{N}{PB}$.

Recall that the merging degree influences the I/O complexity of the PEM merge sort by a factor $\log d'$. For asymptotic calculations it is hence sufficient to consider $d' = \mathrm{d} = \max\left\{2, \min\left\{\frac{N}{PB}, \frac{M}{B}\right\}\right\}$: Observe that $\log\frac{N}{PB} \leq 2\log\sqrt{\frac{N}{P}}$ and thus, $\frac{N}{PB}\log_{\sqrt{\frac{N}{P}}}\frac{N}{B} = \mathcal{O}\left(\frac{N}{PB}\overline{\log}_{\frac{N}{PB}}\frac{N}{B}\right)$. □

Lemma 2.13. *The PEM merge sort is optimal if either $P \leq \frac{N}{B\log^{\varepsilon}(N/B)}$ for constant $\varepsilon > 0$, or $B \geq \log_{M/B} N$ holds.*

Proof. For $P \leq \frac{N}{B\log^{\varepsilon}(N/B)}$ and constant $\varepsilon > 0$, the optimality is given by Theorem 2.10. Obviously, a lower bound for permuting N records is a lower bound for sorting N records. If $B \geq \log_{M/B} N$, the permuting complexity from Theorem 2.7 reduces to $\Omega\left(\frac{N}{PB}\log_{\mathrm{d}}\frac{N}{B}\right)$. □

Changing the Number of Runs The PEM merge sort reduces in each iteration the number of sorted runs by a factor d. Stopping the PEM merge sort after k iterations leads obviously to $\lceil N/\mathrm{d}^k \rceil$ runs of length at most d^k each.

Additionally, given r pre-sorted runs, the PEM merge sort can be started with these runs to reduce the number of iterations to $\log_{\mathrm{d}} r$. If all runs have equal size, this corresponds to an arbitrary iteration of the PEM merge sort, and is hence described in [AGNS08]. Otherwise, we divide runs that contain more than N/r records into several runs, and runs that contain less than N/r records are filled with dummy records. This can be done such that there are in the end $c \cdot r$ runs for a constant $c \geq 1$ with N/r records each.

To achieve this modification of runs in parallel, we use the range-bounded load-balancing algorithm described below in Section 2.7.3. With this algorithm, at most $\lceil 2N/P \rceil$ records from at most $\lceil 2r/P \rceil$ runs are assigned to each processor, by inducing $\mathcal{O}\left(\frac{N}{PB} + \log\min\{B, P, r\}\right)$ I/Os. Then, each processor can split and fill its at most $\lceil 2r/P \rceil$ assigned runs within $\mathcal{O}\left(\frac{N}{PB}\right)$ I/Os.

Starting the PEM merge sort with r pre-sorted runs and stopping it as soon as there are less than q runs hence has I/O complexity $\mathcal{O}\left(\frac{N}{B}\,\overline{\log}_{\mathrm{d}}\,\frac{r}{q}\right)$ if the r runs all have the same size, and $\mathcal{O}\left(\frac{N}{B}\,\overline{\log}_{\mathrm{d}}\,\frac{r}{q} + \log\min\{B, P, r\}\right)$ otherwise.

2.6.1 Dense Matrix Transposition

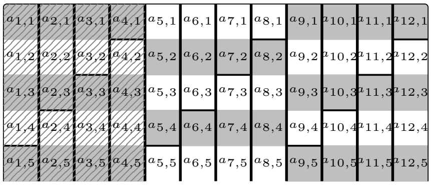
As observed in [AV88], a merge sort can be applied to transpose a dense $N_{\mathrm{y}} \times N_{\mathrm{x}}$ matrix from column to row major layout (and vice versa) with

$$\mathcal{O}\left(\frac{N_{\mathrm{x}}N_{\mathrm{y}}}{B}\,\overline{\log}_{\frac{M}{B}}\min\left\{B, N_{\mathrm{x}}, N_{\mathrm{y}}, \frac{N_{\mathrm{x}}N_{\mathrm{y}}}{B}\right\}\right)$$

I/Os in the (serial) I/O-model. This term reflects the togetherness ratings in the construction of the lower bound in Theorem 2.9. Similar to the initial potential there, depending on the dimensions and the block size, the task becomes less difficult if multiple records that belong to a single output block are in the same input block already. While the original algorithm for the I/O-model involves the classical M/B-way merge sort in [AV88], it is easy to extend the algorithm to the PEM model using the PEM merge sort.

Obviously, a complete resorting of the records can yield a row major layout with $\mathcal{O}\left(\frac{N_{\mathrm{x}}N_{\mathrm{y}}}{PB}\,\overline{\log}_{\mathrm{d}(N_{\mathrm{x}}N_{\mathrm{y}})}\,\frac{N_{\mathrm{x}}N_{\mathrm{y}}}{B}\right)$ I/Os by using the PEM merge sort. If $N_{\mathrm{y}} > B$, the N_{x} columns can be used as pre-sorted runs. Hence, as described above, $\mathcal{O}\left(\frac{N_{\mathrm{x}}N_{\mathrm{y}}}{PB}\,\overline{\log}_{\mathrm{d}(N_{\mathrm{x}}N_{\mathrm{y}})}\,N_{\mathrm{x}}\right)$ I/Os are sufficient.

For the other cases, we aim to create meta-columns of B columns that are in a row-wise ordering. To this end, the matrix is partitioned into $\lceil N_{\mathrm{x}}/B \rceil$ sets of at most B contiguous columns. For each set of columns, the PEM



$a_{1,1}$ $a_{2,1}$ $a_{3,1}$ $a_{4,1}$ $a_{5,1}$ $a_{6,1}$ $a_{7,1}$ $a_{8,1}$ $a_{9,1}$ $a_{10,1}$ $a_{11,1}$ $a_{12,1}$
$a_{1,2}$ $a_{2,2}$ $a_{3,2}$ $a_{4,2}$ $a_{5,2}$ $a_{6,2}$ $a_{7,2}$ $a_{8,2}$ $a_{9,2}$ $a_{10,2}$ $a_{11,2}$ $a_{12,2}$
$a_{1,3}$ $a_{2,3}$ $a_{3,3}$ $a_{4,3}$ $a_{5,3}$ $a_{6,3}$ $a_{7,3}$ $a_{8,3}$ $a_{9,3}$ $a_{10,3}$ $a_{11,3}$ $a_{12,3}$
$a_{1,4}$ $a_{2,4}$ $a_{3,4}$ $a_{4,4}$ $a_{5,4}$ $a_{6,4}$ $a_{7,4}$ $a_{8,4}$ $a_{9,4}$ $a_{10,4}$ $a_{11,4}$ $a_{12,4}$
$a_{1,5}$ $a_{2,5}$ $a_{3,5}$ $a_{4,5}$ $a_{5,5}$ $a_{6,5}$ $a_{7,5}$ $a_{8,5}$ $a_{9,5}$ $a_{10,5}$ $a_{11,5}$ $a_{12,5}$

Figure 2.3: Meta-columns of B columns are sorted by row index using the PEM Merge sort.

merge sort is applied using the B contiguous columns as pre-sorted runs. If N_x is an integer multiple of B, the block structure of all rows is similar (as depicted in Figure 2.3) and the meta-columns can be chosen to match the output block structure. In this case, we are already done. Otherwise, the records of each output block can be contained in two meta-columns. Then, each processor is assigned to create $\frac{N_x N_y}{PB}$ output blocks. With at most two I/Os per assigned block per processor, the row major layout is constructed. This last step induces $\mathcal{O}\left(\frac{N_x N_y}{PB}\right)$ I/Os.

2.7 Building Blocks for Parallel Algorithms

In this section, we describe some basic building blocks that are required frequently for the parallel algorithms in this thesis.

2.7.1 Gather and Scatter Tasks

To form a block from records that are spread over several internal memories, the P involved processors can communicate records in a tree-like fashion to form the complete block in $\mathcal{O}(\log\min\{P, B\})$ I/Os. This is referred to as **gather** operation. If a block is created by computations involving records from P processors (e.g. summing multiple blocks) still $\mathcal{O}(\log P)$ I/Os are sufficient. Similarly, a block can be distributed to multiple processors with $\mathcal{O}(\log P)$ I/Os by a **scatter** operation (cf. Figure 2.4). If n independent blocks for each processor have to be scattered / gathered, this can be serialised. Since each processor is involved only once within each gather/scatter task, $\mathcal{O}(n + \log P)$ I/Os are sufficient. Note that for all gather and scatter tasks, the communication structure has to be known to each participant. This is for example the case when participating processors constitute an ordered set which is known to all. In most cases, the participating processors have consecutive id, and the ids of the first and last processors are known to all processors. Hence, each processor is in knowledge of its rank within the task.

2.7.2 Prefix Sums

Sometimes, we require the computation of prefix sums. This task has been extensively studied in parallel models. For the PEM model see [AGNS08] for a description.

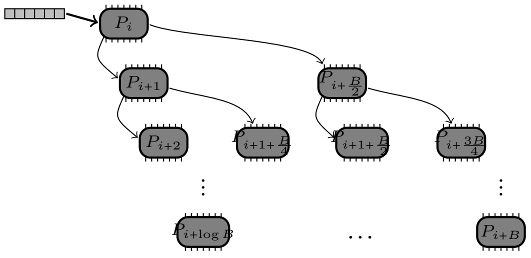


Figure 2.4: Scattering a block to B consecutive processors. Each processor is involved once during the process.

2.7.3 Range-Bounded Load-Balancing

Due to load-balancing reasons, the following task will appear several times. Given n tuples (i, x) that are located in contiguous external memory, ordered by some non-unique key $i \in \{1, \ldots, m\}$ for $m \le n$. Assign the n tuples to P processors such that each processor gets at most $\lceil 2n/P \rceil$ tuples assigned to it, but keys are within a range of size no more than $\lceil 2m/P \rceil$.

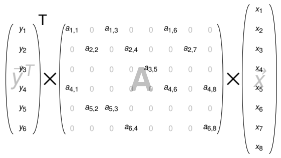
The task can be solved by dividing the P processors into $\lceil P/2 \rceil$ **volume processors** and $\lfloor P/2 \rfloor$ **range processors**. First, data is assigned to the volume processors by volume. To this end, each volume processor gets a consecutive piece of at most $\lceil 2n/P \rceil$ tuples assigned to it. For communication purpose, we assume that for each processor there is an exclusive block reserved for messages in external memory denoted as **inbox**.

For the next step, think of the ordered set of keys $(1, \ldots, m)$ being partitioned into $P/2$ ranges of at most $\lceil 2m/P \rceil$ continuous keys each (further referred to as **key range**). To cope with the problem of having too many different keys assigned to the same volume processor, we reserve the ith range processor to become responsible for tuples within the $(i+1)$th key range. Now, each volume processor scans its assigned area and keeps track of the position in external memory where a new key range begins. This requires only $\mathcal{O}\left(\frac{n}{PB}\right)$ I/Os for scanning the assigned tuples while one block in memory is reserved to buffer and output the starting positions of a new key range.

Afterwards, each volume processors with more than $\lceil 2m/P \rceil$ keys assigned to it, created a list of memory positions where a new key range begins. This list is than scattered block-wise to the range processors reserved for the corresponding key ranges. Although we assume CREW, it is necessary to distribute this information because a program running on a range processor is not aware of the memory position to find the information since it depends on which volume processor got assigned the key range. The distribution can be achieved with $\mathcal{O}\left(\frac{n}{PB} + \log\min\{m, B, P\}\right)$ parallel I/Os where each block of a list is written into the inbox of the first range processor concerned by this block, and from there spread by a scatter operation to the other range processors (cf. Figure 2.4). In this manner, each processor can be informed about both the beginning and the end of its assigned area. Note that it can never happen that a range processor receives information from multiple volume processors because for a key range divided to multiple volume processors only the first can dispose the tuples.

2.7.4 Contraction

Given an input sequence of n records written in m blocks on disk, where some of the memory cells are empty, it is possible to contract the sequence such that empty cells are removed and the sequence is written in the same ordering as before, but with records stored contiguously with $\mathcal{O}(m/P + \log P)$ I/Os. To this end, the input blocks are assigned equally to the P processors. Each processor gets a contiguous piece of up to $\lceil m/P \rceil$ blocks assigned to it, and processors are assigned in ascending order. Within one scan, each processor can determine the number of non-empty cells contained in its assigned area. With this information, and the given ordering of processors, using prefix sum computation, in $\mathcal{O}(\log P)$ I/Os, it is known to each processor where its records shall be placed in a contiguous output. However, some of the output blocks may contain records that are assigned to multiple processors. For each such block, the processor with the first, and the processor with the last record assigned to it are in knowledge of the fact, given the prefix sum results. Since processors are assigned in ascending order, a gather operation can be used to create blocks with records assigned to several processors. Note that each processor only needs to participate in at most two gather operations. The gather process can be achieved by an output of each first and last processor of a block, in that both write their indices into a designated table. Then, each processor can read this information and send its records to the inbox of the responsible processor in the gather process.



3 Bilinear Forms

In this chapter, we show that in terms of the (parallel) semiring I/O complexity the tasks BIL and SPMV are asymptotically similar. This implies that it is sufficient to state a bound on the I/O complexity for one of the tasks in order to derive a bound for the respective other task, up to constant factors. A slightly weaker version of the proofs was published in [GJ10b]. We will use this equivalence throughout the thesis several times to derive upper and lower bounds for both tasks. The equivalence is shown by describing how a program for one of the tasks can be transformed into one for the respective other task.

For the sake of illustration, it will be useful to describe the computation graph of a program as introduced in Definition 2.2. We consider a slightly modified version where nodes with multiple outgoing edges are replaced by nodes with out-degree 1, followed by a copy-node with higher out-degree. These computation graphs reflect the computation operations the I/O- and the PEM model are capable of, according to their definition in Section 1.2.1. We distinguish according to their corresponding operation between sum-nodes, product-nodes and copy-nodes. Sum- and product-nodes are restricted to an out-degree of at most 1. Copy-nodes have exactly one ingoing edge, and an arbitrary number of outgoing edges. This implies that we require every computation operation to delete its operands. A program can be transformed to fulfil this condition by performing an additional copy operation for every operand that shall not be deleted before the computation operation. Since we consider only computation operations involving two operands, this can be achieved without changing the number of I/Os when allowing an internal memory of size $M + 2$. Note that this modification is not a strict requirement, but it eases the transformation without changing the asymptotics in our results.

For the proofs in this chapter, we consider normalised programs according to Chapter 2. Recall that in a normalised program, all intermediate results are predecessors of an output-record. Moreover, we add the following normalisation here: A normalised program does not contain multiplications where one operand is a record created by the operation $m_i := 1$. Since 1 is the neutral element of multiplication, operations that multiply by 1 independently from the input can be removed, which can only reduce the number of I/Os.

The restriction to elements from a semiring allows us to characterise all intermediate results as polynomials over the input-records. Let p_r be the polynomial that describes the record r. The absence of inverse elements implies that the set of variables in p_r is a subset of the variables in p_s for any successors s of r. Similarly, if some product $a \cdot b$ is part of one of the monomials in p_r, then there will be a monomial in p_s that contains $a \cdot b$ for all successors s of r. Hence, for a normalised program for BIL, any directed path in the computation graph contains at most two product-nodes because $z^{(i)} = \sum_{jk} x_k^{(i)} a_{jk} y_j^{(i)}$.

For the transformation, we consider programs in time-inverse execution, exchanging the roles of input and output operations. While this is easy to handle in the single processor I/O-model, in the PEM model, asymmetric access permissions like CREW can present a problem. Therefore, we first assume a parallel program for the transformation to be specified either for the EREW policy, or for CRCW where a concurrent write magically adds up all the elements that are to be written to the same record. We consider a CREW policy later on. The main result of this chapter is the following theorem.

Theorem 3.1. *Let* $\mathbf{A}$ *be a matrix, given in a fixed layout, and* w *pairs of vectors* $\mathbf{x}^{(i)}$, $\mathbf{y}^{(i)}$, $1 \leq i \leq w$. *The evaluation of* BIL *for* w *bilinear forms* $\mathbf{y}^{(i)T}\mathbf{A}\mathbf{x}^{(i)}$, $1 \leq i \leq w$, *has the same asymptotic (parallel) I/O complexity as evaluating* SPMV *for the* w *matrix vector products* $\mathbf{A}\mathbf{x}^{(i)}$.

The proof of one direction of the equivalence in Theorem 3.1 is straightforward and given in the next lemma. Note that the layout of the matrix $\mathbf{A}$ is not of concern, it just has to be the same for both tasks.

Lemma 3.2. *If* SPMV *can be computed for a matrix* $\mathbf{A}$ *and* w *vectors* $\mathbf{x}^{(i)}$, $1 \leq i \leq w$, *with* ℓ *(parallel) I/Os, then* BIL *can be evaluated for* $\mathbf{A}$ *and* w *vector pairs* $\mathbf{y}^{(i)}$, $\mathbf{x}^{(i)}$ *with* $\mathcal{O}(\ell + \log N_y/B)$ *(parallel) I/Os.*

Proof. For each vector pair $\mathbf{x}^{(i)}, \mathbf{y}^{(i)}$, the bilinear form is computed by multiplying $\mathbf{y}^{(i)}$ with the corresponding result vector $\mathbf{c}^{(i)} = \mathbf{A}\mathbf{x}^{(i)}$. Even if $\mathbf{c}^{(i)}$ contains empty records, only the blocks of $\mathbf{y}^{(i)}$ that correspond to non-empty blocks of $\mathbf{c}^{(i)}$ need to be accessed for multiplication. Because the algorithm at least wrote each $\mathbf{c}^{(i)}$, this scanning of $\mathbf{y}^{(i)}$ for each $1 \le i \le w$ takes certainly no more than an additional ℓ I/Os. The non-empty blocks of all $\mathbf{c}^{(i)}$ get assigned equally to the (first) $\min\{wN_{\mathrm{y}}/B, P\}$ processors such that $\min\{N_{\mathrm{y}}/B, P/w\}$ processors are assigned to the same vector pair. Then, each processor reads its assigned blocks together with the corresponding blocks of $\mathbf{y}^{(i)}$, and creates the (partial) scalar products. The partial results of the $\min\{N_{\mathrm{y}}/B, P/w\}$ processors that are assigned to the same task then use a gather operation (cf. Section 2.7.1) to create the output. This induces $\mathcal{O}(\log\min\{N_{\mathrm{y}}/B, P/w\} + 1)$ I/Os. □

To show the other direction of Theorem 3.1 is more involved. We discuss in the remainder of this chapter how a program for BIL can be transformed into one for SPMV. In this transformation, the operations that create elementary products play an important role. We say that an operation (it must be a multiplication) creates an elementary product, if the elementary product is one of the monomials of the result, but not part of any monomial in the direct predecessors. One multiplication can create many elementary products, but the total number of such operations is limited by the total number of elementary products. This relies upon the following Lemma, which is easy to prove.

Lemma 3.3. *In a normalised semiring I/O program for* BIL *on multiple vector pairs, no elementary product is created twice.*

Proof. In a normalised program, every elementary product that is created, is contained in one of the w output records. If two intermediate results that both contain the same elementary product are added or multiplied, all the successors contain this elementary product twice or squared. Recall that the polynomial constituting the ith output record is of the form $\sum_j \sum_k y_j^{(i)} a_{jk} x_k^{(i)}$. Hence, neither the square of an elementary product can be a predecessor of an output record, nor can an elementary product be contained twice in a polynomial constituting an output record. A normalised program therefore never creates the same elementary product twice. □

Lemma 3.4. *If* BIL *can be evaluated for a matrix* $\mathbf{A}$ *and* w *vector pairs* $\mathbf{y}^{(i)}, \mathbf{x}^{(i)}$, *with internal memory size* M *and block size* B *using* ℓ *(parallel) I/Os, then* SPMV *can be computed for* $\mathbf{A}$ *and the* w *vectors* $\mathbf{x}^{(i)}$ *using* 5ℓ *(parallel) I/Os with internal memory size* $3M + 4$ *and block size* B.

Proof. Given a program to evaluate BIL for w bilinear forms using ℓ I/Os, the program can be transformed into a program with ℓ I/Os and computation operations that delete their operands for the semiring I/O-model with internal memory size $M + 2$. By Lemma 3.3, there is a normalised program $\mathcal{P}$ for the same task, which computes only canonical partial results and does not create an elementary product more than once, with at most ℓ I/Os.

Overview The intuition of our approach is the following. In a first phase, $\mathcal{P}$ is transformed into a program $\hat{\mathcal{P}}$ where each block in external memory that results from an output operation of $\hat{\mathcal{P}}$ will never be removed / overwritten. Furthermore, we divide $\hat{\mathcal{P}}$ into rounds of M/B I/Os. After each round, the content of internal memories is copied to disk to produce an image of internal memory at the end of the round. Note that in contrast to other Hong Kung like round arguments, we do not empty internal memory between rounds.

These outputs / intermediate results of $\mathcal{P}$ can be used in a second phase to create the matrix vector product. The program $\hat{\mathcal{R}}$ for the second phase assumes internal memories of size $3M+4$. It also operates in rounds where each round reflects a round of $\hat{\mathcal{P}}$. A round of $\hat{\mathcal{P}}$ by a processor can be simulated in $\hat{\mathcal{R}}$ by loading the at most $2M/B$ blocks that where accessed by the processor in $\hat{\mathcal{P}}$ during this specific round. This supplies $\hat{\mathcal{R}}$ with all the records that are in internal memory at some point in time within the round of the processor in $\hat{\mathcal{P}}$. Hence, the same computations performed by $\hat{\mathcal{P}}$ in this round can be performed by $\hat{\mathcal{R}}$. However, the ordering of the simulated rounds in $\hat{\mathcal{R}}$ is inverse to the ordering of the rounds in $\hat{\mathcal{P}}$.

To this end, in each processor $2M$ internal memory records are reserved in $\hat{\mathcal{R}}$ to keep the unmodified records of $\hat{\mathcal{P}}$ throughout a round. Another $M+4$ records are used by the following program. A time-inverse variant of $\mathcal{P}$ leads intermediate results that belong to $\mathbf{c}^{(i)} = \mathbf{A}\mathbf{x}^{(i)}$ towards the initial position of $\mathbf{y}^{(i)}$ on disk. Recall that the polynomial describing the result $z^{(i)} = \mathbf{y}^{(i)T}\mathbf{A}\mathbf{x}^{(i)}$ is a sum of elementary products $y_j^{(i)} a_{j,k} x_k^{(i)}$. To create each of these elementary products, one copy of some $y_j^{(i)}$ is involved in the computation at some point. The copy process of $y_j^{(i)}$ records in $\mathcal{P}$ can hence be used in the time-inverse program, to lead elementary products and partial sums to the initial positions of $y_j^{(i)}$ (cf. Figure 3.1).

Phase 1 The program $\hat{\mathcal{P}}$ simulates $\mathcal{P}$ where $\hat{\mathcal{P}}$ remaps the block indices of input and output operations of $\mathcal{P}$ as follows to avoid overwriting operations. Let $(H + N_{\mathsf{x}} + N_{\mathsf{y}})/B =: J$ be the size of the input which we assume to be located contiguously starting at the first block in external memory. If the lth

I/O operation of the pth processor is an output, it is written to the $(J+p\ell+l)$th block. Inputs are mapped accordingly to the altered index of the required block.

After each M/B (parallel) I/Os performed by $\mathcal{P}$, the content of internal memory of each processor is copied to external memory. To avoid conflicts with $\mathcal{P}$, these outputs are written contiguously starting from the $(J+P\ell+1)$th block. We call such a sequence of M/B I/Os in the simulation of $\mathcal{P}$ a **round**. Note that given the blocks that are read by processor p during the ith round of $\mathcal{P}$, and the copy of p's internal memory performed by $\hat{\mathcal{P}}$ at the end of the previous round, all the computations within the ith round of p can be replicated. The program $\hat{\mathcal{P}}$ is executed with input $\mathbf{A}$ and $\mathbf{x}^{(i)}$, $1 \le i \le w$, as given, but all vectors $\mathbf{y}^{(1)} = \cdots = \mathbf{y}^{(w)} = (1, \ldots, 1)$. $\hat{\mathcal{P}}$ performs only I/Os made by $\mathcal{P}$, plus an additional $\frac{M}{B}$ outputs per round. This results in $\ell + \left\lfloor \frac{\ell B}{M} \right\rfloor \cdot \frac{M}{B} \le 2\ell$ I/Os.

Phase 2 In the second phase, a program $\hat{\mathcal{R}}$ is executed which simulates $\mathcal{P}$ in a time-inverse fashion within the first M records of internal memory. For now, just notice that in this simulated time-inverse program – we denote it by $\mathcal{R}$ – the ordering of operations is reversed. We explain in the next paragraph how operations have to be adapted to yield a correct program. For the ith operation ρ_i^p of processor p in $\mathcal{P}$, we denote the corresponding operation in $\mathcal{R}$ by $\overline{\rho}_i^p$. According to the partition of $\mathcal{P}$ into rounds, we consider $\mathcal{R}$ partitioned into rounds of M/B I/Os backwards, starting with the last I/O. Hence, rounds are enumerated in reverse order in $\mathcal{R}$. In each round of $\hat{\mathcal{R}}$, the blocks that are accessed during the round in $\mathcal{P}$ are buffered in $2M$ records of internal memory, denoted as $\mathcal{M}_{\mathcal{R}}$. Before the first operation of the ith round of processor p in $\mathcal{R}$, all blocks that are input by p during round i in $\mathcal{P}$ are loaded into $\mathcal{M}_{\mathcal{R}}$. For $i > 1$, additionally the image of internal memory after the previous round in $\hat{\mathcal{P}}$ is loaded into $\mathcal{M}_{\mathcal{R}}$. This results in at most $3M/B$ I/Os per round.

Observe that any record of $\mathcal{P}$ that is involved in operation ρ_i^p can be replicated from the provided records when $\overline{\rho}_i^p$ is performed in $\mathcal{R}$. To this end, we reserve some extra space in internal memory for the replicating computations. Let m_s be the record involved in ρ_i^p which shall be replicated, and $r(\rho_i^p)$ be the round containing ρ_i^p. Hence, m_s stems either from an input during $r(\rho_i^p)$, it was in memory at the beginning of $r(\rho_i^p)$, or it was generated by a computation operations during $r(\rho_i^p)$. In the first two cases, m_s is present in $\mathcal{M}_{\mathcal{R}}$ when $\overline{\rho}_i^p$ is performed. If m_s was generated by a computation operation, the same three choices hold in turn for the origin of the operands. Since a round contains a limited number of operations, there is a set of predecessors of m_s in $\mathcal{M}_\mathcal{R}$ which is sufficient to compute m_s. Recall that each record describes a polynomial of maximum degree three. The replication of m_s in $\hat{\mathcal{R}}$ can be performed with two additional records in internal memory: One record to keep a partial sum of the polynomial, and a second record to write (and add) the results of elementary products.

In the following, we describe how operations of $\mathcal{P}$ are adapted in $\mathcal{R}$. To this end, we introduce the following notation. If an operation in $\mathcal{P}$ accesses a record m, we denote the corresponding counterpart record in $\hat{\mathcal{R}}$ by $\overline{m}$. To simplify $\mathcal{R}$, we remove all records $\overline{m}$ throughout $\mathcal{R}$ where the record m in $\mathcal{P}$ does not describe a polynomial containing any $y_j^{(i)}$.

Naturally, an input translates into an output when time is inverted, and vice versa. However, if not in a magically adding CRCW environment, we have to consider this more in detail since records are copied and not moved. First of all, when performing an output operation ρ in $\mathcal{P}$, the output records are allowed to stay in internal memory. This introduces implicit copy operations that have to be considered. To tackle this problem in $\hat{\mathcal{R}}$, we allow one extra block in internal memory where $\overline{\rho}$ performs the input to. Then, each record $\overline{m}$ that is read by $\overline{\rho}$ is added to the position where m was located before ρ if m was not deleted/overwritten immediately after ρ. Otherwise, it is copied to the position of m. Recall that in a normalised program, all records that are not a predecessor of an output-record are removed immediately after an I/O. A similar problem is caused by the fact that a block can be read by $\mathcal{P}$ multiple times. This is handled in the following way. Let t_i be the block that is input in $\mathcal{P}$, and let $m_{j_1}, \ldots, m_{j_B}$ be the records that are input. In $\mathcal{R}$, the records of the corresponding block $\overline{t}_i$ where the output is performed to are loaded into the additional block of internal memory. Then, each record $\overline{m}_j$ is added to the location of m_j in $\mathcal{P}$ if it is not immediately removed after the input, and it is copied otherwise. Afterwards, the results are written back to $\overline{t}_i$. This increases the number of I/Os by a factor of at most 2.

Now consider the transformation of computation operations. Each copy operation of $\mathcal{P}$ that sets $m_r := m_s$ is replaced by a sum operation $\overline{m}_s := \overline{m}_s + \overline{m}_r$ in $\mathcal{R}$. Each sum operation $m_q := m_r + m_s$ is replaced by a copy operation $\overline{m}_r := \overline{m}_s := \overline{m}_q$ in $\mathcal{R}$. Operations in $\mathcal{P}$ that set a record to 0 or 1 are simply ignored in $\hat{\mathcal{R}}$, i.e. nothing has to be created. Multiplication operations are more involved to deal with since they serve to lead elementary products to the initial position of the $\mathbf{y}^{(i)}$ records in $\mathcal{R}$. We define the transformation here and explain its effects in the subsequent paragraphs. Let ρ be a multiplication operation in $\mathcal{P}$ of the form $m_q := m_r \cdot m_s$ where m_r is a polynomial containing $y_j^{(i)}$. Recall that we can replicate the record m_s in $\mathcal{R}$. The operation $\overline{\rho}$ performs the multiplication $\overline{m}_r := m_s \cdot \overline{m}_q$ after replicating m_s.

According to this construction, $\hat{\mathcal{R}}$ is executed where each of the inputs $\overline{z}^{(i)}$ for $\mathcal{R}$ is set to 1. This finishes the second phase and the result vector $\mathbf{c}^{(i)}$ is stored at the initial positions of $\mathbf{y}^{(i)}$ in external memory for $1 \le i \le w$.

Correctness To illustrate the modifications in $\mathcal{R}$, we consider the computation graphs $G_{\mathcal{P}}$ of $\mathcal{P}$ and $G_{\mathcal{R}}$ of $\mathcal{R}$. Note that in a normalised program canonical partial results never contain input-records from different vector pairs. Hence, it suffices to consider the operations for each vector separately.

In the following, we denote a record $\overline{m}_r$ in $\hat{\mathcal{R}}$ a $y_j^{(i)}$**-container** (or simply y**-container**) if $m_r = y_j^{(i)}$ in $\hat{\mathcal{P}}$. Note that m_r has not been involved in any algebraic operation before. The record m_r can result only from an initial input of $y_j^{(i)}$ or be a copy of it. Hence, $\overline{m}_r$ will be summed up with other results (possibly 0) in $\hat{\mathcal{R}}$. If an elementary product is written into a $y_j^{(i)}$-container in $\hat{\mathcal{R}}$, it will eventually be contained as a summand in the input-record $y_j^{(i)}$ in external memory. It is thus our goal to show that all elementary products of the form $a_{jk}x_k^{(i)}$ are written into a $y_j^{(i)}$-container.

Observe that on every directed path from $y_j^{(i)}$ to $z^{(i)}$ in $G_{\mathcal{P}}$, there is at least one multiplication operation. Otherwise, $y_j^{(i)}$ appears as a monomial in the polynomial describing $z^{(i)}$ which contradicts that $\mathcal{P}$ is a correct program for BIL. In $\hat{\mathcal{R}}$, a sum operation sums up several y-containers, and writes the result into a y-container. The only operation that writes into a y-container without having y-containers as its operands is a transformed multiplication operation. We will show in the following that indeed all the elementary products $a_{jk}x_k^{(i)}$ for non-zero entries a_{jk} will be written into a $y_j^{(i)}$-container, and that no other elements are introduced into a $y_j^{(i)}$-container. To this end, we consider all the possible multiplications ρ that set $m_q := m_r \cdot m_s$ and their corresponding counterpart $\overline{\rho}$. We consider the operations first where $\overline{\rho}$ writes into a y-container. Recall that we eliminated records that do not contain some $y_j^{(i)}$ so that operations that multiply $a_{j,k}$ with $x_k^{(i)}$ can be ignored.

Let $m_s = \sum a_{jk}x_k^{(i)}$, and hence $m_r = y_j^{(i)}$. By construction, the operation $\overline{\rho}$ performs the multiplication $\overline{m}_r := m_s \cdot \overline{m}_q$. There is no multiplication operation succeeding ρ since otherwise a product of four variables would be contained in the polynomial describing $z^{(i)}$. Since there is no multiplication succeeding ρ, $\overline{m}_q$ is a copy of $z^{(i)} = 1$ in $\hat{\mathcal{R}}$. Hence, $m_s = \sum a_{jk}x_k^{(i)}$ is written into the $y_j^{(i)}$-container $\overline{m}_r$ (cf. Figure 3.1, upmost transfer).

Now consider the case $m_s = a_{jk}$ so that m_r is either $y_j^{(i)}$ or $y_j^{(i)}x_k^{(i)}$. In the first case, ρ must be succeeded by a multiplication operation ρ' that involves $x_k^{(i)}$. More precisely, either $m_q = y_j^{(i)}a_{jk}$ is directly multiplied with $x_k^{(i)}$, or a sum $\sum y_j^{(i)}a_{jk}$ containing m_q is multiplied with $x_k^{(i)}$. Let m_t be the record that is multiplied with $x_k^{(i)}$ in ρ'. Note that in both cases m_t is a successor

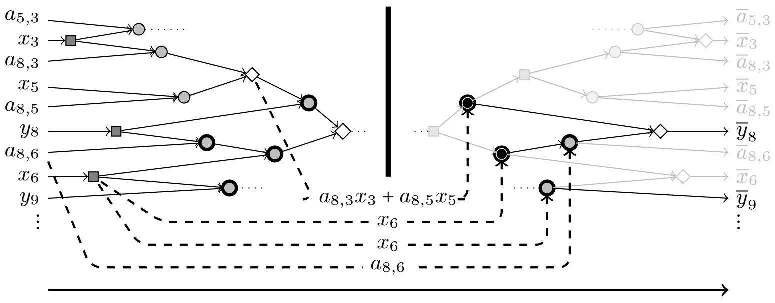


Figure 3.1: The computation graph of the transformed program. Squares are copy-nodes, circles are product-nodes, and diamonds are sum-nodes. The left-hand side illustrates $\mathcal{P}$, the right-hand side the time-inverse $\mathcal{R}$. Dashed lines represent replicated records involved in multiplication operations of $\mathcal{R}$.

of m_q. Furthermore, there is no other multiplication operation succeeding ρ'. Hence, $x_k^{(i)}$ is written – multiplied by 1 – into record $\overline{m}_t$ by $\overline{\rho}'$. The record $\overline{m}_t$ is a predecessor of $\overline{m}_q$ implying that $\overline{m}_q$ is equal to, or a copy of $\overline{m}_t$ (recall that ρ and ρ' are the only multiplication operations on this directed path in $G_{\mathcal{P}}$). The record $\overline{m}_q = x_k^{(i)}$ is then multiplied with $m_s = a_{jk}$, and written into the $y_j^{(i)}$-container $\overline{m}_r$ by $\overline{\rho}$ (cf. Figure 3.1, lowest transfer). The second case, i.e. $m_r = y_j^{(i)}x_k^{(i)}$, is covered implicitly in the next paragraph.

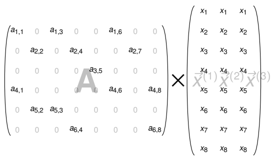
In the last case, we assume $m_s = x_k^{(i)}$ which means that m_r is either $y_j^{(i)}$ or $\sum y_j^{(i)}a_{jk}$. The second case is already covered by the previous paragraph: For each summand in $\sum y_j^{(i)}a_{jk}$, there must be a preceding multiplication. This corresponds to the case considered in the previous paragraph. The first case instead, can be shown analogously: There is a succeeding multiplication operation ρ' that involves a_{jk} in $\mathcal{P}$. Hence, the operation $\overline{\rho}'$ provides a_{jk} as a predecessor of $\overline{m}_q$. In $\overline{\rho}$, the record $\overline{m}_q = a_{jk}$ is multiplied by $x_k^{(i)}$ and written into the $y_j^{(i)}$-container $\overline{m}_r$.

This case distinction shows that only elementary products $a_{jk}x_k^{(i)}$, or partial sums $\sum_k a_{jk}x_k^{(i)}$ are written into a $y_j^{(i)}$-container. Furthermore, every elementary product $y_j^{(i)}a_{jk}x_k^{(i)}$ has exactly one input-record $y_j^{(i)}$ as a predecessor in $\mathcal{P}$. Thus, for each elementary product $y_j^{(i)}a_{jk}x_k^{(i)}$ that is produced in $\mathcal{P}$, the elementary product $a_{jk}x_k^{(i)}$ is added to the polynomial describing the final record $y_j^{(i)}$ in external memory in the time-inverse program $\hat{\mathcal{R}}$. Since all the wH elementary products $y_j^{(i)}a_{jk}x_k^{(i)}$ have to be created for BIL, the created vectors $\mathbf{y}^{(i)} = \mathbf{A}\mathbf{x}^{(i)}$, $1 \le i \le w$, are correct. □

Observation 3.5. *Using Lemma 3.4, a non-uniform program for* SPMV *in CREW* *models can be obtained with an additional ℓ I/Os.*

Proof. Depending on the processor, all outputs of $\hat{\mathcal{R}}$ are performed to separate sections on disk such that no concurrent write is required. Using this transformation, there will be P partial outputs in the end of the program. In a non-uniform setting, the processors can then be assigned to the output blocks to perform a parallel sum operation in order to yield the result vectors. Let m be the overall number of the output blocks in this separated program. We assign the m blocks evenly, non-uniformly to the P processors such that for $P < m$, blocks containing partial results from the same row are assigned to as few processors as possible. Afterwards, with a gather operation according to Section 2.7.1, the blocks can be summed together to form the final output blocks. Since each of the m output blocks was written the original number of I/Os of $\mathcal{R}$ is $\ell \geq m/P$. □

To finish the proof of Theorem 3.1, it remains to show that the asymptotical I/O complexity of the tasks does not change, when altering M by constant factors. We show this by presenting upper and lower bounds in Chapter 4.



4 Multiple Vectors

4.1 Introduction

In this chapter, we consider the problem of computing $w < B$ matrix vectors products $\mathbf{c}^{(i)} = \mathbf{A}\mathbf{x}^{(i)}$, $1 \leq i \leq w$, where $\mathbf{A}$ is an $N_\mathsf{y} \times N_\mathsf{x}$ matrix with H non-zero entries, and $\mathbf{x}^{(1)}, \ldots, \mathbf{x}^{(w)}$ are dense vectors. The results presented here are based on [GJ10a, GJ10b]. However, we improve on some of these results here. Furthermore, the results are extended to the PEM model. In Chapter 3, we showed that the (non-uniform) I/O complexities of SPMV and BIL differ in our model only by constant factors and an additive term $\mathcal{O}(\log N_\mathsf{y}/B)$. Hence, we derive upper and lower bounds for both tasks. Recall that BIL has the additional input $\mathbf{y}^{(1)}, \ldots, \mathbf{y}^{(w)}$ and asks for the computation of $z^{(i)} = \mathbf{y}^{(i)T}\mathbf{A}\mathbf{x}^{(i)}$ for $1 \leq i \leq w$. While all bounds are uniform for BIL, for SPMV some of the algorithms presented in this chapter remain non-uniform (adapted to the conformation of the matrix) for load-balancing reasons. A uniform algorithm can be obtained when including a preprocessing step to determine the conformation of the matrix.

Bender, Brodal, Fagerberg, Jacob and Vicari [BBF+07,BBF+10] determined the I/O complexity of computing SPMV for a square matrix with a single vector in the I/O-model. For the number of non-zero entries $H \leq N^{2-\varepsilon}$ with matrix dimension N they present upper and lower bounds that match up to constant factors for worst-case and column major layout, and for the best-case layout given $H \leq N^{1+1/3}$. These results are generalised here in the following ways: (i) the dimensions of the matrix (and thus the dimension of the vectors) are relaxed to arbitrary, *non-square* situations, (ii) the product of one matrix with *several vectors*, performed *simultaneously*, is considered, (iii) the parameter range is extended to cover *all ranges* of H (from sparse up to dense matrices), (iv) the complexity is determined in the *PEM model*. For most of the considered parameters, we present lower bounds and upper bounds in form of algorithms that match up to constant factors.

The results from Chapter 3 allow to extend all upper and lower bounds for column major layout to row major layout. Given $\mathbf{A}$ in row major layout, we can reduce $\mathbf{Ax}$ to $\mathbf{y}^T\mathbf{Ax}$ using Chapter 3. Observe that $\mathbf{y}^T\mathbf{Ax} = (\mathbf{x}^T\mathbf{A}^T\mathbf{y})^T$, so that both tasks have the same complexity. Again by Chapter 3, $(\mathbf{x}^T\mathbf{A}^T\mathbf{y})^T$ can be reduced to $\mathbf{A}^T\mathbf{y}$ which is a matrix vector product with a matrix in column major layout. The algorithms can be transformed in a similar way. Hence, all bounds for column major layout yield respective bounds for row major layout in which only the dimensions N_{x} and N_{y} need to be swapped.

Considering the evaluation of matrix vector products for $w < B$ vectors is a step towards sparse matrix dense matrix multiplication since the task is to create the product $\mathbf{Y} = \mathbf{A} \cdot \mathbf{X}$ where $\mathbf{X}$ is a dense $N_{\mathrm{x}} \times w$ matrix. Note that the vectors are assumed to be given as contiguous records in memory. Thus, the matrix $\mathbf{X}$ constituted by the vectors is always given in column major layout, and the matrix $\mathbf{Y}$ has to be output in column major layout. It is important to consider the case $w < B$ separately from $w \geq B$ because in the former case, independently of the layout of $\mathbf{X}$, a block always contains records from multiple rows of $\mathbf{X}$ (since a row contains less than B records). This is not the case if $\mathbf{X}$ is a matrix in row major layout with more than B columns. Multiplying a sparse matrix with a dense matrix with more than B columns is considered in Chapter 5.

On the algorithmic side, it will turn out that it can be useful to change the layout of $\mathbf{A}$ into a best-case layout when performing multiple products. Moreover, for very asymmetric situations, and also for rather dense matrices, a class of algorithms becomes relevant where tuples of input vector entries are generated initially. For the lower bounds, it is worth noting that the lower bound for best-case layouts of [BBF+10] needs to be combined with an argument originating from the considerations by Hong and Kung [HK81], in order to derive matching bounds for multiple vectors. To this end, the number of records of a specific type within a sequence of M/B (parallel) I/Os is considered. In order to achieve a lower bound for the PEM model, we adapt this argument in comparison to the published results for the I/O-model in [GJ10a, GJ10b].

We make use of the results of Chapter 3 by stating some of the algorithms for matrix vector products and some for the evaluation of bilinear forms. This simplifies some expositions. The transformations in Chapter 3 can be used to derive an algorithm for the respective other task. However, all the presented algorithms are simple enough to allow for a direct transformation. In any case, we were not able to present efficient uniform algorithms for all parameter ranges in the CREW PEM model for SPMV because of load-balancing reasons. While all our lower bounds hold for the non-uniform I/O complexity, thus implying a lower bound for uniform programs, the table-based and direct algorithms in Section 4.2 are uniform only for BIL. If the records of the matrix are annotated with their ranking in a row major layout, also for SPMV a uniform program can be stated. The problem of non-uniform algorithms, however, appears only in the parallel case where a proper load-balancing between processors is required.

The following paragraphs give an overview over the I/O-complexities that are achieved by the algorithms presented in Section 4.2, and bounded by the lower bounds in Section 4.3. For some restrictions on the parameter space, we obtain upper and lower bounds matching up to constant factors.

Results for Column Major Layout

Theorem 4.1. *Given an $N_y \times N_x$ matrix* $\mathbf{A}$ *with H non-zero entries in column major layout and PEM parameters P, M, and B where $M \geq 4B$. Evaluating w instances of* SPMV *or* BIL *simultaneously for* $\mathbf{A}$ *over an arbitrary semiring has (worst-case) parallel I/O complexity*

$$\mathcal{O}\left(\min\left\{\frac{H\log\min\left\{\frac{N_y}{B},\frac{N_xN_y}{H}\right\}}{P\log\min\left\{\frac{H}{B},N_x\right\}} + \frac{w(N_x+N_y)}{PB}\log_{\mathrm{d}(w(N_x+N_y))} w\,,\right.\right.$$
$$\left.\left.\frac{H}{PB}\log_{\mathrm{d}(H)}\min\left\{\frac{N_y}{B},\frac{N_xN_y}{H}\right\} + \frac{wH}{PB}\log_{\mathrm{d}(wH)}\frac{N_xN_y}{BH}\right\}\right)$$
$$+\,\mathcal{O}\left(\frac{wH}{PB} + \log\min\left\{P,\frac{H}{B}\right\}\right).$$

Recall that $\mathrm{d}(n) = \max\left\{2,\min\left\{\frac{M}{B},\frac{n}{PB}\right\}\right\}$. The first term in the minimum is achieved by an algorithm in which c-tuples of vector records are created initially. This allows for an input of c (non-neighbouring) vector dimensions with one I/O. The second term can be achieved by a sorting approach similar to [BBF+10] where columns of the matrix are merged by a merge sort algorithm. For multiple vectors, it can make sense to change the layout of the matrix to a best-case layout initially. All algorithms are formulated for a restricted number of processors. However, for larger P note that the algorithmic complexity is dominated by the term $\log N_x$.

Theorem 4.2. *Given an $N_y \times N_x$ matrix $\mathbf{A}$ with H non-zero entries in column major layout and PEM parameters P, M, B where $M \geq 4B$. Evaluating w instances of* SpMV *or* BIL *simultaneously for $\mathbf{A}$ with $w \leq B$ over an arbitrary semiring has (worst-case) parallel I/O complexity*

$$\Omega\left(\min\left\{\frac{H \log \min\left\{\frac{N_y}{B}, \frac{N_x N_y}{H}\right\}}{P \log H}, \frac{H}{PB} \log_{\mathrm{d}(H)} \min\left\{\frac{N_y}{B}, \frac{N_x N_y}{H}\right\} + \frac{wH}{PB} \log_{\mathrm{d}(wH)} \frac{N_x N_y}{BH}\right\} + \log \min\left\{\frac{H}{B}, N_x\right\}\right).$$

The lower bounds are obtained by a modification of the proof in [BBF⁺10] for a single matrix vector product, also keeping track of the different matrix dimensions. The bounds are extended to the PEM model using the considerations from Section 2.2 Additionally, the lower bounds of Theorem 4.5 for best-case layout apply which yield the second term of the sum in Theorem 4.2. Finally, a lower bound of $\Omega\left(\log \min\left\{\frac{H}{B}, N_x\right\}\right)$ for gather operations as described in Section 2.4 applies. Note again that this dominates the bound in Theorem 4.2 for any $P > \frac{H}{B}$.

For the following parameter ranges, we obtain asymptotically matching upper and lower bounds.

Lemma 4.3. *Let $M \geq 4B^{1+\varepsilon}$ (tall cache), $P \leq wH/B^{1+\varepsilon}$, and $B^{1+\varepsilon} \leq N_y \leq N_x^c$ for constant $\varepsilon > 0$ and constant c. Then, evaluating $w \leq B$ matrix vector products over an arbitrary semiring has (worst-case) parallel I/O complexity*

$$\Theta\left(\min\left\{\frac{H}{P} \log_{N_x} \frac{N_x N_y}{H}, \frac{wH}{PB} \log_{\mathrm{d}(wH)} \frac{N_x N_y}{H}\right\} + \log N_x\right) \tag{4.1}$$

if

$$H \leq \min\left\{\frac{N_x N_y}{M}, (N_y N_x)^{\frac{1}{1+\xi \frac{w}{B}}}\right\}$$

for constant $\xi > 0$.

Proof. First, observe that the lower bound from Theorem 4.2 is $\Omega\left(\frac{wH}{PB}\right)$ in case $H \leq \min\left\{N_x N_y / M, N_y N_x^{1-\xi w/B}\right\}$: Because $N_y \geq B^{1+\varepsilon}$, we have $\log N_y / B \geq \varepsilon \log N_y$. This reduces the first term of the minimum since $H \geq N_x$, and we can further estimate

$$\frac{H}{P} \log_H \frac{N_x N_y}{H} \geq \frac{H}{P} \log_H H^{\xi w/B} \geq \xi \frac{wH}{PB}.$$

Similarly, the second term in Theorem 4.2 can be bounded

$$\frac{wH}{PB}\log_{\mathrm{d}(wH)}\frac{N_{\mathsf{x}}N_{\mathsf{y}}}{BH} \geq \frac{wH}{PB}\log_{\mathrm{d}(wH)}\frac{M}{B} \geq \xi\frac{wH}{PB}$$

since $\mathrm{d}(wH) \leq M/B$. Furthermore, assuming $N_{\mathsf{y}} \leq N_{\mathsf{x}}^c$ yields

$$\log H \leq 2\log\max\{N_{\mathsf{y}}, N_{\mathsf{x}}\} \leq 2c\log N_{\mathsf{x}}$$

and, by using $H \geq N_{\mathsf{y}} \geq B^{1+\varepsilon}$, we obtain

$$\log\frac{H}{B} \geq \log H - \frac{1}{1+\varepsilon}\log H \geq \frac{\varepsilon}{2}\log H \geq \frac{\varepsilon}{2}\log N_{\mathsf{x}}$$

which together yields the base of the logarithm in the first term of the minimum in (4.1). By similar arguments, we have $\log H/B \leq 2c\log N_{\mathsf{x}}$ yielding the last term. Finally, observe that for $M \geq 4B^{1+\varepsilon}$ and $P \leq wH/B^{1+\varepsilon}$, it holds

$$\log_{\mathrm{d}(wH)} w \leq \log_{\mathrm{d}(wH)} B \leq \frac{\log B}{\log B^{\varepsilon}} \leq 1/\varepsilon$$

since $\mathrm{d}(wH) = \max\left\{2, \min\left\{\frac{M}{B}, \frac{wH}{PB}\right\}\right\}$. Using this, $\frac{w(N_{\mathsf{x}}+N_{\mathsf{y}})}{PB}\log_{\mathrm{d}(w(N_{\mathsf{x}}+N_{\mathsf{y}}))} w \leq \frac{wH}{PB}\log_{\mathrm{d}(wH)} w \leq \frac{1}{\varepsilon}\frac{wH}{PB}$. Additionally, we can ignore parameters at most B in the logarithm of the second term in the minimum.

□

Results for Best-Case Layout

Theorem 4.4. *Given an $N_y \times N_x$ matrix $\mathbf{A}$ with H entries in best-case layout and PEM parameters P, M, B where $M \geq 4B$. W.l.o.g. let $N_y \leq N_x$. Evaluating $w \leq B$ matrix vector products with $\mathbf{A}$ over an arbitrary semiring has (worst-case) parallel I/O complexity*

$$\mathcal{O}\left(\min\left\{\frac{H\log\left(\frac{w^2N_xN_y}{HB\min\{M,H/P\}}\log\frac{N_x}{\min\{M,H/P\}}\right)}{P\log\frac{N_x}{\min\{M,H/P\}}} + \frac{wN_x}{PB}\log_{\mathrm{d}(wN_x)} w\,,\right.\right.$$
$$\left.\left.\frac{wH}{PB}\log_{\mathrm{d}(wH)}\frac{N_xN_y}{HB}\right\}\right) + \mathcal{O}\left(\frac{wH}{PB} + \log\min\left\{P, \frac{H}{B}\right\}\right).$$

The algorithms are similar to the ones for column major layout. A simple extension of the sorting algorithm in [BBF⁺10] yields the first term of the minimum. Similar to [BBF⁺10], the optimal layout consists of meta-columns of width m which are internally ordered row-wise. For the parallel case, we use $m = \max\{B, \min\{M - B, \lceil H/P \rceil\}\}$. The first term of the minimum is obtained by an algorithm that creates tuples of vector records. For this algorithm, an optimal layout is given by a partition of the matrix into meta-columns of even larger width.

Theorem 4.5. *Given an $N_y \times N_x$ matrix $\mathbf{A}$ with H entries in best-case layout and PEM parameters P, M, B. Let $M \geq 4B$, $w \leq B$, and w.l.o.g. $N_y \leq N_x$. Evaluating w matrix vector products with $\mathbf{A}$ over an arbitrary semiring has (worst-case) parallel I/O complexity*

$$\Omega\left(\min\left\{\frac{H \log\left(\frac{wN_xN_y}{HB\min\{M, wH/P\}} \log H\right)}{P \log H}, \frac{wH}{PB} \log_{\mathrm{d}(wH)} \frac{N_xN_y}{HB}\right\} + \log \frac{N_x}{B}\right).$$

The lower bound for best-case layout in [BBF+10] can be adapted straightforward to non-square matrices. However, the proof does not extend directly to the consideration of multiple vectors. To tackle this, only the vector $\mathbf{x}^{(i)}$ that contributes the least I/O volume within a program is considered. Again, Section 2.4 yields a lower bound of $\Omega\left(\log \frac{N_x}{B}\right)$.

The second part of the minimum in Theorem 4.5 is especially interesting for iterative multiplications of the form $\mathbf{x}^{(i+1)} = \mathbf{A}\mathbf{x}^{(i)}$. For any program, that writes each (intermediate result) $x_j^{(i)}$ at some time to external memory, the following insight is obtained: For the case $\frac{w}{B} \log_{M/B} H \leq 1$, it is optimal to generate the complete vectors $\mathbf{x}^{(i)}$ one after another. Hence, in this setting it is not of importance whether all vectors are given initially, or derived throughout the computation.

Lemma 4.6. *Assuming $M \geq 4B^{1+\varepsilon}$ (tall cache), $P \leq wH/B^{1+\varepsilon}$, $N_x \geq B^{1+\varepsilon}$ for constant $\varepsilon > 0$ and w.l.o.g. $N_y \leq N_x$, evaluating $w \leq B$ matrix vector products over an arbitrary semiring has (worst-case) parallel I/O complexity*

$$\Theta\left(\min\left\{\frac{H}{P} \log_{N_x} \frac{N_xN_y}{H}, \frac{wH}{PB} \log_{\mathrm{d}(wH)} \frac{N_xN_y}{H}\right\} + \log N_x\right).$$

if $H \leq \left(\frac{N_xN_y}{\min\{M, wH/P\}}\right)^{\frac{1}{1+\xi\frac{w}{B}}}$ for constant $\xi > 0$.

Proof. We start again by showing that the lower bound, given by Theorem 4.5, is $\Omega\left(\frac{wH}{PB}\right)$ for H bounded according to the lemma. Let $m := \min\{M, wH/P\}$. First note, that if the first term in the minimum applies, we have $\log H \geq \frac{B}{w} \log \mathrm{d}(wH)$. Thus, we have

$$\frac{H \log\left(\frac{wN_xN_y}{HBm} \log H\right)}{P \log H} \geq \frac{H \log\left(\frac{N_xN_y}{Hm}\right)}{P \log H} \geq \xi \frac{wH}{PB}$$

by the bound on H. Obviously, the second term is also bounded from below by $\Omega\left(\frac{wH}{PB}\right)$ since we have a stronger bound on H than in Lemma 4.3.

Analogously to Lemma 4.3, the tall-cache assumption and the limit on the number of processors imply that parameters B and w inside the logarithms with base $\mathrm{d}(wH)$, H or N_{x}/m lead only to constant factors. Hence, under the given assumptions, we have an upper bound of

$$\mathcal{O}\left(\min\left\{\frac{H}{P}\log_{\frac{N_{\mathsf{x}}}{m}}\left(\frac{N_{\mathsf{x}}N_{\mathsf{y}}}{H}\log\frac{N_{\mathsf{x}}}{m}\right), \frac{wH}{PB}\log_{\mathrm{d}(wH)}\frac{N_{\mathsf{x}}N_{\mathsf{y}}}{H}\right\} + \log\min\left\{P, \frac{H}{B}\right\}\right)$$

and a lower bound of

$$\Omega\left(\min\left\{\frac{H}{P}\log_H\frac{N_{\mathsf{x}}N_{\mathsf{y}}}{Hm}, \frac{wH}{PB}\log_{\mathrm{d}(wH)}\frac{N_{\mathsf{x}}N_{\mathsf{y}}}{H}\right\} + \log\frac{N_{\mathsf{x}}}{B}\right).$$

Similarly to parameters B and w, the parameter m can be ignored inside the logarithm of the first term of the lower bound: If the minimum in the lower bound corresponds to the first term, then $\log H \geq \frac{B}{w}\log\mathrm{d}(wH)$ holds. Following the proof of Lemma 4.3, it thus holds $\log_H B \leq w/(\varepsilon B)$. Additionally, in this case holds $\log_H m \leq \log_H B\mathrm{d}(wH) \leq w/(\varepsilon B) + 1$. Hence, if $\frac{N_{\mathsf{x}}N_{\mathsf{y}}}{H} < m^2$, the lower bound becomes no stronger than $\Omega\left(\frac{wH}{PB}\right)$. Otherwise, we can ignore the parameter m.

For the upper bound, assume that $\log\frac{N_{\mathsf{x}}}{m}$ is the leading term in the logarithm of the first term, i.e. $\frac{N_{\mathsf{x}}N_{\mathsf{y}}}{H} < \log\frac{N_{\mathsf{x}}}{m}$. By our lower bound of $\Omega\left(\frac{wH}{PB}\right)$, we can infer $\log\log\frac{N_{\mathsf{x}}}{m}/\log\frac{N_{\mathsf{x}}}{m} \geq \epsilon\frac{w}{B}$ for some $\epsilon > 0$. This is equivalent to $\log\frac{N_{\mathsf{x}}}{m}/\log\log\frac{N_{\mathsf{x}}}{m} \leq \frac{B}{\epsilon w}$, so we can estimate $\log\frac{N_{\mathsf{x}}}{m} < \left(\frac{B}{\epsilon w}\right)^2$ which yields again an upper bound of $\mathcal{O}\left(\frac{wH}{PB}\right)$.

For the bases of the logarithms in the respective first terms of the upper and lower bound, consider the case $N_{\mathsf{x}} < m^2$. Since $H \geq N_{\mathsf{x}}$, the first term reduces to $\mathcal{O}\left(\frac{wH}{PB}\right)$. On the other hand, $N_{\mathsf{x}} > m^2$ implies $\log N_{\mathsf{x}}/m \geq \frac{1}{2}\log N_{\mathsf{x}}$ which is hence sufficient in the upper bound. Since we assume furthermore that $N_{\mathsf{x}} \geq N_{\mathsf{y}}$, and we generally have $H \leq N_{\mathsf{x}}N_{\mathsf{y}}$, we can use $\log N_{\mathsf{x}} \geq \frac{1}{2}H$ for the lower bound.

Finally, for $N_{\mathsf{x}} \geq B^{1+\epsilon}$ for constant $\epsilon > 0$, we have $\log(N_{\mathsf{x}}/B) \geq \frac{\varepsilon}{1+\varepsilon}\log N_{\mathsf{x}}$. For the last term of the upper bound, we can estimate $\log(H/B) \leq \log H \leq \frac{1}{2}\log N_{\mathsf{x}}$.

□

4.2 Algorithms

We remark again, that our algorithms yield upper bounds for both bilinear forms and matrix vector products. In general, it is easier to transform our algorithms when stated for bilinear forms towards matrix vector products than by applying the transformation of Chapter 3. However, using the transformation in Chapter 3 yields an algorithm with asymptotically the same I/O complexity for the respective other task. This transformation requires sometimes internal memory size to be larger by a constant factor. Observe that this does not change the asymptotic complexities.

Note that we describe the algorithms for CREW. For a transformation, we required however a CRCW or EREW model. Hence, the algorithms that are stated for BIL yield only CRCW algorithms for SPMV. However, in a non-uniform setting where the algorithm is tuned for the conformation of the matrix, CREW algorithms for SPMV are obtained according to Lemma 3.5. Such an algorithm can be obtained with a pre-processing step, and can hence be optimal if $\mathbf{A}$ is used for several multiplications. Unfortunately, we were not able to obtain uniform algorithms for SPMV for the tasks stated for BIL in the parallel case because of load-balancing reasons.

4.2.1 Direct Algorithm

Though a more general version of this algorithm is explained later on, we describe the straightforward direct algorithm for completeness. When creating SPMV or BIL for a single vector / for a single vector pair only, each elementary product that is required is created by directly accessing its multiplicands, and adding the result to a partial result of the output. Note that this can induce a constant number of I/Os for each non-zero entry because matrix or vector /records are accessed in an irregular pattern. For $w < B$ vectors / vector pairs, the w (complete) elementary products that involve the same non-zero entry a_{ij} can be created with a constant number of I/Os. Thus, the asymptotic worst-case complexity of this algorithm does not change for any $w < B$. Like all the algorithms in this section, this algorithm is optimal for certain parameter ranges. It is described in the following for BIL.

Reading the non-zero entries of $\mathbf{A}$ in an arbitrary order, the computation of $w \leq B$ bilinear forms is possible with

$$\mathcal{O}\left(\frac{H}{P} + \frac{w(N_{\mathsf{x}} + N_{\mathsf{y}})}{PB} \overline{\log}_{\mathrm{d}(w(N_{\mathsf{x}}+N_{\mathsf{y}}))} w + \log \min\left\{\frac{H}{B}, N_{\mathsf{x}}\right\}\right)$$

I/Os. Recall that $\overline{\log}(x)$ is at least 1 and corresponds to $\log(x)$ for $x > 2$. To this end, w records are reserved in internal memory of each processor to contain partial sums of the results $z^{(1)}, \ldots, z^{(w)}$. The records of $\mathbf{A}$ are distributed evenly among the processors, yielding at most $\lceil H/P \rceil$ records per processor. For every assigned non-zero entry a_{jk} the vector records $x_k{}^{(1)}, \ldots, x_k{}^{(w)}$ and

${y_k}^{(1)}, \ldots, {y_k}^{(w)}$ are read and the w elementary products ${x_k}^{(i)} a_{jk} {y_j}^{(i)}$, $1 \le i \le w$, are created and added to the respective current partial sums $z^{(1)}, \ldots, z^{(w)}$. For this to incur only a constant number of I/Os per non-zero entry, the values ${x_k}^{(1)}, \ldots, {x_k}^{(w)}$ need to be stored in one block (or at least consecutively on disk), similarly to ${y_j}^{(1)}, \ldots, {y_j}^{(w)}$. Since we allow concurrent read, the records can then be accessed by all processors when required. This can be achieved by transposing the matrices $\mathbf{X} = \left[\mathbf{x}^{(1)} \ldots \mathbf{x}^{(w)}\right]$ and $\mathbf{Y} = \left[\mathbf{y}^{(1)} \ldots \mathbf{y}^{(w)}\right]$, which is possible with $\mathcal{O}\left(\frac{w(N_\text{x}+N_\text{y})}{PB} \overline{\log}_\text{d} w\right)$ I/Os for $\text{d} = \max\left\{2, \min\left\{\frac{M}{B}, \frac{w(N_\text{x}+N_\text{y})}{PB}\right\}\right\}$ using the modified PEM merge sort described in Section 2.6. Finally, with a gather step (cf. Section 2.7.1), the P blocks partial sums of each processor can be summed together to form the output records with $\mathcal{O}(\log H/B)$ I/Os.

4.2.2 Sorting Based Algorithms

Reordering the records during the computation of SPMV has been identified in [BBF+10] to be optimal for square matrices in the I/O-model for many (real world) parameter settings. We first extend the algorithms presented there to SPMV for *non-square* matrices with a *single* vector $\mathbf{x}$ in the PEM model. The case of several vectors will be considered later on. For the single vector case, we consider column major layout and best-case layout. For multiple vectors instead, it is asymptotically optimal to transform a column major layout into a best-case layout. This is described in the paragraph on multiple vectors. In general, these algorithms aim to create elementary products within one scan of $\mathbf{A}$ and $\mathbf{x}$, followed by a sorting phase. After the sorting phase, elementary products can be summed immediately in scanning time to create the result vector.

Column Major Layout The sorting based algorithms perform three phases (cf. Figure 4.1): First, elementary products are created and written into the matrix $\mathbf{A}$. In a second step, columns of this layout of elementary products are merged together to form H/N_x meta-columns. Then, the elementary products are summed together to create the output vector.

For a matrix $\mathbf{A}$ given in column major layout, the ordering of $\mathbf{A}$ allows elementary products to be created by scanning $\mathbf{A}$ and $\mathbf{x}$ simultaneously. To this end, the records of $\mathbf{A}$ are distributed evenly among the P processors such that each processor gets at most $\left\lceil \frac{H}{PB} \right\rceil$ blocks assigned to it. Now, each processor scans its assigned records of $\mathbf{A}$ together with the at most $\left\lceil \frac{H}{PB} \right\rceil$ blocks of $\mathbf{x}$ that are required to create elementary products. The elementary products are written back in the same layout as $\mathbf{A}$ – either using the space of **A**, or if **A** is still required, to another area of external memory. Since we allow concurrent reads, multiple processors can access the same records of **x**. Furthermore, by using only $P \leq \frac{H}{2B}$ processors, there are no two processors that write to the same block at the same time. The creation of elementary products hence takes $\mathcal{O}\left(\frac{H}{PB}\right)$ (parallel) I/Os.

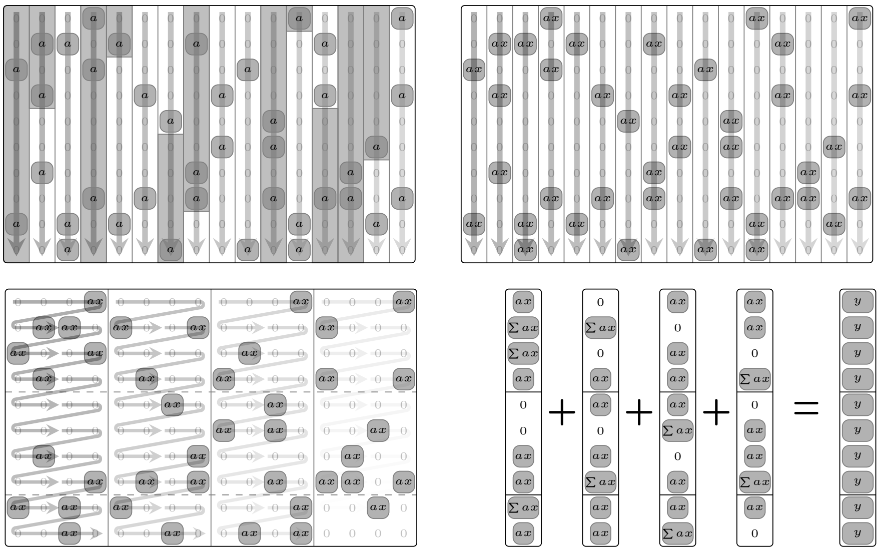


Figure 4.1: Steps of the sorting-based algorithm for SPMV: Initial column major layout of **A** with blocks visualised for $B = 4$ *(upper left)*, layout of elementary products $\mathbf{A}'$ *(upper right)*, meta-columns of elementary products after merging *(lower left)*, summing on average dense meta-columns *(lower right)*.

Let $\mathbf{A}'$ denote the matrix of elementary products. Since $\mathbf{A}'$ is also in column major layout, the columns of $\mathbf{A}'$ constitute N_{x} runs of records that are each sorted by row index. However, if $H/N_{\mathrm{x}} < B$, i.e. the average number of entries per column is smaller than a block, the pre-sorted columns are insignificant, and each block of $\mathbf{A}'$ serves as a run. The $r = \min\{N_{\mathrm{x}}, H/B\}$ runs are bottom up merged, sorted by their row index, by using the modified PEM merge sort algorithm described in Section 2.6. This sorting step, each time reducing the number of runs by a factor of $\mathrm{d}(H) = \max\left\{2, \min\left\{\frac{M}{B}, \frac{H}{PB}\right\}\right\}$, is continued as long as there are more than H/N_{y} runs remaining. The total number of I/Os for this is bounded by $\mathcal{O}\left(\frac{H}{PB}\overline{\log}_{\mathrm{d}(H)} rN_{\mathrm{y}}/H + \log r\right)$. Each of the at most H/N_{y} remaining runs contains on average N_{y} elementary products.

In a last phase, we aim to sum up elementary products to form the output vector $\mathbf{y}$ in parallel. To this end, we think of dividing $\mathbf{A}'$ into tiles where each tile consists of the records in one meta-column that are to be summed into the same block of $\mathbf{y}$. Hence, there are at most $H/N_{\mathrm{y}} \cdot \lceil N_{\mathrm{y}}/B \rceil \approx H/B$ tiles. While these tiles are given on disk ordered by column, processors shall be assigned to them in a row major ordering to allow for an efficient creation of row sums.

First, we create a well-organised layout where each tile consists of exactly one block to ensure a good load-balancing. To this end, elementary products within a meta-column of $\mathbf{A}'$ that belong to the same row are summed together. Since meta-columns are in row major layout, elementary products of the same row are written as consecutive records. By explicitly writing a 0 record for each empty row in a meta-column, afterwards each meta-column will consist of N_{y} records. Hence, the output position of each partial sum is perfectly determined. For this pre-summing, we assign records to processors by the range-bounded load-balancing described in Section 2.7.3 where the tile index is used as key. Hence, with $\mathcal{O}\left(\frac{H}{PB} + \log\min\left\{\frac{H}{P}, B, P\right\}\right)$ I/Os, at most $\lceil 2H/P \rceil$ contiguous records from at most $\lceil 2H/PB \rceil$ tiles are assigned to each processor. If the records of a tile are assigned to the same processor, records can be summed immediately to create one block of partial sums. Summing consecutive records is clearly possible with $\mathcal{O}\left(\frac{H}{PB}\right)$ I/Os. For tiles with records spread among multiple processors, a gather step according to Section 2.7.1 is necessary. By the range-bounded load-balancing, we divided processors into volume processors and range processors. Within the set of range processors, the records of a tile are assigned to a unique processor. For volume processors, the records of a tile can be spread among multiple (contiguous) processors. Hence, it is sufficient to divide this phase into a first part where volume processors operate and perform gather operations, followed by second part where range processors add their partial results. Note that each volume processor is involved in at most two gather operations: It can be the last processor in a collection of processors with records from a certain tile, and be the first processor in a collection of processors for another tile. Additionally, some tiles can be assigned exclusively to this processor. If it is involved in two gather operations, both gather-operations can be performed simultaneously. Since it is the last processor for one gather operation and the last for another, the two operations will not intersect. After performing the gather operations, within another $\mathcal{O}\left(\frac{H}{PB}\right)$ I/Os, the partial sums of range processors can be added. Altogether, the pre-summing is possible with $\mathcal{O}\left(\frac{H}{PB} + \log\frac{H}{B}\right)$ I/Os.

To finally sum partial results, the processors get reassigned in a row-wise ordering of the tiles in the new layout of partial sums. Since the layout of partial sums is independent from the conformation of $\mathbf{A}$, the beginning of each tile is clear to all processors. Assigning tiles to processors in a row-wise ordering, we distinguish the case $P > N_y/B$ and $P \leq N_y/B$. In the latter case, compete rows of tiles are distributed equally among the processors such that at most $\lceil N_y/PB \rceil$ rows of tiles are be assigned to each processor. Then, each processor can sum up tiles to create blocks of $\mathbf{c}$ with one scan inducing $\mathcal{O}\left(\frac{H}{PB}\right)$ I/Os. For $P > N_y/B$, PB/N_y (contiguous) processors are assigned to each row of tiles. A gather task is invoked for each row of tiles, inducing $\mathcal{O}\left(\log \frac{PB}{N_y}\right) = \mathcal{O}\left(\log \frac{H}{N_y}\right)$ I/Os because $P \leq H/B$. Since we have $P > N_y/B$ for this case, and $H \leq N_y N_x$, this is $\mathcal{O}\left(\log \min\left\{H/B, N_x\right\}\right)$. This final summing of tiles constitutes the result vector $\mathbf{c} := \mathbf{A}\mathbf{x}$. The overall number of (parallel) I/Os is thus $\mathcal{O}\left(\frac{H}{PB} \overline{\log}_{d(H)} \min\left\{\frac{N_x N_y}{H}, \frac{N_y}{B}\right\} + \log \frac{H}{B}\right)$.

Best-Case Layout The best-case layout reflects a layout which allows for the best possible worst-case I/O-performance (over all possible matrix conformations). In this layout, we have the non-zero entries of $\mathbf{A}$ partitioned into meta-columns each of which consisting of $m := \min\left\{M - B, H/P\right\}$ consecutive columns. Each meta-column is given in row major layout in external memory. This allows to load and keep the m vector records $\mathbf{x}_{(j-1)m+1}, \ldots, \mathbf{x}_{jm}$ corresponding to the jth meta-column of $\mathbf{A}$ in internal memory of a processor, then scan the jth meta-column (with the remaining block), and write elementary products back to external memory. To this end, we divide the records of $\mathbf{A}$ evenly upon the P processors, each getting assigned no more than $\lceil H/P \rceil$ contiguous records. Since we allow concurrent read, the vector records required to form elementary products can be read simultaneously by multiple processors. Note that the number of vector records required for the computation of elementary products by one processor does not exceed the number of matrix records assigned to it. This step incurs thus $\mathcal{O}\left(\frac{H}{PB}\right)$ I/Os, and creates the matrix $\mathbf{A}'$ of elementary results in the same layout as $\mathbf{A}$. The final process is similar to the algorithm above: Meta-columns are merged together in parallel as long as there are more than H/N_y runs left. These runs are finally summed together in the same manner as described for the previous algorithm. This constitutes the result vector $\mathbf{c}$. Since there are $\lceil N_x/m \rceil$ pre-sorted runs in the beginning, the task is possible with $\mathcal{O}\left(\frac{H}{B} \overline{\log}_{d(H)} \frac{N_x N_y}{mH} + \log \frac{N_x}{m}\right) = \mathcal{O}\left(\frac{H}{PB} \overline{\log}_{d(H)} \frac{N_x N_y}{BH} + \log \frac{N_x}{B}\right)$ (parallel) I/Os.

With slight modifications, this algorithm can be described for a broader class of best-case layouts. In this class, the matrix $\mathbf{A}$ is given as a partition of its columns into meta-columns where each meta-column is written in row

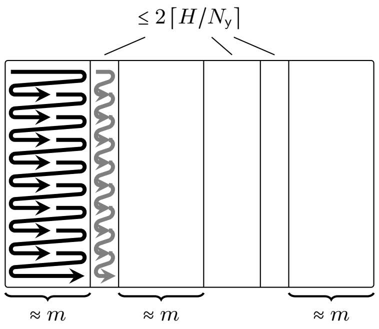


Figure 4.2: The structure of a best-case layout for the sorting based algorithm.

major layout. The columns within a meta-column have to be contiguous, each column is assigned to only one meta-column, and each meta-column consists of an arbitrary number of columns, but at most m. Additionally, the number of meta-columns is limited to at most $2\lceil N_x/B \rceil + 2\lceil H/N_y \rceil$ (cf. Figure 4.2). Since each meta-column consists of no more than $m \leq M - B$ contiguous columns, the corresponding records of $\mathbf{x}$ fit into internal memory while the meta-column is scanned in order to create elementary products. From the so created matrix $\mathbf{A}'$ of elementary products, meta-columns are again merged until there are at most $3\lceil H/N_y \rceil$ runs. Note that there might be no merging process required at all. However, this process can be completed with $\mathcal{O}\left(\frac{H}{PB} \overline{\log}_{d(H)} \frac{N_x/B+H/N_y}{H/N_y}\right) = \mathcal{O}\left(\frac{H}{PB} \overline{\log}_{d(H)} \frac{N_x N_y}{BH}\right)$ I/Os. Again, the resulting runs can be summed together to form the output vector $\mathbf{c}$ with $\mathcal{O}\left(\frac{H}{PB} + \log \frac{H}{B}\right)$ I/Os.

Multiple Vectors For the evaluation of w matrix vector products, we consider the w matrix vectors products as w independent SPMV tasks. Hence, the previous algorithms can be executed for each single vector. To this end, the processors are divided equally among the w SPMV tasks. For $P \geq w$, there are at least $\lfloor P/w \rfloor$ processors assigned to each matrix vector product. This reduces the number of processors by a factor w in the asymptotic complexity of the single vector sorting based algorithms. For $P < w$, each processor performs its $\lceil \frac{w}{P} \rceil$ tasks one after another. Unfortunately, the gather operations induced throughout the summing process require $\frac{w}{P} \log \frac{H}{B}$ I/Os when performed one after another. However, this can be reduced by serialising the gather tasks as follows. Because all products are created with the same matrix, the structure of the summing tasks is similar for all vectors. First, the w/P sorting phases to create meta-columns of elementary products are performed. Then, the output vectors can be obtained by synchronising the w/P summing phases before each gather operation. The at most $2w/P$ gather operations performed by each processor are then serialised (as described in Section 2.7.1) to induce only $\mathcal{O}\left(\frac{wH}{PB} + \log\frac{H}{B}\right)$ I/Os. Altogether, this increases the overall running time for the sorting and scanning phases by a factor w, while the degree is changed to $\mathrm{d}(H, M, B, P/w) = \mathrm{d}(wH, M, B, P) = \max\left\{2, \min\left\{\frac{M}{B}, \frac{wH}{PB}\right\}\right\}$ and only $\mathcal{O}\left(\frac{wH}{PB} + \log H/B\right)$ I/Os are induced by gather operations.

However, if given in column major layout, a change of the layout of $\mathbf{A}$ into a best-case layout can speed-up the process. In our layout transformation, we distinguish two cases, depending on the parameters, similar to the one vector algorithm for column major layout above. Both cases mimic the creation of initial meta-columns of length m while the number of meta-columns shall not fall below H/N_{y}.

The first case handles situations where $N_{\mathsf{x}} \leq H/m$, i.e. a column consists on average of more than $m = \min\{M - B, H/P\}$ (already sorted) records. Then, the N_{x} columns are merged by the PEM merge sort, reducing the number of runs by a factor d in each iteration. The merging process is stopped after at most $\lfloor \log_{\mathrm{d}(H)} m \rfloor$ iterations, or if the number of meta-columns falls below H/N_{y}. In the former case, a meta-column consists finally of at least B, and at most m columns. This parallel merging is upper bounded by $\mathcal{O}\left(\frac{H}{PB} \overline{\log}_{\mathrm{d}(H)} \min\left\{m, \frac{N_{\mathsf{x}} N_{\mathsf{y}}}{H}\right\} + \log N_{\mathsf{x}}\right)$ I/Os, and afterwards there are at most $\max\{\lceil N_{\mathsf{x}}/B\rceil, \lceil H/N_{\mathsf{y}}\rceil\}$ meta-columns.

The second case assumes $H/m \leq N_{\mathsf{x}}$. The possibility of columns having vastly different numbers of entries makes this slightly more involved. We aim to merge groups of at most N_{y} records that span at most m columns in order to create meta-columns.

For $P \leq \frac{N_{\mathsf{x}}}{m} + \frac{H}{N_{\mathsf{y}}}$, each processor performs the following sequential algorithm where the records of $\mathbf{A}$ are distributed evenly among the P processors, such that each processor gets at most $\lceil H/P \rceil$ records assigned to it. The assigned records of $\mathbf{A}$ are scanned, and each maximal group of at most N_{y} records from at most m columns is transformed into row major layout by merging records in $\mathcal{O}\left(\frac{H}{PB} \overline{\log}_{\mathrm{d}(H)} \frac{N_{\mathsf{y}}}{B}\right)$ I/Os. This yields at most $\lceil H/N_{\mathsf{y}} \rceil + \lceil N_{\mathsf{x}}/m \rceil + P \leq 2\lceil H/N_{\mathsf{y}} \rceil + 2\lceil N_{\mathsf{x}}/m \rceil$ meta-columns consisting of up to m columns each.

If $P \geq \frac{N_{\mathsf{x}}}{m} + \frac{H}{N_{\mathsf{y}}}$, we divide the set of processors into two sets of size $\lceil P/2 \rceil$ and $\lfloor P/2 \rfloor$. The records of $\mathbf{A}$ are then distributed among the first set of $\lceil P/2 \rceil$

processors such that each processor gets at most $\lceil 2H/P \rceil$ records assigned. Now, the records are grouped to construct meta-columns of a best-case layout as follows. Every N_{y}th record in $\mathbf{A}$ begins a new group. Additionally, the first record of every mth column begins a new group. This yields $\lceil H/N_{\mathrm{y}} \rceil + \lceil N_{\mathrm{x}}/m \rceil$ groups. These groups shall then be turned into a row-major layout. Groups that are spread among multiple processors have to be merged in parallel by the processors they are assigned to. To this end, the range of processors has to be determined for each group in order to apply the PEM merge sort algorithm. Each processor containing a border of a group, but not the complete group, writes its id into a table, hence identifying that it is the first or last in the range of processors on this group. This takes at most two parallel output operations. By reading concurrently from the table, each processor can determine the range of processors that belong to the same group. Because each group contains at most N_{y} records, a group can be transformed into row major layout with $\mathcal{O}\left(\log_{\mathrm{d}(H)} \frac{N_{\mathrm{y}}}{B}\right)$ PEM merge sort iterations. However, a processor can belong to up to two such groups because it can be the first processor in one group and the last processor in another. To tackle this problem, the role of a processor containing the first records of a group (other than the first group) is replaced by a processor from the second set of processors (consisting of $\lfloor P/2 \rfloor$ processors). To this end, the ith processor in the second set of processors is reserved to take over the role of processor $i+1$ in the first set. Hence, each group that is spread over multiple processors, is assigned to a unique set of processors. After $\mathcal{O}\left(\frac{H}{PB} \overline{\log}_{\mathrm{d}(H)} \frac{N_{\mathrm{y}}}{B}\right)$ I/Os, these groups are transformed into row major layout. Transforming groups that are assigned to a single processor only can be achieved according to the case $P \leq \frac{N_{\mathrm{x}}}{m} + \frac{H}{N_{\mathrm{y}}}$ with another $\mathcal{O}\left(\frac{H}{PB} \overline{\log}_{\mathrm{d}(H)} \frac{N_{\mathrm{y}}}{B}\right)$ I/Os. The whole transformation process is hence possible with $\mathcal{O}\left(\frac{H}{PB} \overline{\log}_{\mathrm{d}(H)} \frac{N_{\mathrm{y}}}{B}\right)$ (parallel) I/Os, and yields $\lceil H/N_{\mathrm{y}} \rceil + \lceil N_{\mathrm{x}}/m \rceil$ meta-columns.

4.2.3 Table Based Algorithms

In the worst-case, the direct algorithm performs one input of vector records for every single non-zero entry of $\mathbf{A}$. Thus, an access to a block of vector records leads to the creation of a single elementary product for each vector only. Though for rather sparse cases sometimes optimal, this can be improved by initially creating tuples of vector records. We exploit this in the following algorithms which are formulated for bilinear forms.

In these algorithms, t tables of c-tuples are created in an initial phase. A c-tuple consists of c rows of $\mathbf{Y} = \left[\mathbf{y}^{(1)} \ldots \mathbf{y}^{(w)}\right]$, hence, containing cw records.

Each of the t tables contains all the c-tuples from a range of $\lceil N_y/t \rceil$ rows, in lexicographical order of row indices. Thus, a tuple consists of the cw records $([y_{i_1}{}^{(1)}, \ldots, y_{i_1}{}^{(w)}], \ldots, [y_{i_c}{}^{(1)}, \ldots, y_{i_c}{}^{(w)}])$ for $i_1, \ldots, i_c$ within one of the t ranges in $[1, \ldots, N_y]$ (cf. Figure 4.3). Note that the indices $i_1, \ldots, i_c$ are an arbitrary (ordered) subset of the t row indices of a table, i.e. they do not have to refer to contiguous positions in a vector. The total number of tuples in all tables is thus given by $t\binom{N_y/t}{c}$. Since we consider ordered tuples, it obviously has to hold $c \leq N_y/t$. We restrict ourselves here to $N_y/t \geq 3c$.

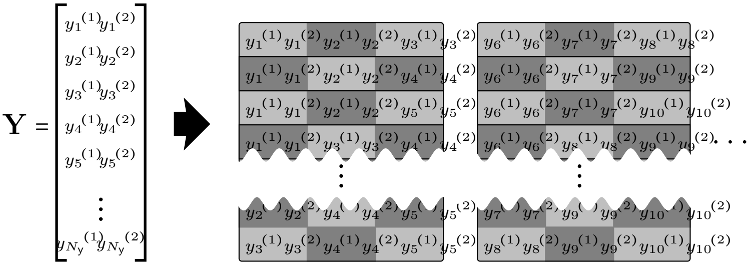
Note that it is sufficient to assume $c \leq B/w$ for an optimal algorithm. For any $c' > c = B/w$, the size of the tables is strictly larger than for c. In contrast, any c'-tuple – which requires $\lceil c'w/B \rceil$ I/Os to be loaded – can be loaded by accessing $\lceil c'/c \rceil = \lceil c'w/B \rceil$ c-tuples.

When loading a tuple into internal memory, on average at least a constant fraction of the c contained records can be used to create elementary products for sufficiently large t. Given the tables of tuples, this leads to algorithms with I/O complexity $\mathcal{O}\left(\frac{H}{cP}\right)$ which improve on the direct algorithm for non-constant c. In order to achieve a low overall I/O complexity, the creation of tuple tables is also approximately I/O-bounded by this term. It will turn out that the creation of the tables is dominated by the output operations to write them which requires $\left\lceil t\binom{N_y/t}{c} \cdot cw/PB \right\rceil$ I/Os. Hence, we require

$$t\binom{N_y/t}{c} \cdot \frac{cw}{PB} \leq \frac{H}{cP}$$

which is implied by

$$c \leq \frac{\log \frac{HB}{c^2wt}}{\log \frac{eN_y}{ct}} \tag{4.2}$$



$\mathbf{Y} =$
$y_1^{(1)} y_1^{(2)}$
$y_2^{(1)} y_2^{(2)}$
$y_3^{(1)} y_3^{(2)}$
$y_4^{(1)} y_4^{(2)}$
$y_5^{(1)} y_5^{(2)}$
$y_{N_y}^{(1)} y_{N_y}^{(2)}$
$y_1^{(1)} y_1^{(2)} y_2^{(1)} y_2^{(2)} y_3^{(1)} y_3^{(2)}$
$y_1^{(1)} y_1^{(2)} y_2^{(1)} y_2^{(2)} y_4^{(1)} y_4^{(2)}$
$y_1^{(1)} y_1^{(2)} y_2^{(1)} y_2^{(2)} y_5^{(1)} y_5^{(2)}$
$y_1^{(1)} y_1^{(2)} y_3^{(1)} y_3^{(2)} y_4^{(1)} y_4^{(2)}$
$y_2^{(1)} y_2^{(2)} y_4^{(1)} y_4^{(2)} y_5^{(1)} y_5^{(2)}$
$y_3^{(1)} y_3^{(2)} y_4^{(1)} y_4^{(2)} y_5^{(1)} y_5^{(2)}$
$y_6^{(1)} y_6^{(2)} y_7^{(1)} y_7^{(2)} y_8^{(1)} y_8^{(2)}$
$y_6^{(1)} y_6^{(2)} y_7^{(1)} y_7^{(2)} y_9^{(1)} y_9^{(2)}$
$y_6^{(1)} y_6^{(2)} y_7^{(1)} y_7^{(2)} y_{10}^{(1)} y_{10}^{(2)}$
$y_6^{(1)} y_6^{(2)} y_8^{(1)} y_8^{(2)} y_9^{(1)} y_9^{(2)}$
$y_7^{(1)} y_7^{(2)} y_9^{(1)} y_9^{(2)} y_{10}^{(1)} y_{10}^{(2)}$
$y_8^{(1)} y_8^{(2)} y_9^{(1)} y_9^{(2)} y_{10}^{(1)} y_{10}^{(2)}$

Figure 4.3: Tables of c-tuples for $w = 2$ vectors with $c = 3$ and $t = 5$. A block spans B/cw rows in a table.

using $\binom{n}{k} \leq \left(\frac{en}{k}\right)^k$.

The creation of tuple tables can be achieved as follows. By transposing $\mathbf{Y}$ – which we assume to be given column/vector wise – into row major layout, the trivial 1-tuples are created. This takes $\mathcal{O}\left(\frac{wN_\mathrm{y}}{PB} \overline{\log}_{\mathrm{d}(wN_\mathrm{y})} w\right)$ I/Os. Afterwards, the size of tuples is increased iteratively. Given a table T of $(c-1)$-tuples, a table of c-tuples can be generated by extending each tuple of T by one row of $\mathbf{Y}$. Since we aim to create a table of unique ordered sets, it is sufficient to extend a $(c-1)$-tuple by rows that have higher row index than the greatest row index present in the $(c-1)$-tuple. We divide the P processors equally among the $\binom{N_\mathrm{y}/t}{c}$ tuples that have to be created such that each processor is assigned to at most $\left\lceil \binom{N_\mathrm{y}/t}{c}/P \right\rceil$ contiguous tuples of the task. Each processor creates its tuples by scanning the required tuples of T, and for each $(c-1)$-tuple by scanning the corresponding rows of $\mathbf{Y}$ required to extend the tuple. Since we have $M \geq 4B$, internal memory can hold one block of T, one of $\mathbf{Y}$ and a block to buffer output tuples. The creation of a table is then dominated by writing the tuples in each iteration. Increasing the size of tuples by 1 increases the size of the table by a factor at least

$$\frac{c\binom{N_\mathrm{y}/t}{c}}{(c-1)\binom{N_\mathrm{y}/t}{c-1}} = \frac{c}{c-1}\frac{N_\mathrm{y}/t-(c-1)}{c} \geq \frac{N_\mathrm{y}}{ct} - 1\,.$$

Note that each iteration requires at least one parallel I/O. The total number of I/Os to write all tables in all iterations is thus bounded by

$$\sum_{i=1}^{c}\left\lceil t\binom{N_\mathrm{y}/t}{i}\frac{iw}{PB}\right\rceil \leq \left\lceil t\binom{N_\mathrm{y}/t}{c}\frac{cw}{PB}\sum_{j=0}^{\infty}\left(\frac{1}{\frac{N_\mathrm{y}}{ct}-1}\right)^{j}\right\rceil + c \leq 2t\binom{N_\mathrm{y}/t}{c}\frac{cw}{PB} + c$$

where the last inequality results from $N_\mathrm{y}/t \geq 3c$.

Note that in the current construction, a c-tuple can cross block borders in external memory so that two I/Os are required to read this tuple. To tackle this problem, during the construction of the final table of c-tuples, this case is avoided by writing only $\left\lfloor \frac{B}{cw} \right\rfloor$ tuples in each block. This increases the number of I/Os by a factor 2 at most.

It remains to choose t in order to minimise the I/O complexity. For large t, the total number of tuples in all tables becomes small. This leads to a larger c according to (4.2). However, if chosen too large, there might be less than c contiguous non-zero entries in a range of N_y/t rows in the layout of $\mathbf{A}$. In this case, loading a tuple leads to less than c elementary products. Hence, we choose t in the following such that in the layout of $\mathbf{A}$, c contiguous non-zero entries span on average N_y/t rows. The total number of I/Os to create the tables and the results $z^{(1)}, \ldots, z^{(w)}$ is then $\mathcal{O}\left(\frac{H}{cP} + c\right)$.

Column Major Layout The bilinear forms $\mathbf{y}^{(i)^T}\mathbf{A}\mathbf{x}^{(i)}$ for w vector pairs with a matrix $\mathbf{A}$ in column-major layout can be evaluated with

$$\mathcal{O}\left(\max\left\{\frac{wH}{PB}, \frac{H\log\min\left\{\frac{N_\mathsf{x}N_\mathsf{y}}{H}, \frac{N_\mathsf{y}}{B}\right\}}{P\log\min\left\{N_\mathsf{x}, \frac{H}{B}\right\}}\right\} + \log\frac{H}{B}\right)$$

(parallel) I/Os. To this end, let

$$t := \max\left\{\frac{H}{3N_\mathsf{x}c}, \frac{B}{c}\right\} \tag{4.3}$$

which leads by using (4.2) and estimating $cw \leq B$ to

$$c := \min\left\{\left\lfloor\frac{B}{w}\right\rfloor, \left\lfloor\frac{\log\min\left\{3N_\mathsf{x}, \frac{H}{B}\right\}}{\log\min\left\{\frac{3eN_\mathsf{x}N_\mathsf{y}}{H}, \frac{eN_\mathsf{y}}{B}\right\}}\right\rfloor\right\}. \tag{4.4}$$

The intuition behind the choice of t stems from the following considerations. Recall that t is chosen so that c contiguous non-zero entries span on average N_y/t rows. Like in the sorting based algorithm for column major layout in Section 4.2.2, we distinguish between two bounds depending on the cases $H/N_\mathsf{x} \leq B$ and vice versa. In a column there are on average H/cN_x groups of c contiguous non-zero entries, leading to the first term of the maximum in (4.3). Similarly, in each block there are B/c groups of c non-zero entries which yields the second term of the maximum. Note that we assumed $N_\mathsf{y} \geq M \geq 4B$ so that $B/c \leq N_\mathsf{y}/4c$. Since moreover $H \leq N_\mathsf{x}N_\mathsf{y}$, the number of tables t is bounded from above by $N_\mathsf{y}/3c$ as we require.

In the algorithm, processors are assigned by the range-bounded load-balancing such that each processor gets assigned at most $\left\lceil\frac{2H}{P}\right\rceil$ records from at most $\left\lceil\frac{2N_\mathsf{x}}{P}\right\rceil$ columns. Each processor scans its assigned records of $\mathbf{A}$ simultaneously with the corresponding records of $\mathbf{X}$. To this end, $\mathbf{X}$ is transposed first which takes $\mathcal{O}\left(\frac{wN_\mathsf{x}}{PB}\,\overline{\log}_{\mathrm{d}(wN_\mathsf{x})} w\right)$ I/Os. For each c non-zero entries, the corresponding c-tuple is loaded if it is existent in a table. Otherwise, the c non-zero entries correspond to rows from multiple tables, and from each of these tables one tuple containing the required records is loaded. The created elementary products are then summed into a reserved block in internal memory, keeping the partial results of $z^{(1)}, \ldots, z^{(w)}$. Finally, the partial results of all processors are summed together with a gather step to form the final output with $\mathcal{O}(\log\min\{P, H/B\})$ I/Os.

On average each c non-zero entries induce at most 2 accesses to a table: The first case induces at most $\frac{H}{cP}$ inputs of c-tuples. The second case can occur at most two times per table in each column (once for a c-tuple ending in table T, and a second time for another tuple starting T), thus incurring no more than $2\left\lceil\frac{2N_x}{P}\right\rceil t$ I/Os per processor. It is however also restricted to occur twice at the most per table for each block of $\mathbf{A}$ which yields a stronger bound for $B \geq H/N_x$. Hence, the total number of table accesses is upper bounded by $\frac{H}{cP} + 2t \cdot \min\left\{\left\lceil\frac{2N_x}{P}\right\rceil, \left\lceil\frac{2H}{PB}\right\rceil\right\} \leq 5\left\lceil\frac{H}{cP}\right\rceil$.

Best-Case Layout The bilinear forms $\mathbf{y}^{(i)^T}\mathbf{A}\mathbf{x}^{(i)}$ for w vector pairs with a matrix $\mathbf{A}$ in best-case layout can be evaluated with

$$\mathcal{O}\left(\max\left\{\frac{wH}{PB}, \frac{H\log\left(\frac{w^2 N_x N_y}{BH\min\{M,H/P\}}\log\frac{N_x}{\min\{M,H/P\}}\right)}{P\log\frac{N_x}{\min\{M,H/P\}}} + \log\frac{H}{B}\right\}\right)$$

(parallel) I/Os. For this, let

$$t := \min\left\{\frac{BH\min\{M,H/P\}}{c^2w^2N_x}, \frac{N_y}{3c}\right\} \tag{4.5}$$

so that it is sufficient for (4.2) to choose

$$c := \min\left\{\left\lfloor\frac{B}{w}\right\rfloor, \left\lfloor\frac{\log\frac{Nx}{\min\{M,H/P\}}}{\log\left(\frac{ew^2N_xN_y}{BH\min\{M,H/P\}}\log\frac{Nx}{\min\{M,H/P\}}\right)}\right\rfloor\right\}. \tag{4.6}$$

To exploit these settings of c and t, the following layout is used: The matrix $\mathbf{A}$ is organised in meta-columns consisting of $s = \left\lceil\frac{Bm}{cw^2}\right\rceil$ columns for $m := \min\left\{M - B, \left\lceil\frac{H}{P}\right\rceil\right\}$ and each meta-column is in row major layout. Thus, in each meta-column there are sH/N_x non-zero entries, and hence sH/cN_x groups of c contiguous non-zero entries, on average (cf. Figure 4.4).

The processors are assigned to the records of $\mathbf{A}$ using the range-bounded load-balancing using the meta-column index as key such that each processor gets assigned at most $\lceil 2H/P\rceil$ records from at most $\left\lceil\frac{2\lceil N_x/s\rceil}{P}\right\rceil$ different meta-columns. Recall that the range-bounded load-balancing algorithm for $N_x/s = \mathcal{O}(N_x/B)$ keys induces $\mathcal{O}\left(\frac{H}{PB} + \log N_x/B\right)$ I/Os. During the algorithm, each processor reads its assigned records of $\mathbf{A}$ in pieces of m/w contiguous non-zero entries. For each piece, the corresponding vector records of $\mathbf{X}$ and tuples of $\mathbf{Y}$ are loaded to create elementary products. Hence, $\mathbf{X}$ is initially transposed again, taking $\mathcal{O}\left(\frac{wN_x}{PB}\,\overline{\log}_{d(wN_x)} w\right)$ I/Os. Loading the $\mathbf{X}$

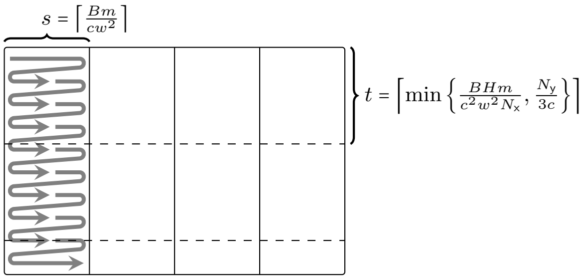

$s = \left\lceil \frac{Bm}{cw^2} \right\rceil$
$t = \left\lceil \min\left\{ \frac{BHm}{c^2w^2N_x}, \frac{N_y}{3c} \right\} \right\rceil$

Figure 4.4: Meta-columns of the best-case layout for the table based algorithm. In each tile are on average c non-zero entries.

records corresponding to a meta-column requires $\lceil sw/B \rceil = \lceil m/cw \rceil$ I/Os. If a piece covers more than one meta-column, reading the corresponding records of $\mathbf{X}$ is more expensive. However, this can happen at most $\left\lceil \frac{2\lceil N_x/s \rceil}{P} \right\rceil - 1$ times per processor, inducing $\mathcal{O}\left(\frac{wN_x}{mP}\right)$ I/Os in total. The overall number of I/Os to records of $\mathbf{X}$ is thus restricted to $\left\lceil \frac{\lceil 2H/P \rceil}{m/w} \right\rceil \cdot \lceil m/cw \rceil + \mathcal{O}\left(\frac{wN_x}{mP}\right) = \mathcal{O}\left(\frac{H}{cP} + \frac{wN_x}{PB}\right)$.

Bounding the accesses to $\mathbf{Y}$ tuples follows closely the analysis of the algorithm for column-major layout. Each c non-zero entries require the input of at least one c-tuple. This induces at most $3\frac{H}{cP}$ parallel inputs of c-tuples and another at most $t\left\lceil \frac{2\lceil N_x/s \rceil}{P} \right\rceil = \mathcal{O}\left(\frac{H}{cP}\right)$ I/Os for groups of c non-zero entries that cover several ranges of tables.

4.3 Lower Bounds

A lower bound for sparse $N \times N$ matrices in the I/O-model was presented in [BBF+10]. For the proof of a lower bound in column major layout, we follow closely their description but have to restate the proof for the PEM model. Moreover, the dimensions are substituted by N_x and N_y. To obtain a lower bound for the best-case layout with multiple vectors, the approach of [BBF+10] has to be extended further. In all the presented lower bounds, we only consider SPMV, implying lower bounds for BIL by Chapter 3.

4.3.1 Column Major Layout

In this section, we consider the product of a sparse matrix with a single vector only. Algorithmically, it can make sense to change the layout of the matrix **A** for multiple evaluations of SPMV into a best-case layout. Lower bounds for SPMV with multiple vectors and a matrix in best-case layout are presented in the next section, and imply lower bounds for matrices in column major layout. Thus, Theorem 4.2 is a combination of the results in this section, and the lower bound in Section 4.3.2. To obtain a lower bound for non-square matrices in the I/O-model, it is sufficient to replace the dimensions of **A** in the proof in [BBF$^+$10]. Since we extend the bounds to the PEM model using the considerations given in Section 2.2, we also restate the required arguments.

Following [BBF$^+$10], to obtain a lower bound for SPMV with a matrix in column major layout, it suffices to consider the multiplication of the matrix with the all-ones-vector. This corresponds to the task of creating row sums of the matrix. In [BBF$^+$10], a copy task is considered where a sparse matrix is created from copies of the vector records of a given vector. The analysis of this task yields a preliminary result for the analysis of SPMV. In Section 7.3.1, a broad range of tasks is analysed involving an extension of the copy task in the PEM model. While Bender et al. [BBF$^+$10] gave a transformation from the copy task to SPMV by reversing the direction of time in a program, we consider SPMV directly. To this end, the change of configurations is considered backwards in time. Observe that there are multiple input forms depending on the conformation that can all create the same output.

Abstract Configurations We consider programs that are normalised according to Chapter 2. Hence, only direct predecessors of the output records exist, and after each operation, records that are not required further are removed immediately. Moreover, to apply Lemma 2.6, we consider abstract configurations as described in Section 2.2.2: The ordering and multiplicity of records in a block and in internal memory is ignored, as well as empty blocks in external memory.

We make one further abstraction here, in that we reduce records to their row indices. A similar abstraction is also made in [BBF$^+$10]. This is possible because we consider only records of **A** and sums of them in this task. Since in a normalised program, only records from the same row of **A** are summed, each intermediate result can be related to a single row index. Together with the previous abstraction, blocks and internal memories are considered subsets of $[N_y]$, each set having limited size. Hence, the ordering and the number of distinct records with the same row index is ignored.

Description of Abstract Programs In this view, copy operations and sum operations do not change the abstract configuration. Similarly, deletions of operands after a sum operation become invisible in the sequence of abstract configuration. Furthermore, in the task of creating row sums, no multiplication operations are required (and are hence not performed in a normalised program). Thus, there is only a single abstract computation trace in the set of normalised programs for this task, and it remains to consider the change of abstract configurations made by I/O operations.

By Lemma 2.6, the number of initial abstract configurations, that can lead to a single abstract (output) configuration after ℓ I/Os is bounded by

$$\prod_{l=1}^{\ell} 3^P \prod_{p=1}^{P} \binom{M_{p,l}+B}{B} 2^B 2(\lceil n/B \rceil + P\ell)$$

where $M_{p,l}$ is the number of records in internal memory of processor p after the lth I/O is performed, and n is the total input size. Recall that in an abstract configuration according to Section 2.2.2, records that do not belong to the final output are ignored. For fixed parameters N_{x}, N_{y}, and H, the abstract output configuration is hence unique. In contrast, the initial configuration depends on the conformation of the matrix, i.e. the positions of the non-zero entries. The changes of abstract configurations can be consider as a tree rooted in the final configuration. Each leaf corresponds to a different matrix conformation and each layer of depth i corresponds to the configurations at time $\ell - i$.

Note that during the execution of a normalised program for SpMV with a single vector, in any configuration there can be no more than H non-empty records, and thus at most H non-empty blocks on disk. Observe furthermore that the product of several binomial coefficients can be bounded above by a single binomial coefficient: The number of possibilities to choose multiple times from small sets is exceeded by the number of choices when drawing all at once from the union of all sets. The product of binomial coefficients can hence be bounded by

$$\prod_{p=1}^{P} \binom{M_{p,l}+B}{B} \leq \binom{\sum_{p=1}^{P} M_{p,l} + PB}{PB} \leq \binom{\min\{MP, H\} + PB}{PB}$$

and together with bounding the number of blocks $2(\lceil n/B \rceil + P\ell)$ that can be read, we obtain

$$\left(3^P \binom{\min\{MP, H\} + PB}{PB} 2^{PB} H^P \right)^{\ell}$$

as an upper bound on the number of abstract configurations that can lead to the same configuration after ℓ I/Os.

Number of Abstract Matrix Conformations It will be sufficient to consider only matrices that have the same number of non-zero entries in each column. To this end, we assume for the lower bound that H is an integer multiple of N_y. Then, there are in total $\binom{N_\mathsf{y}}{H/N_\mathsf{x}}^{N_\mathsf{x}}$ different conformations of $N_\mathsf{y} \times N_\mathsf{x}$ matrices with H/N_x non-zero entries per column. However, since we consider abstract configurations and ignore the ordering and multiplicity of records within a block, the number of initial abstract configurations is smaller. For an abstract configuration, it is not clear whether a row index in a block stems from a single or from multiple columns, neither from which of them. Following [BBF⁺10], we distinguish three cases depending on how the number of records per column H/N_x and the block size B relate to each other. If $H/N_\mathsf{x} = B$, each block corresponds to exactly one column so that each abstract configuration describes only a single conformation. In case $H/N_\mathsf{x} > B$, a block contains entries from at most two rows. Hence, a column index in the abstract description of a block can originate either from the first, second or both rows. For $H/N_\mathsf{x} < B$, a block contains entries from $\lceil BN_\mathsf{x}/H \rceil$ different columns. A column can however only contain indices from at most two blocks. Thus, there are $\binom{2B}{H/N_\mathsf{x}}^{N_\mathsf{x}}$ orderings of records in external memory with the same abstract description.

Calculations The above considerations yield the following inequality

$$\left(3^P \binom{\min\{MP, H\} + PB}{PB} 2^{PB} (2H)^P\right)^\ell \geq \binom{N_\mathsf{y}}{H/N_\mathsf{x}}^{N_\mathsf{x}} / \tau$$

with

$$\tau = \begin{cases} 3^H & \text{for } H/N_\mathsf{x} > B, \\ 1 & \text{for } H/N_\mathsf{x} = B, \\ \binom{2B}{H/N_\mathsf{x}}^{N_\mathsf{x}} & \text{for } H/N_\mathsf{x} < B. \end{cases}$$

For $H/N_\mathsf{x} < B$, by taking logarithms and estimating binomial coefficients according to Observation 2.1, we obtain

$$\ell P \left(\log 6H + B \log \frac{2e(m + PB)}{PB} \right) \geq H \log \frac{N_\mathsf{y}}{H/N_\mathsf{x}} - H \log \frac{2eBN_\mathsf{x}}{H}$$

with $m := \min\{MP, H\}$ (and the Euler number e). Because we assume $M \geq 2B$ and $H \geq PB$, we get

$$\ell \geq \frac{H}{P} \frac{\log \frac{N_\mathsf{y}}{2eB}}{\log 6H + B \log \frac{4em}{B}} . \tag{4.7}$$

For $H/N_{\mathrm{x}} \geq B$, we have

$$\ell P\left(\log 6H + B\log\frac{2e(m+PB)}{PB}\right) \geq H\log\frac{N_{\mathrm{x}}N_{\mathrm{y}}}{H} - H\log 3$$

which yields

$$\ell \geq \frac{H}{P}\frac{\log\frac{N_{\mathrm{x}}N_{\mathrm{y}}}{3H}}{\log 6H + B\log\frac{4em}{B}}\,. \tag{4.8}$$

Combining (4.7) and (4.8), we obtain

$$\ell \geq \frac{H}{P}\frac{\log\min\left\{\frac{N_{\mathrm{y}}N_{\mathrm{x}}}{3H}, \frac{N_{\mathrm{y}}}{2eB}\right\}}{\log 6H + B\log\frac{4em}{B}}\,, \tag{4.9}$$

and it remains to distinguish the leading terms in the denominator.

For $\log 6H \leq B\log\frac{4em}{B}$, (4.9) asymptotically matches the sorting based algorithm. Observe that for $\mathrm{d} = \max\left\{2, \min\left\{\frac{M}{B}, \frac{H}{PB}\right\}\right\}$, it holds $\log\frac{m}{B} + \log 4e < \log\frac{m}{B} + 3.5 \leq \frac{9}{2}\log\mathrm{d}$. Thus, we get

$$\ell \geq \frac{H}{9PB}\log_{\mathrm{d}}\min\left\{\frac{N_{\mathrm{x}}N_{\mathrm{y}}}{3H}, \frac{N_{\mathrm{y}}}{2eB}\right\}\,,$$

and it holds

$$\ell \geq \frac{H}{12PB}\log_{\mathrm{d}}\min\left\{\frac{N_{\mathrm{x}}N_{\mathrm{y}}}{H}, \frac{N_{\mathrm{y}}}{B}\right\} \tag{4.10}$$

for $\log_2\min\left\{\frac{N_{\mathrm{x}}N_{\mathrm{y}}}{H}, \frac{N_{\mathrm{y}}}{B}\right\} \geq 10$ using $x - \log_2 2e \geq \frac{3}{4}x$ for $x \geq 10$. Otherwise, if $\log_2\min\left\{\frac{N_{\mathrm{x}}N_{\mathrm{y}}}{H}, \frac{N_{\mathrm{y}}}{B}\right\} < 10$, a scanning bound of $\frac{H}{PB}$ for reading $\mathbf{A}$ implies (4.10).

Now, consider the case $\log 6H > B\log\frac{4em}{B}$. Using $H \geq 6$, we have

$$\ell \geq \frac{H}{4P}\frac{\log\min\left\{\frac{N_{\mathrm{x}}N_{\mathrm{y}}}{3H}, \frac{N_{\mathrm{y}}}{2eB}\right\}}{\log H}\,,$$

implying

$$\ell \geq \frac{H}{7P}\frac{\log\min\left\{\frac{N_{\mathrm{x}}N_{\mathrm{y}}}{H}, \frac{N_{\mathrm{y}}}{B}\right\}}{\log H}\,. \tag{4.11}$$

for the assumption $\log_2\min\left\{\frac{N_{\mathrm{x}}N_{\mathrm{y}}}{H}, \frac{N_{\mathrm{y}}}{B}\right\} \geq 6$ so that we can use $x - \log_2 2e \geq \frac{4}{7}x$ for $x \geq 6$. Otherwise, for $\log_2\min\left\{\frac{N_{\mathrm{x}}N_{\mathrm{y}}}{H}, \frac{N_{\mathrm{y}}}{B}\right\} < 6$, a scanning bound of $\frac{H}{PB}$ holds. Additionally, we have $\log 6H > B\log\frac{4em}{B} \geq 2B$ so that $\frac{H}{PB} \geq \frac{H}{P\log H}$ holds which justifies (4.11) for this case as well.

4.3.2 Best-Case Layout

As described in Section 1.2.1, for the best-case layout, we consider a layout for $\mathbf{A}$ that allows for the best possible I/O complexity for SPMV. The proof presented in the previous section is based on the task of computing row sums. To obtain a lower bound for the best-case layout, we have to use a different approach because producing row sums is trivial when using a row major layout. Instead, the movement of both, input-records from $\mathbf{X}$ and partial results of the output $\mathbf{C} = \left[\mathbf{c}^{(1)} \dots \mathbf{c}^{(w)}\right]$, is considered. Additionally, we consider the computation trace of a program. The accesses to $\mathbf{A}$ are ignored which can only weaken the lower bound. We think of having the set of programs for SPMV with at most ℓ I/Os given, and extract the trace information to count the number of different matrix conformation that can be handled by programs in the set.

This approach was used in [BBF+10] for a lower bound for SPMV with a square sparse matrix in best-case layout with one vector. While it is straightforward to adapt their proof to non-square situations, the consideration of SPMV with multiple vectors requires a different technique. Their original proof uses the counting arguments of Section 2.2 to identify the traces of $\mathbf{X}$ and $\mathbf{C}$ records. To this end, the movement of partial results of $\mathbf{C}$ is traced backwards from the final configuration according to Section 2.2.2, similar to Section 4.3.1. Additionally, the movement of $\mathbf{X}$ records is traced forwards from the initial configuration according to Section 2.2.1. Furthermore, the computation trace is identified by considering the number of possible multiplication operations in a program with fixed traces of $\mathbf{X}$ and $\mathbf{C}$ records.

For multiple evaluations of SPMV, we extract a program for the evaluation of only one of the w matrix vector products. This extracted program computes SPMV for the input vector $\mathbf{x}^{(i)}$ which induces the smallest number of records in internal memories and in blocks that are input/output during the execution of the program. For each conformation of the matrix $\mathbf{A}$, there must be at least one extracted program computing $\mathbf{c}^{(i)} = \mathbf{A}\mathbf{x}^{(i)}$. In order to identify the number of extracted programs for different matrix conformations, we describe how to determine the matrix conformation uniquely for a given extracted program. Since we abstract from the ordering and multiplicity of records in Section 2.2, the I/O-trace of records alone does not identify the conformation uniquely. However, given the computation trace, the matrix conformation can be determined. A multiplication operation with operand $x_k^{(i)}$ which creates an elementary product that is a predecessor of $c_j^{(i)}$, describes the existence of a non-zero entry a_{jk} in $\mathbf{A}$ (cf. Figure 4.5).

A version of the lower bound for the I/O-model was published in [GJ10a] using a statement that combines the counting argument with Hong Kung

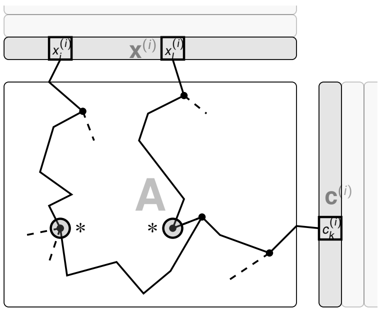


Figure 4.5: Tracing the records for only one vector pair $\mathbf{x}^{(i)}$, $\mathbf{c}^{(i)}$. The movement of records, and the multiplications in a program identifies the conformation of the matrix.

rounds (cf. Section 2.1). Recall that by Lemma 2.3, any program with ℓ (parallel) I/Os can be converted into a program for internal memories of size $2M$, operating in $\lceil \frac{\ell B}{M} \rceil$ rounds. Each round consists of up to $2M/B$ (parallel) input operations, followed by a sequence of computation operations, and ends with at most $2M/B$ output operations. All previous results are extended to the PEM model in this thesis. A classical round-based program, however, excludes communication between processors during a round. Hence, we state the lower bound here for normal programs, using insights that are gained from the analysis of round-based programs. We remark that an argument similar to Hong Kung rounds exists for PEM programs as well (see Section 2.1). However, applying it does not simplify the proof nor does it yield further insights.

The extracted programs analysed below are obtained as follows. Consider a program $\mathcal{P}$ for w instances of SPMV with a matrix $\mathbf{A}$. Every block transferred by an I/O in $\mathcal{P}$ can be separated into values belonging to the w different tasks of SPMV implied by the w pairs of vectors. Hence, for the lth I/O performed by processor p, we have the number $b_{p,l}^{(i)}$ of records belonging to vector pair $\mathbf{x}^{(i)}$, $\mathbf{c}^{(i)}$. Similarly, we have the number of records belonging to vector pair i in internal memory of processor p at the time the lth I/O is performed $m_{p,l}^{(i)}$. Clearly, we have $\sum_{p,i,l} b_{p,l}^{(i)} \leq \ell PB$, and $\sum_{p,i,l} m_{p,l}^{(i)} \leq \ell PM$. Let $f^{(i)} = \sum_p \sum_l (b_{p,l}^{(i)} + \frac{B}{M} m_{p,l}^{(i)})$ so that $\sum_i f^{(i)} \leq 2\ell PB$. By averaging, there is some i such that $f^{(i)} \leq 2\ell PB/w$.

We extract a program $\mathcal{P}^{(i)}$ for $\mathbf{A}\mathbf{x}^{(i)}$ from $\mathcal{P}$. To this end, all records from $\mathbf{x}^{(j)}$, and partial sums of $\mathbf{c}^{(j)}$ for $j \neq i$ are ignored. Similarly, all records that are not a predecessor of some $c_k^{(i)}$ are ignored, and computation operations that do not create a predecessor of some $c_k^{(i)}$ are removed. Note that this does however not change the number of I/O operations. Ignoring the computation for other $\mathbf{c}^{(j)}$, $j \neq i$ allows for a normalisation of the extracted program. Since we ignore records that are no predecessors of the output $\mathbf{c}^{(i)}$, the multiplication operations in $\mathcal{P}^{(i)}$ can be moved. In a normalised program, a multiplication operation is performed immediately when the required records are in internal memory of a processor. Obviously, this must be immediately after an input. This modification is possible because in an extracted program, a non-zero entry $a_{j,k}$ is used for exactly one elementary product, and hence, involved in exactly one multiplication operation. The result of the multiplication can hence be written into the record containing $a_{j,k}$ after the preceding input is performed. Modifying a program according to this normalisation does not change the number of I/Os, nor does it change the conformation(s) that can be handled by the program.

Abstract Configurations We distinguish between two abstractions for each configuration, similar as in [BBF+10]. In the abstract row configuration, only records that contain a partial result of $\mathbf{c}^{(i)}$ are considered. We abstract similar to Section 4.3.1 such that only sets of row indices are considered in the abstract row configuration. In the abstract column configuration, only records from $\mathbf{x}^{(i)}$ are considered, and we abstract from the content of a record $x_j^{(i)}$ to its (column) index j. Hence, in the abstract column configuration, blocks and internal memories are considered subsets of $[N_\mathrm{x}]$. The abstraction to row and column indices still allows for the multiplication trace to identify the conformation of the matrix uniquely.

The I/O Trace To determine the number of distinct I/O-traces described by the two abstract configurations, we apply Lemma 2.5 and 2.6. Recall that the maximum number of distinct (abstract) conformations that can be reached by a family of programs with ℓ I/Os for fixed input, or that can reach a fixed output, is

$$\prod_{l=1}^{\ell} 3^P \prod_{p=1}^{P} \binom{M_{p,l} + B}{B} 2^B 2(\lceil n/B \rceil + P\ell)$$

for input size (output size respectively) n. The term $\binom{M_{p,l}+B}{B}$ describes the number of possible relations between the records of an input/output block and internal memory. In the time-forward analysis, it is the number of possibilities to choose the records from internal memory for an output, in the time-backwards analysis, it refers to the number of choices which records have been input. However, note that in an extracted program, the number of records transferred by an I/O is on average B/w. A better trace can be obtained when specifying the number of records that are transferred by each I/O. There are $B^{\ell P}$ possibilities to choose the ℓP values $b_{p,l}^{(i)} \leq B$. The values $m_{p,l}^{(i)}$ are determined by the previous I/Os considered in the I/O-trace, and the computation trace below. Knowing the number of records $b_{p,l}^{(i)}$ chosen for the considered I/O, there is no longer need to choose from B empty records (as provided by the term $M_{p,l} + B$ in the binomial coefficient). For the given values of $b_{p,l}^{(i)}$ and $m_{p,l}^{(i)}$, we get

$$\left[\prod_{l=1}^{\ell} 3^P \prod_{p=1}^{P} \binom{m_{p,l}^{(i)}}{b_{p,l}^{(i)}} 2^B 2 \left(\left\lceil \frac{H}{B} \right\rceil + P\ell\right)\right] \cdot B^{\ell P} \tag{4.12}$$

possibilities for the two I/O traces defining the sequence of abstract row and column configurations where we use $N_{\mathsf{x}}, N_{\mathsf{y}} \leq H$.

Because there are H elementary products created, and each elementary product requires only one record of $\mathbf{x}^{(i)}$, the total number of copies of $\mathbf{x}^{(i)}$ records at a time in internal and external memory is bounded above by H for a normalised program. Similarly, there are at most H partial results that are predecessors of the output vector $\mathbf{c}^{(i)}$. Furthermore, using the bound on $f^{(i)}$, we have $\sum_p \sum_l b_{p,l}^{(i)} \leq 2\ell PB/w$, and $\sum_p \sum_l m_{p,l}^{(i)} \leq 2\ell PM/w$ leading to the total number of reached abstract configuration being bounded above by

$$\left(3^P \binom{\min\{2\ell PM/w, \ell H\}}{2\ell PB/w} 2^{PB} 2^P \left(\left\lceil \frac{H}{B} \right\rceil + P\ell\right)^P\right)^{2\ell} \cdot B^{\ell P} \tag{4.13}$$

where we consider both abstract traces, leading to an exponent $2 \cdot \ell$.

The Computation Trace In the abstract configurations described above, sum and copy operations are ignored. It remains to consider the possible multiplication operations and their connected deletions of operands. For the latter consideration, there are at most 2^H possibilities whether to delete the $x_j^{(i)}$ operand of each multiplication. Additionally, in the sequence of abstract row configurations, the row index created by a multiplication operation can be present in internal memory before the multiplication operation, or be introduced as a new index. This contributes another factor 2^H.

Now, we determine the number of possible multiplications that can be performed in an extracted program with fixed I/O trace. We modified extracted programs such that a multiplication is performed as soon as both operands are in internal memory of a processor. After the lth input operation of processor p, there are at most $b_{p,l}^{(i)}$ new records of $\mathbf{x}^{(i)}$ introduced into internal memory. Each of which can be multiplied to create an elementary product contributing to one of the at most $m^{(i)}_{p,l+1}$ partial results in internal memory before the next I/O operation. This leads to $\prod_{p,l} b^{(i)}_{p,l} \cdot m^{(i)}_{p,l+1}$ possible positions for a multiplication operation (defining a non-zero entry of $\mathbf{A}$) which is maximised for $b^{(i)}_{p,l} = B$ and $m^{(i)}_{p,l+1} = M$ for $\ell P/w$ index pairs (p, l). Hence, there are at most $4^H \binom{MB\ell P/w}{H}$ different multiplication traces for a fixed I/O trace. Moreover, because for fixed l and i, $\sum_p m^{(i)}_{p,l} \le H$, the number of multiplication traces for a fixed I/O trace is also bounded by $4^H \binom{\ell BH}{H}$.

Note that maximising the term in this manner contrasts the table based algorithms where in each round, the numbers $b^{(i)}_{p,l}$ and $m^{(i)}_{p,l}$ are approximately B/w, M/w respectively, for all i, p, l. The table-based algorithm can hence not be matched by the lower bounds derived here. In the calculations of the lower bounds, we make use of the following technical lemmas which can be found in [BBF+10]. The proofs are cited here for completeness.

Lemma 4.7 (Lemma A.1 in [BBF+10]). *For every $x > 1$ and $b \ge 2$, the following inequality holds:* $\log_b 2x \ge 2 \log_b \log_b x$.

Proof from [BBF+10]. By exponentiating both sides of the inequality we get $2x \ge \log_b^2 x$. Define $g(x) := 2x - \log_b^2 x$, then, $g(1) = 2 > 0$ and

$$g'(x) = 2 - \frac{2}{\ln^2 b}\frac{\ln x}{x} \ge 2 - \frac{2}{\ln^2 b} \cdot \frac{1}{e} \ge 2 - \frac{2}{\ln^2 2} \cdot \frac{1}{e} > 0$$

for all $x \ge 0$, since $\ln(x)/x \le 1/e$. Thus g is always positive and the claim follows. □

Lemma 4.8 (Lemma A.2 in [BBF+10]). *Let $b \ge 2$ and $s, t > 0$. For all positive real numbers x, we have $x \ge \frac{\log_b(s/x)}{t}$ implies $x \ge \frac{1}{2}\frac{\log_b(s\cdot t)}{t}$.*

Proof from [BBF+10]. If $s \cdot t \le 1$, the implied inequality holds trivially. Hence, in the following we assume $s \cdot t > 1$. Assume $x \ge \log_b(s/x)/t$ and, for a contradiction also $x < 1/2 \log_b(s \cdot t)/t$. Then we get

$$\begin{aligned} x &\ge \frac{\log_b(s/x)}{t} > \frac{\log_b \frac{2s\cdot t}{\log_b(s\cdot t)}}{t} = \frac{\log_b(2s\cdot t) - \log_b \log_b(s\cdot t)}{t} \\ &\ge \frac{\log_b(2s\cdot t) - \frac{1}{2}\log_b(2s\cdot t)}{t} = \frac{1}{2}\frac{\log_b(2s\cdot t)}{t}, \end{aligned}$$

where the last inequality stems from Lemma 4.7. This contradiction to the assumed upper bound on x establishes the lemma. □

Calculations With the above discussion, we get

$$4^H\binom{\ell BPm/w}{H}\cdot\left(3^P\binom{4\ell Pm/w}{2\ell PB/w}2^{PB}\left(\left\lceil\frac{H}{B}\right\rceil+P\ell\right)^P\right)^{2\ell}B^{\ell P}\geq\binom{N_\mathsf{y}}{H}^{N_\mathsf{x}}$$

with $m=\min\{M,wH/P\}$ as a requirement for a family of programs with ℓ being able to evaluate SpMV for w vectors. W.l.o.g. we assume $N_\mathsf{x}\geq N_\mathsf{y}$ in the following. Taking logarithms, estimating binomial coefficients according to Observation 2.1, and rearranging terms yields

$$2\ell P\left(\frac{2B}{w}\log 4e\frac{m}{B}+\log 6PB\ell\right)\geq H\log\frac{N_\mathsf{x}N_\mathsf{y}/4H}{e\ell BPm/wH}\,.$$

This is equivalent to

$$\ell\geq\frac{H}{2P}\cdot\frac{\log\frac{wN_\mathsf{x}N_\mathsf{y}}{4e\ell BPm}}{\frac{2B}{w}\log 4e\frac{m}{B}+\log 6PB\ell}\,.$$

Applying Lemma 4.8 for $x=\ell$, $t=\frac{2P}{H}\left(\frac{2B}{w}\log 4e\frac{m}{B}+\log 6PB\ell\right)$, $s=\frac{wN_\mathsf{x}N_\mathsf{y}}{4eBPm}$, and estimating $t\geq\frac{2P}{H}\left(\frac{2B}{w}+\log 6PB\ell\right)$, we get

$$\ell\geq\frac{H}{2P}\cdot\frac{\log\left(\frac{wN_\mathsf{x}N_\mathsf{y}}{4eBPm}\cdot\frac{2P}{H}\left(\frac{2B}{w}+\log 6PB\ell\right)\right)}{\frac{2B}{w}\log 4e\frac{m}{B}+\log 6PB\ell}\,.$$

Now, it remains to distinguish according to the leading term in the denominator.

I. For the first case, we assume $\frac{2B}{w}\log 4e\frac{m}{B}\geq\log 6PB\ell$ which yields

$$\ell\geq\frac{wH}{8PB}\frac{\log\frac{N_\mathsf{x}N_\mathsf{y}}{eHm}}{\log 4e\frac{m}{B}}=\frac{wH}{8PB}\left(\frac{\log\frac{4N_\mathsf{x}N_\mathsf{y}}{HB}}{\log 4e\frac{m}{B}}-1\right).$$

Using again $\log 4e\frac{m}{B}\leq\frac{9}{2}\log\mathrm{d}$ for $\mathrm{d}=\max\left\{2,\min\left\{\frac{M}{B},\frac{wH}{PB}\right\}\right\}$, we get

$$\ell\geq\frac{wH}{36PB}\log_\mathrm{d}\frac{N_\mathsf{x}N_\mathsf{y}}{HB}-1$$

which matches asymptotically the sorting based algorithm.

II. If $\frac{2B}{w}\log 4e\frac{m}{B}<\log 6PB\ell$, we get

$$\ell\geq\frac{H}{4P}\frac{\log\left(\frac{wN_\mathsf{x}N_\mathsf{y}}{2eBHm}\log 6\ell PB\right)}{\log 6PB\ell}>\frac{H}{4P}\frac{\log\left(\frac{N_\mathsf{x}N_\mathsf{y}}{2eBH\min\{M,H/P\}}\log 2HB\right)}{\log 2HB}$$

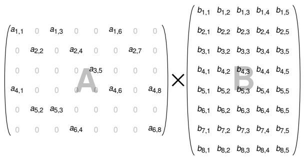
which matches asymptotically the bound obtained by the table based algorithm for many parameter settings. Note furthermore that $H < 2B$ makes the task trivially solvable with no more than 4 I/Os. Otherwise, it holds $\log 2HB \le 2 \log H$ so that we obtain

$$\ell \ge \frac{H}{16P} \frac{\log\left(\frac{N_{\mathrm{x}} N_{\mathrm{y}}}{BH \min\{M, H/P\}} \log H\right)}{\log H} .$$

4.4 Conclusion

We presented upper and lower bounds for the task of evaluating SPMV for a matrix $\mathbf{A}$ with $w \le B$ vectors simultaneously in this chapter. All the bound extend to BIL by the considerations in Chapter 3. This extends on previous work by Bender et al. [BBF⁺07] in that *multiple* vectors are multiplied with a *non-square* matrix. Additionally, we improve on previously published results [GJ10a] in that we consider the complexity in the *PEM model*, for matrices with *arbitrary density*. The presented lower bounds hold for an arbitrary number of processors. However, for a number of processors exceeding wH/B, a lower bound for the gather tasks dominates the I/O complexities.

Furthermore, the reduction to BIL yields algorithms and lower bounds for matrices in row major layout. Some of the algorithms for SPMV are non-uniform in that a proper load-balancing among the processors has to be determined in a preprocessing step. While this applies especially to column major layout, in the CREW PEM model, a row major layout does not reflect the problem of non-uniformity. Since the non-uniformity is required for a balanced gathering during the summing process, uniform algorithms are easily obtained for row major layout. The same holds for matrices in best-case layout with $N_{\mathrm{x}} \ge N_{\mathrm{y}}$ so that in the table-based algorithm, a table is created for the input vectors only.



5 Sparse × Dense

5.1 Introduction

We consider the multiplication of a sparse $N_y \times N_x$ matrix $\mathbf{A}$ containing H non-zero entries with a dense $N_x \times N_z$ matrix $\mathbf{B}$ for $N_z \geq B$. The I/O complexity of computing the $N_y \times N_z$ matrix $\mathbf{C} = \mathbf{A} \cdot \mathbf{B}$ over an arbitrary semiring is determined here. This is referred to as SDM, as mentioned in Chapter 1. We present lower bounds on the I/O complexity and upper bounds in form of algorithms that match up to constant factors. One of the three presented algorithms is non-uniform in the sense that we ask for a program that, depending on the conformation of $\mathbf{A}$, computes $\mathbf{C}$ with few I/Os. Such a program that is adapted to a certain matrix conformation can be generated in a preprocessing step which will be discussed in Section 5.3. Each of the presented algorithms is optimal within a certain parameter range. Furthermore, the complexity results for SDM are obtained for a best-case layout of the matrices $\mathbf{A}$, $\mathbf{B}$ and $\mathbf{C}$. However, in many cases transforming the layout of $\mathbf{A}$ from another layout is dominated by the I/Os for SDM. Obtaining the required layouts – mostly a column or row major layout suffices – is discussed in Section 5.2.1.

While the results in this chapter where published for square matrices in the I/O-model in [GJ10c], we extend the bounds to the PEM model as in the previous chapters, and consider non-square matrices. In contrast to the previous chapter where the number of processors was not limited, we restrict the number of processors here to $P \leq \frac{HN_z}{M^{3/2}}$. As we will see, this guarantees that internal memory of a processor is fully exploited: By our lower bounds, expressed below in equation (5.1), the number of I/Os required for SDM is $\Omega\left(\frac{HN_z}{PB\sqrt{M}}\right)$. Restricting the number of processors, each processor reads $\Omega(M)$ records. However, this number of processors still allows for algorithms that replicate the input. Using a larger number of processors will be discussed in the conclusion of this chapter.

Additionally, we propose an efficient algorithm to determine parts of the matrix $\mathbf{A}$ that are denser than average. This can be used as a preprocessing step in order to generate an efficient program for SDM within a certain parameter range. Given a tall cache $M \geq B^{1+\varepsilon}$ the parallel I/O complexity of SDM is

$$\Theta\left(\max\left\{\frac{HN_z}{PB\Delta}, \frac{HN_z}{PB\sqrt{M}}, \frac{H + N_xN_z + N_yN_z}{PB}, \log\frac{N_x}{B}\right\}\right) \tag{5.1}$$

where

$$\Delta = \max\left\{\frac{\ln\frac{N_x+N_y}{M}}{\overline{\ln}\left(\frac{N_xN_y}{HM}\ln^2\frac{N_x+N_y}{M}\right)}, \sqrt{\frac{HM}{N_xN_y}}\right\}$$

and we use $\overline{\ln}(x) = \ln(x)$ for $x > e$ and $\overline{\ln}(x) = 1$ otherwise.

This expression yields three interesting ranges of the density H/N_xN_y of $\mathbf{A}$. For all densities the I/O complexity boils down to the question how many of the HN_z elementary products can be performed on M records that are simultaneously in internal memory of a processor. This corresponds to the consideration of Hong-Kung rounds/sequences according to Section 2.1. For larger H/N_xN_y, the situation is similar to that of multiplying two dense matrices. In particular, for $H = N_yN_x$ it coincides with the classical result of Hong and Kung [HK81] that multiplying two square dense matrices in the I/O-model has complexity $\Theta\left(\frac{N^3}{B\sqrt{M}}\right)$, i.e, that at most $M^{\frac{3}{2}}$ multiplications per round are possible and can be achieved by using $\sqrt{M} \times \sqrt{M}$ tiles. This statement was extended to non-square situations in a parallel model by Irony et al. [ITT04] yielding a parallel I/O complexity of $\Theta\left(\frac{N_xN_yN_z}{BP\sqrt{M}}\right)$. A tile-based approach with differing tile-dimensions depending on the sparsity of $\mathbf{A}$ is presented here. For small density, (5.1) resembles the situation of $\mathbf{A}$ being a permutation matrix where M multiplications per round are best possible (i.e. loaded records cannot be reused).

Additionally, there is a density range where the complexity (given by the reuse of loaded operands) can be described by above average dense submatrices consisting of M entries and having on average $\min\{\Delta, \sqrt{M}\}$ entries per row and column. Our complexity analysis proceeds by showing that there exist matrices that have essentially no denser submatrices. We get a matching upper bound by showing that every matrix that has sufficiently many entries must have such dense submatrices. The resulting algorithm hence depends upon the conformation of the sparse input matrix in a complicated manner (which does not influence the theoretical statement). One key difference in the considerations here and the previous chapters is that in this chapter the block size B is basically irrelevant whereas SpMV becomes trivial for $B = 1$. It only matters when changing the layout of the matrices into the layout required for the optimal algorithm.

Observe that for $H \min\left\{\frac{N_x}{N_y}, \frac{N_y}{N_x}\right\} \leq M$ the complexity reduces to scanning the matrices. This case corresponds to a square segment of H with dimensions $\min\{N_x, N_y\}$ fitting entirely into internal memory. Let w.l.o.g. $N_x \geq N_y$. Since $\Delta \geq \sqrt{\frac{HM}{N_x N_y}}$, it holds that $\max\left\{\frac{HN_z}{PB\Delta}, \frac{HN_z}{PB\sqrt{M}}\right\} \leq \sqrt{\frac{HN_xN_y}{M}}\frac{N_z}{PB}$. For $HN_y/N_x \leq M$, this term is bounded above by $\frac{N_xN_y}{PB}$. An algorithm for this special case is stated in Section 5.2.5.

5.2 Algorithms

Theorem 5.1. SDM *is possible with*

$$\mathcal{O}\left(\min\left\{\frac{HN_z}{PB}, \sqrt{\frac{HN_xN_y}{M}}\frac{N_z}{B}, \frac{HN_z}{PBD}\right\} + \frac{H + N_xN_z + N_yN_z}{PB} + \log N_x\right)$$

I/Os for

$$D = \min\left\{\frac{\ln\frac{N_x+N_y}{M}}{\overline{\ln}\left(\frac{N_xN_y}{HM}\ln^2\frac{N_x+N_y}{M}\right)}, \sqrt{M}\right\}$$

if $M \geq B^{1+\varepsilon}$ (tall-cache assumption). Here, we use again

$$\overline{\ln}(x) := \begin{cases}\ln(x) & \textit{if } x > e\\ 1 & \textit{otw.}\end{cases}$$

This is equivalent to an I/O complexity of

$$\mathcal{O}\left(\max\left\{\frac{HN_z}{PB\Delta}, \frac{HN_z}{PB\sqrt{M}}, \frac{H + N_xN_z + N_yN_z}{PB}, \log\frac{N_x}{B}\right\}\right)$$

for

$$\Delta = \max\left\{\frac{\ln\frac{N_x+N_y}{M}}{\overline{\ln}\left(\frac{N_xN_y}{HM}\ln^2\frac{N_x+N_y}{M}\right)}, \sqrt{\frac{HM}{N_xN_y}}\right\}$$

given the tall-cache assumption, and a number of processors such that each processor reads at least $M = \Omega(\log B)$ records. Note that Δ is lower bounded by a constant.

It can be helpful to visualise a matrix multiplication in the following way. The computation can be considered a cuboid whose faces are described by the three matrices aligned at their common dimension such that opposing faces correspond to the same matrix (cf. Figure 5.1). A discrete point in the cuboid, described by the coordinates i, j, k, corresponds to the elementary product $a_{ij}b_{jk}$, which is required for the computation of c_{ik}. Since **B** (and, hence, **C**) are dense matrices while **A** is a sparse matrix, the non-zero entries of **A** induce poles orthogonal to **A** inside the cuboid.

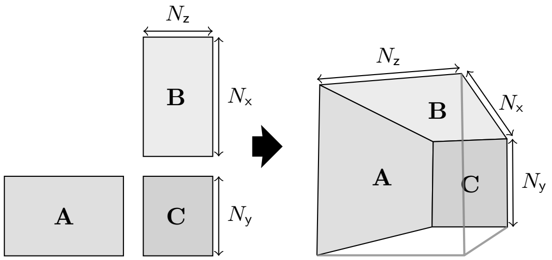


Figure 5.1: A cuboid view of matrix matrix multiplication.

For convenience, we omit the use of ceiling functions for fractions greater or equal 1 in the following. Since the number of products involving such terms in our calculations is a constant, this only increases the bounds by constant factors.

5.2.1 Layouts

As mentioned above, we assume the matrices **A**, **B** and **C** to be in best-case layout. The I/O cost for transforming the layouts is not counted in Theorem 5.1.

Note that given a permutation matrix **A**, SDM is the permutation of rows in **B**. For arbitrary **A** with H non-zero entries, each row in **C** can be considered a weighted sum over a set of rows of **B**. Using this intuition, row major layout seems to be a reasonable layout for **B** and **C**. It turns out that for a wide range of parameter settings – expressed in the optimality of two out of the three presented algorithms – row major layout leads indeed to an optimal number of I/Os. However, for the tile-based algorithm described in

Section 5.2.3, a column major layout allows for the separation into tiles (since we assume a tall cache). Recall that transposing **B** and **C** from column major to row major layout, and vice versa, is possible with $\mathcal{O}\left(\frac{(N_x+N_y)N_z}{PB}\right)$ I/Os, given the tall cache ($M \geq B^{1+\varepsilon}$). As we prove by lower bounds, a different layout than row or column major, chosen by the algorithm, does not lead to an asymptotic speed up.

The layout of the matrix **A** is unimportant for the direct algorithm. For all the other algorithms, the desired layout can be obtained by sorting. Sorting the records of **A** with the PEM merge sort from Section 2.6 induces $\mathcal{O}\left(\frac{H}{PB}\overline{\log}_d \frac{H}{B}\right)$ I/Os for $d = \max\{2, \min\{M/B, H/(PB)\}\}$. Transposing the matrix from column to row major layout (or vice versa) is considered in Chapter 7, and has I/O complexity

$$\mathcal{O}\left(\min\left\{\frac{H}{P}, \frac{H}{PB}\overline{\log}_d \min\left\{\frac{HB}{N_x N_z}, N_x, N_y\right\}\right\}\right).$$

5.2.2 Direct Algorithm

In the direct algorithm for permuting, each record is simply moved to its destination. This concept can be extended from permutation to arbitrary matrices, and from single records to rows that have to be moved/summed. We assume that **A**, **B** and **C** are given in row major layout. Let $\mathbf{b}_i$ be the ith row of **B**, and $\mathbf{c}_i$ the ith row of **C**. By one (parallel) scan of **A** while adding for each non-zero entry a_{ij} the product $a_{ij}\mathbf{b}_j$ to (a processor-owned copy on disk of) $\mathbf{c}_i$, partial results involving all the HN_z elementary products can be computed. Hence, a single processor can compute SDM with $\mathcal{O}\left(\frac{HN_z}{B}\right)$ I/Os.

In a multi-processor environment, an instance of SDM can be divided into subtasks using a layout where the matrices **B** and **C** are organised in $\min\{N_z/B, P\}$ meta-columns. Given the tall-cache assumption, a row major layout is sufficient for this, since a meta-column contains at least B columns, and hence, one block per row. For each meta-column, SDM can then be evaluated independently. If $P \leq N_z/B$, each subtask is solved by a single processor algorithm. Each processor scans **A** and creates the output for the N_z/P meta-columns of **C** with $\mathcal{O}\left(\frac{HN_z}{PB}\right)$ I/Os.

Otherwise, there are $p = PB/N_z$ processors assigned to each meta-column (consisting of B columns) of **B** and **C**. Then, **A** is divided among the p processors. For a proper load-balancing, we apply the range-bounded load-balancing from Section 2.7.3 where the row index of a record in **A** serves as key. Thus, each processor gets at most $2H/p$ non-zero entries from at most $\lceil 2N_y/p \rceil$ rows of **A** assigned to it. Each processor scans its assigned records of **A** and creates partial sums from rows of **C** for the meta-column assigned to it. This is done with $\mathcal{O}(H/p) = \mathcal{O}\left(\frac{HN_z}{PB}\right)$ I/Os.

Because several processors might have been assigned to the same row of **A** (thus created partial sums for the same row of **C**), partial results have to be gathered to form the final output. The gather process is invoked for each row of each meta-column separately, creating a single block only. Note that each processor is involved in at most two gather processes, and to each gather process, volume processors are assigned with contiguous id. If a processor is involved in two processes, it is the first and the last processor for the respective tasks. Hence, the gather processes can be performed in parallel without any collision. To this end, each processor determines whether it is assigned to the first or last record of a row in **A**, and if so, writes its index into a table of $2N_y$ blocks. This takes $\mathcal{O}(N_y/p) = \mathcal{O}(N_yN_z/PB)$ I/Os. Each processor that is involved in a gather task can then determine the range of volume processors (according to Section 2.7.3) that are assigned to the same row of **A**. With $\mathcal{O}(N_yN_z/PB + \log N_x/B)$ I/Os, partial sums of the volume processors are summed up by the gather processes. With another $\mathcal{O}(N_y/p)$ I/Os, the range processors add their partial sums to create the final output.

It turns out that for $H \le (N_xN_y/M)^{1-\varepsilon}$ and any constant $\varepsilon > 0$ this algorithm is asymptotically optimal because Δ in Theorem 5.1 becomes a constant and Theorem 5.6 yields a matching lower bound.

5.2.3 Tile-Based Algorithm

For denser cases of **A**, a modification of the tile-based algorithm from [KW03] clearly outperforms the direct algorithm. This modified algorithm works for any $H \ge \frac{N_xN_y}{M}$ and $M \le \min\{N_x/N_y, N_y/N_x\} \cdot H$. The other cases are covered by the algorithms in the remaining subsections. For the ease of notation, let $3M$ be the internal memory size. In this approach, the matrix **B** is assumed to be given partitioned into tiles of size $a \times \min\{M/a, N_z\}$ for $a = \sqrt{MN_xN_y/H}$. The output matrix **C** is generated in the same tiles as **B**, while **A** is given partitioned into tiles of size $a \times a$ (cf. Fig. 5.2). Note that this requires $a \le \min\{M, N_x, N_y\}$ in order that a tile does not exceed the dimensions of a matrix. This requirement is reflected in the conditions above for the algorithm to work. Let $\mathbf{A}_{ij}$, $\mathbf{B}_{ij}$, and $\mathbf{C}_{ij}$ denote the jth tile within the ith tile row of the respective matrix. Clearly, it holds $\mathbf{C}_{ij} = \sum_{l=1}^{n_x} \mathbf{A}_{il}\mathbf{B}_{lj}$ with $n_x = N_x/a$. Throughout the calculation of a certain tile $\mathbf{C}_{ij}$, partial results can be kept in internal memory while pairs of $\mathbf{A}_{il}$ and $\mathbf{B}_{lj}$ are loaded consecutively for each l. Since each **B**-tile contains at most M records, each such tile can be loaded in a whole and it can be kept in internal memory throughout the calculation of a $\mathbf{C}_{ij}$. Keeping the partial results of $\mathbf{C}_{ij}$ and the records of $\mathbf{B}_{lj}$ in internal memory, the records of $\mathbf{A}_{il}$ can be scanned to create all the required elementary products.

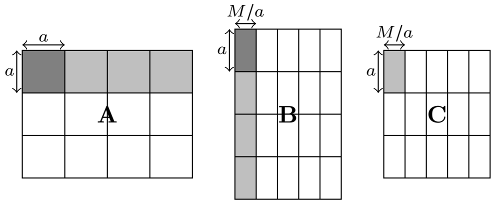


Figure 5.2: An illustration of the tiles in **A**, **B**, and **C**. The grey tiles in **A** and **B** are required for the grey tile in **C**.

In the single processor case, the following I/O complexity is obtained by this algorithm. For saving all **C**-tiles, no more than $\frac{N_y N_z}{B}$ I/Os are required. Each tile $\mathbf{A}_{il}$, and thus, each record in **A** has to be loaded for the calculation of all tiles of **C** in the corresponding row. There are $\lceil N_z / \frac{M}{a} \rceil$ tiles in a row of **C**. Therefore, loading the non-zero entries of **A** requires at most $\frac{H}{B} \cdot \max\left\{1, \frac{N_z\sqrt{N_x N_y}}{\sqrt{HM}}\right\}$ I/Os. Loading **B**-tiles costs at most $\frac{N_x N_z}{B} \cdot \frac{N_y}{a} = \sqrt{\frac{HN_xN_y}{M}}\frac{N_z}{B}$ I/Os. Altogether, this sums up to

$$\mathcal{O}\left(\max\left\{\sqrt{\frac{HN_xN_y}{M}}\frac{N_z}{B}, \frac{H}{B}\right\}\right)$$

I/Os for the computation of **C**.

The following considerations yield an efficient parallelisation of the above algorithm. Note that since the matrices **B** and **C** are dense, the ranges of their tiles in external memory can be determined by calculation, and are hence known to each processor. Similar to the direct algorithm, we first distribute **C** column-wise among the processors. Let $n_z := \lceil N_z a / M \rceil$ be the number of tiles per row in **C**. For $1 < P \le n_z$, each processor computes SDM for n_z/P columns of tiles of **C** by the described single processor approach. By the previous paragraph, loading **A** requires at most $\frac{H}{B} \cdot \frac{n_z\sqrt{N_xN_y}}{\sqrt{HM}}$ I/Os (note that $n_z > 1$). The access of **B**-tiles induces at most $\frac{N_x n_z}{B} \cdot \frac{N_y}{a}$ I/Os. This yields an algorithm with $\mathcal{O}\left(\sqrt{\frac{HN_xN_y}{M}}\frac{N_z}{PB}\right)$ I/Os.

For larger P, $p = P/n_z$ processors are assigned to each column of tiles. Each group of p processors is then assigned to $\mathbf{A}$ using the range-bounded load-balancing policy of Section 2.7.3 with the tile index (which are assumed to be ordered macroscopically in row major layout) as key. Hence, a processor is assigned to at most $2H/p$ records of $\mathbf{A}$ from at most $2N_\mathrm{x}N_\mathrm{y}/a^2p = 2H/Mp$ tiles. Each processor creates partial sums for a local copy of tiles of $\mathbf{C}$. For each new tile $\mathbf{A}_{il}$, the corresponding tile $\mathbf{B}_{lj}$ is loaded into internal memory. By scanning $\mathbf{A}_{il}$, elementary products for $\mathbf{C}_{ij}$ can be created. Loading records of $\mathbf{A}$ and $\mathbf{B}$ causes hence $\mathcal{O}\left(\frac{H/p}{B} + \frac{H}{Mp}\frac{M}{B}\right) = \mathcal{O}\left(\frac{Hn_z}{PB}\right) = \mathcal{O}\left(\max\left\{\frac{H}{PB}, \sqrt{\frac{HN_\mathrm{x}N_\mathrm{y}}{M}}\frac{N_\mathrm{z}}{PB}\right\}\right)$ I/Os, where we distinguished the cases $n_z = 1$ and $N_\mathrm{z}a/M > 1$. Again, the records of $\mathbf{C}$ are kept in internal memory of each processor until no more elementary products are created for this tile of $\mathbf{C}$. Each processor creates partial results for at most $\frac{2N_\mathrm{y}}{ap} + 1$ rows of tiles in $\mathbf{C}$. Hence, writing partial sums of $\mathbf{C}$ takes $\mathcal{O}\left(\frac{N_\mathrm{y}N_\mathrm{z}}{PB}\right)$ I/Os.

Finally, partial sums of multiple processors that were created for the same tile of $\mathbf{C}$ have to be gathered to form the final output. Similar to the gathering part of the direct algorithm, the range of volume processors that created partial results for the same tile of $\mathbf{C}$ can be identified. However, since each tile contains more than one block, the gather process itself is more involved than for the direct algorithm. Summing partial results from one tile can be serialised such that the gather process takes only $\mathcal{O}\left(\frac{N_\mathrm{y}N_\mathrm{z}}{PB} + \log\frac{N_\mathrm{x}}{B}\right)$ I/Os. Note that the gather processes of different tiles can interfere. To tackle this problem, first the gather processes of odd rows of tiles is invoked for the volume processors, and afterwards for even rows. Similarly, the range processors can gather their partial results to create the final output $\mathbf{C}$.

5.2.4 Using Dense Parts of A

In this section, we show that by loading M records from each matrix, even for $H < \frac{N_\mathrm{x}N_\mathrm{y}}{M}$ where the tile-based algorithm is not applicable, a number of $\omega(M)$ elementary products can be obtained, i.e. more than the direct algorithm achieves. This is done by loading denser than average parts of $\mathbf{A}$.

For the sake of illustration, we consider the matrix $\mathbf{A}$ as an adjacency matrix of a bipartite graph $G = (U \cup V, E)$, where $a_{ij} \neq 0$ constitutes a connection between the ith node of U and the jth node of V. Any induced subgraph reflects a submatrix in $\mathbf{A}$ (cf. Figure 5.3). If there are sufficiently many subgraphs containing $\mathcal{O}(M)$ edges, with average degree $\Omega(D)$, SDM is possible with $\mathcal{O}\left(\frac{HN_\mathrm{z}}{PBD}\right)$ I/Os.

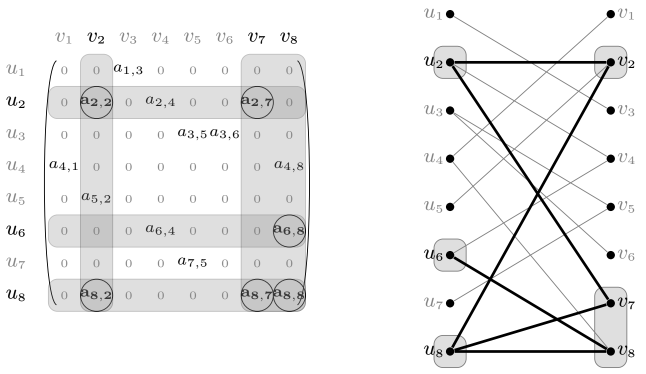

v_1 v_2 v_3 v_4 v_5 v_6 v_7 v_8
u_1 u_2 u_3 u_4 u_5 u_6 u_7 u_8
$a_{1,3}$
$a_{2,2}$ $a_{2,4}$ $a_{2,7}$
$a_{3,5}$ $a_{3,6}$
$a_{4,1}$ $a_{4,8}$
$a_{5,2}$
$a_{6,4}$ $a_{6,8}$
$a_{7,5}$
$a_{8,2}$ $a_{8,7}$ $a_{8,8}$

Figure 5.3: An induced subgraph and the corresponding submatrix.

Lemma 5.2. *Given a bipartite graph $G = (U \cup V, E)$, $|U| = N_x$, $|V| = N_y$, $|E| = H$ with $N_x \geq N_y \geq 8M$. Then, for $2N_x \leq H \leq \frac{N_x N_y}{32M} \ln^2 \frac{N_x}{M}$ there exist two non-empty subsets $X \subseteq U$, $Y \subseteq V$, such that the subgraph induced by X and Y has average degree at least*

$$D = \min\left\{ \frac{\ln \frac{N_x}{M}}{2\,\overline{\ln}\left(\frac{N_x N_y}{4MH} \ln^2 \frac{N_x}{M}\right)}, \sqrt{\frac{M}{2}}, \frac{H}{2N_x} \right\}$$

and it holds that $|X|, |Y| \leq M/D$.

Before showing this, we need the following lemma and the subsequent observations.

Lemma 5.3. *For $H \leq \frac{N_x N_y}{32M} \ln^2 \frac{N_x}{M}$, $N_x \geq N_y \geq 8M$, and D according to Lemma 5.2, the inequality*

$$H \leq \frac{N_x N_y D}{4M} \ln \frac{N_x}{M} \tag{5.2}$$

is satisfied.

Proof. We distinguish the three cases of the minimum for D in Lemma 5.2. If the first term dominates the minimum, substituting $D = \frac{\ln \frac{N_x}{M}}{2\ln\left(\frac{N_x N_y}{4HM} \ln^2 \frac{N_x}{M}\right)}$ in (5.2) yields

$$H \ln\left(\frac{N_x N_y}{4HM} \ln^2 \frac{N_x}{M}\right) \leq \frac{N_x N_y}{8M} \ln^2 \frac{N_x}{M}. \tag{5.3}$$

Observe that for $\frac{x}{k} \geq e$, the term $k \ln \frac{x}{k}$ for $x > 0$ is monotonically increasing in k. Its derivative is $\ln \frac{x}{k} - 1$, and hence, is non-negative for $\frac{x}{k} \geq e$. Since by assumption $\frac{N_x N_y}{4HM}\ln^2\frac{N_x}{M} \geq 8 > e$, we can substitute both appearances of H in (5.3) resulting in

$$\frac{N_x N_y \ln 8}{32M}\ln^2\frac{N_x}{M} \leq \frac{N_x N_y}{8M}\ln^2\frac{N_x}{M}$$

which is obviously true.

If the second term of the minimum in D applies, i.e. $D = \sqrt{M/2}$, we distinguish the cases $\frac{N_x N_y}{32M}\ln^2\frac{N_x}{M} < N_x N_y$ and vice versa. Note that in the latter case $H \leq N_x N_y$ is the only restriction on H. By substituting D and H in (5.2) both cases hold within the desired range: For the first case $\frac{N_x N_y}{32M}\ln^2\frac{N_x}{M} < N_x N_y$, multiplying both sides by the left-hand side and taking the square root on both sides, we get

$$\frac{N_x N_y}{32M}\ln^2\frac{N_x}{M} < \frac{N_x N_y}{4\sqrt{2M}}\ln\frac{N_x}{M} = \frac{N_x N_y D}{4M}\ln\frac{N_x}{M} .$$

Hence, this upper bound holds for H as well, yielding (5.2). For the other case, i.e. $\frac{N_x N_y}{32M}\ln^2\frac{N_x}{M} \geq N_x N_y$, taking the square root and multiplying both sides by $\sqrt{N_x N_y}$ yields

$$\frac{N_x N_y}{4\sqrt{2M}}\ln\frac{N_x}{M} \geq N_x N_y .$$

Substituting $\sqrt{2M} = M/D$, we obtain

$$\frac{N_x N_y D}{4M}\ln\frac{N_x}{M} \geq N_x N_y .$$

Hence, the right hand side of (5.2) is greater or equal $N_x N_y$, and since $H \leq N_x N_y$, the claim holds again.

Finally, for $D = H/2N_x$, (5.2) is equivalent to $H \leq \frac{HN_y}{8M}\ln\frac{N_x}{M}$ which obviously holds for $N_x \geq N_y \geq 8M$. □

Observation 5.4. *For $0 \leq x \leq 1/2$, it holds that $\ln(1-x) \geq -2x$.*

Proof. Consider $f(x) = \ln(1-x) + 2x$. Observe that $f(0) = 0$ and

$$f'(x) = \frac{-1}{1-x} + 2 = \frac{1}{x-1} + 2 \geq \frac{-1}{1/2} + 2 = 0 .$$

Hence, $f(x) \geq 0$ for $0 \leq x \leq 1/2$. □

Observation 5.5. *For $D, x, y > 0$, the inequality $D\ln(Dy) \leq x$ is satisfied for*

$$D \leq \frac{x}{\overline{\ln}(xy)} .$$

Proof. By substitution, we obtain

$$D\ln(Dy) \le \frac{x}{\overline{\ln}(xy)} \ln\left(y\frac{x}{\overline{\ln}(xy)}\right) \le \frac{x}{\overline{\ln}(xy)} \ln\left(y\frac{x}{1}\right) \le x$$

since $\overline{\ln}(x) = 1$ for $x \le e$ and thus, $\overline{\ln}(x) \ge 1$. □

Proof of Lemma 5.2. To show the existence of the sets X and Y, we consider $Y \subseteq V$, $|Y| = M/D$, chosen uniformly at random, and bound the expected number of nodes in U that have at least D neighbours in Y. Let $Z_u^Y \in \{0,1\}$ be the random variable indicating if $u \in U$ has at least D neighbours in Y. Showing that $\mathrm{E}\left[\sum_{u\in U} Z_u^Y\right] \ge M/D$ implies that there is at least one set Y such that M/D nodes from U have degree at least D into Y. These nodes from U constitute the set X.

To exploit linearity of expectation, and consider Z_u^Y for a single u only, we use a transformation of G with fixed degree for each node. To this end, G is transformed so that the maximal degree in U is restricted to at most $k/2$ for $k := H/N_\times$. Note that by assumption $k/2 \ge 1$. Each node $u_i \in U$ with degree $d_i > k/2$ is split into $u_{i,1}, \dots, u_{i,\lceil 2d_i/k\rceil}$ so that each new node has degree $k/2$ except at most one. Let U' denote this transformation of U, E' the transformed set of edges, and $G' = (U' \cup V, E')$ the created graph. By construction, the size of U will increase by no more than $2H/k = 2N_\times$, implying that $|U'| \le 3N_\times$. We can conclude that there are at least $N_\times$ nodes with degree $k/2$: Suppose that c nodes in the original set U have degree less than $k/2$. Hence, the degrees of the remaining $N_\times - c$ nodes sum up to at least $H - ck/2$. For each node in G with degree $d_i > k/2$, by construction of U', there will be at most one new node with degree less than $k/2$. This leads to no less than $H - ck/2 - (N_\times - c)k/2 = H/2$ edges that have to belong to nodes with degree $k/2$. Dividing the at least $H/2$ edges among nodes with degree $k/2$, there have to be at least $N_\times$ nodes with degree $k/2$. We call the subset of these nodes U''.

Observe that any subgraph G'_S in G' with average degree D consisting of nodes $X \subseteq U'$ and $Y \subseteq V$ can be transformed into a subgraph G_S of G with average degree at least D by simply replacing any $u_{i,j} \in X$ by the corresponding node u_i of the original graph. The subgraph G_S induced by X and the vertices corresponding to Y contains at least the edges of G'_S and no more nodes than G'_S. Hence, it suffices to show the existence of the desired X and Y for G'. We show $\mathrm{E}\left[Z_u^Y\right] \ge \frac{M}{D|U''|}$ for fixed $u \in U''$ which yields the result by linearity of expectation. Since $Z_u^Y \in \{0,1\}$, the aim is to find an appropriate D such that $\Pr\left[Z_u^Y = 1\right] \ge \frac{M}{DN_\times}$.

To estimate this probability, choose $Y \subseteq V$ uniformly at random and consider a vertex $u \in U''$. The number of vertices chosen for Y in the neighbourhood of u is given by a hypergeometric distribution. Choosing Y resembles drawing M/D times without replacement from an urn with N_y marbles, $k/2$ of which are black. The event we are interested in is that at least D of the drawn marbles are black.

We lower bound this probability by considering only the case of drawing precisely D black marbles. This probability is given by

$$\frac{\binom{k/2}{D}\binom{N_\mathrm{y}-k/2}{M/D-D}}{\binom{N_\mathrm{y}}{M/D}} = \frac{\binom{M/D}{D}\binom{N_\mathrm{y}-M/D}{k/2-D}}{\binom{N_\mathrm{y}}{k/2}}.$$

This well-known equality, which is easy to check, leads to the insight of expressing the probability as follows: Draw $k/2$ times from an urn with N_y marbles, M/D of which are black. Now consider drawing the $k/2$ marbles one after another, and fix precisely D positions where black marbles are drawn. The probability of such a drawing can be calculated as the product of the fractions of black (white) marbles that are left in the urn before each drawing. For black marbles the fraction is at least $p = (\frac{M}{D} - D)/N_\mathrm{y}$, for white it is at least $q = 1 - \frac{M}{D}/(N_\mathrm{y} - \frac{k}{2})$. In the following, we use $D \leq \sqrt{M/2}$, i.e. $D \leq \frac{M}{2D}$, and $k \leq N_\mathrm{y}$ to simplify these expressions. Hence, we obtain $p \geq \frac{M}{2DN_\mathrm{y}}$ and $q \geq 1 - \frac{2M}{DN_\mathrm{y}}$.

The overall probability of drawing D black marbles can then be bounded by summing the probabilities of all possible choices to position the D black marbles in the consecutive drawing. For Y_i being the number of black marbles drawn, we can lower bound the probability similar to a binomial distribution:

$$\Pr[Y_i = D] \geq \binom{k/2}{D} p^D q^{k/2-D} \geq \left(\frac{k}{2D}\frac{M}{2DN_\mathrm{y}}\right)^D \left(1 - \frac{2M}{DN_\mathrm{y}}\right)^{k/2}$$

where we used $q^{k/2-D} \geq q^{k/2}$ since $q < 1$, and Observation 2.1 to estimate the binomial coefficient. Taking logarithm yields

$$\ln \Pr[Y_i = D] \geq D \ln \frac{Mk}{4N_\mathrm{y}D^2} + \frac{k}{2}\ln\left(1 - \frac{2M}{N_\mathrm{y}D}\right) \geq D \ln \frac{Mk}{4N_\mathrm{y}D^2} - k\frac{2M}{N_\mathrm{y}D}$$

where the last inequality is justified by Observation 5.4 and $4M \leq N_\mathrm{y}$. Since we consider at least N_x nodes, the goal is now to choose the biggest D satisfying $\Pr[Y_i = D] \geq \frac{M}{DN_\mathrm{x}}$. This holds by implication if

$$D \ln \frac{4N_\mathrm{y}D^2}{Mk} + k\frac{2M}{N_\mathrm{y}D} \leq \ln \frac{N_\mathrm{x}D}{M} .$$

By Lemma 5.3, $H \leq \frac{N_\mathrm{x}N_\mathrm{y}D}{4M} \ln \frac{N_\mathrm{x}}{M}$, i.e. $k\frac{2M}{N_\mathrm{y}D} \leq \frac{1}{2} \ln \frac{N_\mathrm{x}}{M}$, holds. Hence, we are interested in

$$D \ln \frac{4N_\mathrm{y}D^2}{Mk} \leq \frac{1}{2} \ln \frac{N_\mathrm{x}}{M} + \ln D$$

which is implied by

$$D \ln \frac{2\sqrt{N_\mathrm{y}}D}{\sqrt{Mk}} \leq \frac{1}{4} \ln \frac{N_\mathrm{x}}{M} \qquad (5.4)$$

because $D \geq 1$.

Now, we can use Observation 5.5 with $y = \sqrt{\frac{4N_\mathrm{y}}{Mk}}$ and $x = \frac{1}{4} \ln \frac{N_\mathrm{x}}{M}$, and get the approximation

$$D \leq \frac{x}{\overline{\ln}\, xf} = \frac{\ln \frac{N_\mathrm{x}}{M}}{2\,\overline{\ln}\left(\frac{N_\mathrm{y}}{4Mk} \ln^2 \frac{N_\mathrm{x}}{M}\right)}$$

for which inequality (5.4) holds. □

In the following, we assume an internal memory of size $2M$ to ease notation. Consider a subgraph $G_S = (U_S \cup V_S, E_S)$ with average degree at least D and $|U_S|, |V_S| \leq M/D$. By construction of $G = (U \cup V, E)$, we considered a non-zero entry a_{ij} as an edge between u_i and v_j. Let I_U, I_V be the set of indices of vertices in U_S, V_S respectively. In order to create elementary products corresponding to E_S, the $m := |E_S| \leq M$ corresponding non-zero entries a_{ij} with $i \in I_U$, $j \in I_V$ have to be loaded. Then, for each column $1 \leq k \leq N_\mathrm{z}$ in $\mathbf{B}$ and $\mathbf{C}$, by loading all records b_{jk} with row indices $j \in I_V$ together, m elementary products can be obtained for $\mathbf{C}$ with row indices in I_U. Hence, by accessing m records of $\mathbf{A}$, and $N_z \cdot m/D$ records each in $\mathbf{B}$ and $\mathbf{C}$, partial results containing mN_z elementary products are created.

To efficiently load certain records of a column in $\mathbf{B}$, we extract these rows into a separate $|I_V| \times N_\mathrm{z}$ matrix and transpose it to column major layout. For a single processor, this is possible with $2\frac{N_\mathrm{z}m}{BD}$ I/Os since we assume $\mathbf{B}$ to be in row major layout. Then, records corresponding to a certain column can be loaded with at most $\frac{m}{DB}$ I/Os. Similarly, partial products can be stored into a $|I_U| \times N_\mathrm{z}$ matrix in column major layout. Transposing this, and adding the rows to the corresponding rows in $\mathbf{C}$ requires no more than $3\frac{N_\mathrm{z}m}{DB}$ I/Os. Hence, given a subgraph with m edges and average degree D, $N_\mathrm{z}m$ elementary products can be created with at most $6\frac{N_\mathrm{z}m}{DB} + \frac{m}{B} = \mathcal{O}\left(\frac{N_\mathrm{z}m}{DB}\right)$ I/Os.

Lemma 5.2 only states the existence of at least one dense subgraph. However, after creating all the elementary products corresponding to the edges of a dense subgraph, one can think of removing these edges. This will decrease the number of edges by m and we can use Lemma 5.2 for graphs with $H - m$ edges again. Clearly, half of the elementary products can be obtained by subgraphs with average degree at least $D(H/2)$ where $D(H/2)$ is obtained from Lemma 5.2 by substituting H with $H/2$. We describe in the next section how these subgraphs can be obtained in a preprocessing step by derandomising the proof of Lemma 5.2.

The $\frac{H}{2M}$ subgraphs of average degree at least $D(H/2)$ are divided equally among the P processors.For $P \leq \frac{H}{2M}$, the $HN_z/2$ elementary products can be created and saved with $\frac{H}{2PM} \cdot \left(\frac{N_z M}{BD(H/2)} + \frac{M}{B}\right) = \mathcal{O}\left(\frac{HN_z}{PBD(H/2)}\right)$ I/Os. In case $P > \frac{H}{2M}$, each subgraph is assigned to $p = \frac{2PM}{H}$ processors. We can hence divide the MN_z elementary products that are created for a subgraph among the p processors. Creating elementary products thus causes $\mathcal{O}\left(\frac{N_z M}{pBD(H/2)}\right) = \mathcal{O}\left(\frac{HN_z}{PBD(H/2)}\right)$ I/Os. Additionally, the extraction and transposition of rows of $\mathbf{B}$, and the transposition and summing of rows of $\mathbf{C}$ has to be parallelised. Extracting the rows can obviously be parallelised leading to $\mathcal{O}\left(\frac{N_z M}{pBD}\right)$ I/Os. Transposing these rows into a column major layout can be achieved with $\mathcal{O}\left(\frac{N_z M}{pBD} \overline{\log}_{d(N_z H/D)} B\right)$ I/Os by the PEM merge sort. Because we assume $P \leq \frac{HN_z}{M^{3/2}}$, we have $\frac{N_z M}{pBD} = \frac{HN_z}{2PBD} \leq \frac{M^{3/2}}{2BD} \leq \frac{M}{2B}$. The complexity for transposing the extracted rows of $\mathbf{B}$ is thus given by $\mathcal{O}\left(\frac{HN_z}{PBD} \log_{M/B} B\right) = \mathcal{O}\left(\frac{HN_z}{PBD}\right)$ since we assume a tall cache. Transposing the partial results of $\mathbf{C}$ has the same complexity. We displace the summing of generated rows to the very end of the algorithm. Evaluating the $H/2$ elementary products associated with the dense subgraphs yields a table of $\mathcal{O}\left(\frac{HN_z}{D}\right)$ records.

Let $D^{(1)}(H) = \frac{\ln \frac{N_x}{M}}{\overline{\ln}\left(\frac{N_x N_y}{4HM} \ln^2 \frac{N_x}{M}\right)}$, i.e. the first argument of the minimum of D in Lemma 5.2. Altogether, the number of I/Os necessary to create all elementary products for $\mathbf{C}$ is bounded above by

$$\begin{aligned} L(H,N) &\leq \frac{H}{PB} + \sum_{i=1}^{\infty} 6 \max\left\{\frac{2HN_z}{2^i PB\sqrt{M}}, \frac{HN_z}{2^i PBD^{(1)}(H/2^i)}\right\} \\ &\leq \frac{H}{PB} + \frac{12HN_z}{PB\sqrt{M}} + 6HN_z \sum_{i=0}^{\infty} \frac{\ln\left(\frac{N_x N_y}{4HM} \ln^2 \frac{N_x}{M}\right) + \ln 2^i}{2^i PB \ln \frac{N_x}{M}} = \mathcal{O}\left(\frac{HN_z}{PBD}\right). \end{aligned}$$

Observe that for $H \geq \frac{N_x N_y}{32M} \ln^2 \frac{N_x}{M}$, $\sqrt{\frac{H}{M}} = \Omega\left(\ln \frac{N_x}{M}\right)$ and thus, the tile-based algorithm is asymptotically better.

Assuming a preprocessing in order to identify dense subgraphs, we include the construction of an efficient scheduling to sum up the created partial sums in parallel. For the final summing, processors are assigned equally, row-wise to the table of partial sums such that each processor gets $\mathcal{O}\left(\frac{HN_{\mathrm{z}}}{PD}\right)$ contiguous records assigned to it. Let t be the number of rows of the table. Each row of the table can be associated with the row index of $\mathbf{C}$ it contributes partial results to. For $P \leq t$, each processor has to be assigned to multiple rows of the table. However, after the preprocessing step that identifies the dense subgraphs, the row indices of $\mathbf{C}$ associated with the rows of the table are determined. Hence, processors can be assigned within the summing process such that rows of the table are assigned greedily, ordered by the associated row index to the processors. In case $P \geq t$, each processor gets assigned to at most one row. By dividing the records of each row among the t/P processors that are assigned to each row at a common column index of $\mathbf{C}$, each processor is involved in only one gather process to form the output. Note that there are at most N_{x} rows of the table that are associated with the same row in $\mathbf{C}$. The gathering of partial results can be serialised, by determining within the preprocessing step which processors are assigned to rows that are associated with the same row index of $\mathbf{C}$. Hence, the output $\mathbf{C}$ is created in a gathering step with $\mathcal{O}\left(\frac{HN_{\mathrm{z}}}{PBD} + \log\min\{P, N_{\mathrm{x}}\}\right)$ I/Os.

5.2.5 Small Instances

For smaller instances where $M \geq \min\left\{\frac{N_{\mathrm{y}}}{N_{\mathrm{x}}}, \frac{N_{\mathrm{x}}}{N_{\mathrm{y}}}\right\} H$, the tile-based algorithm is not applicable whereas a degenerated version of the tile-based approach can be used once $M \geq \min\left\{\frac{N_{\mathrm{y}}}{N_{\mathrm{x}}}, \frac{N_{\mathrm{x}}}{N_{\mathrm{y}}}\right\} H + \min\{N_{\mathrm{x}}, N_{\mathrm{y}}\} + B$. With this algorithm, SDM can be computed with $\mathcal{O}\left(\frac{(N_{\mathrm{x}}+N_{\mathrm{y}})N_{\mathrm{z}}}{PB}\right)$ I/Os. To this end, $\mathbf{A}$ is divided into tiles of dimension $a \times a$ where $a = \min\{N_{\mathrm{x}}, N_{\mathrm{y}}\}$. W.l.o.g. assume $N_{\mathrm{x}} \geq N_{\mathrm{y}}$ in the following. Each tile that consists of more than $HN_{\mathrm{y}}/N_{\mathrm{x}}$ records is divided into subtiles so that each subtile (except at most one per tile) contains $HN_{\mathrm{y}}/N_{\mathrm{x}}$ records. The division into tiles of dimension $a \times a$ introduces at most $N_{\mathrm{x}}/N_{\mathrm{y}}$ borders to separate the records into tiles. Dividing the H non-zero records into subtiles that consist of at most $HN_{\mathrm{y}}/N_{\mathrm{x}}$ records can introduce another $N_{\mathrm{x}}/N_{\mathrm{y}}$ borders. Thus, the total number of (sub)tiles is at most $2N_{\mathrm{x}}/N_{\mathrm{y}} + 1$.

For the single processor case, one (sub)tile of $\mathbf{A}$ after another is loaded into internal memory with $\mathcal{O}\left(\frac{HN_{\mathrm{y}}}{N_{\mathrm{x}}B}\right)$ I/Os. Keeping records of the (sub)tile in memory, elementary products can be generated by loading records of

B column-wise. For each of the N_{z} columns of **B** and **C**, the N_{y} records of **C** corresponding to the current (sub)tile are loaded into internal memory. With the remaining block of internal memory, the N_{y} corresponding records in the column of **B** can be scanned, elementary products are created and summed to the records of **C**. Hence, with another $\mathcal{O}\left(\frac{N_{\mathrm{y}}N_{\mathrm{z}}}{B}\right)$ I/Os per (sub)tile, the output **C** is created. This results in a total I/O complexity of $2\frac{N_{\mathrm{x}}}{N_{\mathrm{y}}} \cdot \mathcal{O}\left(\frac{HN_{\mathrm{y}}}{N_{\mathrm{x}}B} + \frac{N_{\mathrm{y}}N_{\mathrm{z}}}{B}\right) = \mathcal{O}\left(\frac{H+N_{\mathrm{x}}N_{\mathrm{z}}}{B}\right)$ I/Os.

In a multiprocessor setting, processors are evenly assigned to (sub)tiles. Each processor creates partial results of **C** in a private area. In case $P \le 2N_{\mathrm{x}}/N_{\mathrm{y}}$, we apply the range-bounded load-balancing from Section 2.7.3 using the tile index as a key. Note that the tile index of a record in **A** is given by $\lceil i/N_{\mathrm{y}} \rceil$ where i is the column index. Each processor applies the single processor algorithm from above for its assigned area of **A**. This results in $\mathcal{O}\left(\frac{H}{PB} + \frac{N_{\mathrm{x}}N_{\mathrm{z}}}{PB}\right)$ I/Os for creating partial results. Theses partial results can then be summed up in a gathering phase similar to Section 5.2.3.

For $P > 2N_{\mathrm{x}}/N_{\mathrm{y}}$, the set of processors is divided into $p = PN_{\mathrm{y}}/2N_{\mathrm{x}}$ groups of $2N_{\mathrm{x}}/N_{\mathrm{y}}$ processors. Each group creates partial results for a range of N_{z}/p columns of **B** and **C**. The number of I/Os for reading **B** and writing **C** is thus $\mathcal{O}\left(\frac{N_{\mathrm{x}}N_{\mathrm{z}}}{PB}\right)$. However, each processor has to read a complete (sub)tile of **A**, inducing $\mathcal{O}\left(HN_{\mathrm{y}}/N_{\mathrm{x}}\right)$ I/Os. Recalling that we assume $P \le HN_{\mathrm{z}}/M^{3/2}$, we have $\frac{M}{B} \le \frac{HN_{\mathrm{z}}}{PB\sqrt{M}}$. Note that using $H \le N_{\mathrm{x}}N_{\mathrm{y}}$ the latter term is at most $\frac{\sqrt{HN_{\mathrm{x}}N_{\mathrm{y}}}N_{\mathrm{z}}}{PB\sqrt{M}}$ which in turn is bounded above by $\frac{N_{\mathrm{x}}N_{\mathrm{z}}}{PB}$ since $M \ge HN_{\mathrm{y}}/N_{\mathrm{x}}$. Hence, the I/O complexity is dominated by $\mathcal{O}\left(\frac{N_{\mathrm{x}}N_{\mathrm{z}}}{PB}\right)$.

5.3 Derandomisation

Lemma 5.2 from the previous section only proves the existence of two sets X, Y inducing a subgraph with M edges and average degree D. This suffices for an upper bound on the I/O complexity given a preprocessing step. Here, we show that such a dense subgraph can be found with $\mathcal{O}\left(N_{\mathrm{x}}N_{\mathrm{y}}\right)$ computational steps, and $\mathcal{O}\left(N_{\mathrm{x}}N_{\mathrm{y}}/B\right)$ I/Os. We describe the algorithm for the non-parallel case here.

Recall that by the proof of Lemma 5.2, $\mathrm{E}\left[\sum_{u \in U} Z_u^Y\right] \ge M/D$ for $Y \subseteq V$, $|Y| \le M/D$ chosen uniformly at random, and $Z_u^Y \in \{0,1\}$ is the random variable indicating if u has at least D neighbours in Y. Similar to the proof of Lemma 5.2, we consider the graph $G' = (U' \cup V, E')$ obtained by transformation from G such that $|U'| \le 3N_{\mathrm{x}}$, and nodes in U' have degree at most $k/2 = H/2N_{\mathrm{x}}$. As it is shown there, G' contains at least N_{x} nodes in U' that have degree exactly $k/2$. For the following considerations, we use the set of nodes in U' that have degree exactly $k/2$, denoted by U''. Given the matrix $\mathbf{A}$ in column major layout, the graph $G'' = (U'' \cup V, E'')$, $E'' = E' \cap (U'' \times V)$, can be obtained with one scan of $\mathbf{A}$ by dividing columns with more than $k/2$ non-zero entries, and ignoring columns with less than $k/2$ entries, inducing $\mathcal{O}(H/B)$ I/Os. With another H I/Os, the matrix can be transposed to supply the nodes of V with an adjacency list.

From now on, let $Z_u^Y \in \{0, 1\}$ indicate whether $u \in U''$ has exactly D neighbours in Y. Following standard techniques for derandomisation, we make use of the law of total probability, yielding

$$\mathrm{E}\left[\sum_{u \in U''} Z_u^Y\right] = \frac{1}{2}\mathrm{E}\left[\sum_{u \in U''} Z_u^Y \middle| v_i \in Y\right] + \frac{1}{2}\mathrm{E}\left[\sum_{u \in U''} Z_u^Y \middle| v_i \notin Y\right],$$

for any $v_i \in V$ and hence, $\mathrm{E}\left[\sum_{u \in U''} Z_u^Y \middle| v_i \in Y\right] \geq \frac{M}{D}$ or $\mathrm{E}\left[\sum_{u \in U''} Z_u^Y \middle| v_i \notin Y\right] \geq \frac{M}{D}$ has to hold. Furthermore, according to Lemma 5.2,

$$\mathrm{E}\left[\sum_{u \in U''} Z_u^Y\right] = \sum_{u \in U''} \Pr\left[Z_u^Y = 1\right] = \sum_{u \in U''} \mathcal{H}(N_{\mathrm{y}}, \frac{k}{2}, \frac{M}{D}, D)$$

where $\mathcal{H}(N, m, n, k) = \frac{\binom{m}{k}\binom{N-m}{n-k}}{\binom{N}{n}}$ is the hypergeometric distribution, drawing n times from an urn with N marbles, m of which yield a success, and the number of successful draws is k. In the following, we assume an arbitrary ordering $v_1, \ldots, v_{N_{\mathrm{y}}}$ of the nodes in V.

Now, consider the case $v_1 \in Y$ and the remaining $M/D - 1$ nodes in Y are drawn uniformly at random. This reduces the number of possible nodes to draw from to $N_{\mathrm{y}} - 1$, and only $M/D - 1$ nodes are drawn. For a node u which is not a neighbour of v_1, the probability becomes $\Pr\left[Z_u^Y = 1 \mid v_1 \in Y\right] = \mathcal{H}(N_{\mathrm{y}} - 1, \frac{k}{2}, \frac{M}{D} - 1, D)$. For the probability of a node u' adjacent to v_1, the number of successes (chosen neighbours) reduces to $k/2 - 1$, while the required number of neighbours that are chosen at random becomes $D - 1$. Hence the probability is $\Pr\left[Z_{u'}^Y = 1 \mid v_1 \in Y\right] = \mathcal{H}(N_{\mathrm{y}} - 1, \frac{k}{2} - 1, \frac{M}{D} - 1, D - 1)$.

In case, $v_1 \notin Y$ and the remaining $M/D - 1$ nodes in Y are chosen uniformly at random, only the number of nodes to choose from reduces to $N_{\mathrm{y}} - 1$ while it is still drawn M/D times. The probability $\Pr\left[Z_u^Y = 1 \mid v_1 \notin Y\right]$ for a node u not adjacent to v_1 is then given by $\mathcal{H}(N_{\mathrm{y}} - 1, \frac{k}{2}, \frac{M}{D}, D)$. For a node u' adjacent to v_1, the probability is $\Pr\left[Z_{u'}^Y = 1 \mid v_1 \notin Y\right] = \mathcal{H}(N_{\mathrm{y}} - 1, \frac{k}{2} - 1, \frac{M}{D}, D)$.

By calculating the probabilities for each node, the expected values for $\mathrm{E}\left[\sum_{u \in U''} Z_u^Y \mid v_1 \in Y\right]$ and $\mathrm{E}\left[\sum_{u \in U''} Z_u^Y \mid v_1 \notin Y\right]$ can be computed. Depending on which of the two expectations is larger, we either fix v_1 to be included in Y, or to be excluded from Y in the following. Since, by the law of total probability, the larger expectation is at least M/D, there exists a set $Y_1 \subseteq V$, $|Y_1| \leq M/D$, conform with the choice of including/excluding v_1 such that at least M/D nodes in U'' have D neighbours in Y. Subsequently, the probabilities over all $u \in U''$ are evaluated conditioned on both, including and excluding v_2 from Y_1. One node after another, the nodes of V can be included in, and excluded from Y, never reducing the expectation. Eventually, there are M/D nodes fixed to be included in Y. Determining M/D nodes that have degree at least D into Y yields the set X so that the required subgraph is obtained.

We describe in the following an algorithm for this derandomisation. For any setting of sets $A \subseteq Y$ of included nodes, and $B \cap Y = \emptyset$ of nodes that are excluded from Y, the probability for a node u to have D neighbours in Y can be calculated as follows. Let a_u be the number of neighbours of u in A, and b_u the number of neighbours in B. The probability for u is

$$\Pr\left[Z_u^Y \mid A \subset Y \wedge B \cap Y = \emptyset\right] = \mathcal{H}\left(N_{\mathsf{y}} - (|A| + |B|), \frac{k}{2} - (a_u + b_u), \frac{M}{D} - |A|, D - a_u\right)$$

if $a_u \leq D$ and $k/2 - b_u \geq D$, and $\Pr\left[Z_u^Y \mid A \subset Y \wedge B \cap Y = \emptyset\right] = 0$ otherwise (we used Z_u^Y synonymously with the event that $Z_u^Y = 1$). Observe that the denominator $\binom{N_{\mathsf{y}}-|A|-|B|}{M/D-|A|}$ of the hypergeometric distribution $\mathcal{H}$ is independent from the considered node. To distinguish the expectations, it is hence sufficient to normalise from probabilities to the number of possibilities to choose the remaining nodes for an appropriate Y. The change in the number of possibilities when including/excluding a node from Y can be expressed as follows. When including a new neighbour of a node, the binomial coefficient for choosing neighbours gets reduced. This causes a change by a factor $\binom{\frac{k}{2}-(a_u+b_u)-1}{D-a_u-1} / \binom{\frac{k}{2}-(a_u+b_u)}{D-a_u}$. Excluding a neighbour of a node changes the number of possible sets Y that fulfil Z_u^Y by a factor $\binom{\frac{k}{2}-(a_u+b_u)-1}{D-a_u} / \binom{\frac{k}{2}-(a_u+b_u)}{D-a_u}$.

Including a non-neighbour of a node changes the number of possibilities to choose the non-neighbours for Y. Thus, a factor

$$\frac{\binom{N_{\mathsf{y}} - (|A| + |B|) - \left(\frac{k}{2} - (a_u + b_u)\right) - 1}{\frac{M}{D} - |A| - (D - a_u) - 1}}{\binom{N_{\mathsf{y}} - (|A| + |B|) - \left(\frac{k}{2} - (a_u + b_u)\right)}{\frac{M}{D} - |A| - (D - a_u)}}$$

is contributed. Similarly changes the number of non-neighbours to choose from when excluding a non-neighbour. Using $\binom{n-1}{k} = \frac{n-k}{n}\binom{n}{k}$, and $\binom{n-1}{k-1} = \frac{k}{n}\binom{n}{k}$ yields Table 5.1. Since all nodes $u \in U''$ start with the same number of possibilities, it is furthermore sufficient to normalise to the sum of products of factors.

	Include v_i	Exclude v_i
Adjacent to u	$\frac{D-a_u}{\frac{k}{2}-(a_u+b_u)}$	$\frac{\frac{k}{2}-b_u-D}{\frac{k}{2}-(a_u+b_u)}$
Non-adjacent to u	$\frac{\frac{M}{D}-\lvert A\rvert-D+a_u}{N_\mathsf{y}-\lvert A\rvert-\lvert B\rvert-\frac{k}{2}+a_u+b_u}$	$\frac{N_\mathsf{y}-\lvert B\rvert-\frac{k}{2}+b_u-\frac{M}{D}+D}{N_\mathsf{y}-\lvert A\rvert-\lvert B\rvert-\frac{k}{2}+a_u+b_u}$

Table 5.1: Factors for the change of possibilities to have Y with $Z_u^Y = 1$ for a node u when including or excluding v_i from Y.

To this end, we annotate each node $u \in U''$ with the two values a_u and b_u, and a current factor γ_u. For each node v_i one after another, temporary γ_u values are calculated for the case of including and for the case of excluding v_i. Summing over these γ_u values, it is decided whether v_i will be included or excluded. Depending on this decision, the respective temporary γ_u values become the new current factors.

For the computation of the γ_u values, we assume that each node in V has its neighbours given in an adjacency list. To determine the temporary γ_u values for each node $u \in U$, U is scanned simultaneously with the adjacency list of v_i, keeping one block of the adjacency list, and one block of U in internal memory at a time. A new block of the adjacency list is loaded when all the nodes of the current block have been used for calculating their temporary γ_u value. Given the a_u and b_u values, the corresponding factor from Table 5.1 can be multiplied to γ_u which is then saved into a list of temporary γ_u-values. During this process, the algorithm checks if $a_u > D$ or $k/2 - b_u < D$ occurs for a node u which causes $\gamma_u = 0$.

This evaluation to decide whether v_i is included or excluded from Y incurs $\mathcal{O}\left(\frac{N_\mathsf{x}}{B}\right)$ I/Os. The total number of I/Os until appropriate sets Y and X are determined is hence $\mathcal{O}\left(\frac{N_\mathsf{x}N_\mathsf{y}}{B}\right)$, and the number computation operations is $\mathcal{O}(N_\mathsf{x}N_\mathsf{y})$.

5.4 Lower Bounds

Theorem 5.6. *For $1 \le k \le N$ any program for* SDM *needs*

$$\Omega\left(\max\left\{\frac{HN_z}{PB\Delta}, \frac{HN_z}{PB\sqrt{M}}, \frac{H + N_x N_z + N_y N_z}{PB}, \log\frac{N_x}{B}\right\}\right)$$

I/Os, with Δ according to Theorem 5.1.

Theorem 5.6 will be proven throughout this section. First observe that the last term of the maximum can be obtained by the lower bound from Section 2.4 corresponding to gather processes. For the other terms, we make use of the technique of partitioning programs into rounds/sequences of M/B I/Os which was introduced by Hong and Kung and is described in Section 2.1. Recall that in each sequence at most $2M$ input-records can be used to create partial results for at most $2M$ output-records. Upper bounding the number of elementary products that can be created within a sequence yields a lower bound on the number of sequences, hence bounding the number of I/Os from below.

The overall number of elementary products that have to be created for SDM is HN_z. Thus, it suffices to state an upper bound on the number of elementary products that can be created within an arbitrary sequence of M/B I/Os. We will do this by showing that there are matrices with only few dense submatrices of limited size. In other words, by loading M records from **A**, only few elementary products can be obtained. For the sake of illustration, we consider the matrix **A** as an adjacency matrix of a bipartite graph $G = (U \cup V, E)$, $|U| = N_x$, $|V| = N_y$, where $a_{ij} \neq 0$ constitutes a connection between the jth node of U and the ith node of V. By bounding the degree of subgraphs with at most M edges, a lower bound on the number elementary products created in a sequence can be stated.

To prove the statement, we require the following observations:

Observation 5.7. *For $0 < a \le e$, for any $x > 0$ it holds $x \ge a \ln x$.*

Proof. The first derivation of $f(x) = x - a\ln x$ is $f'(x) = 1 - a/x$. Hence, $x = a$ is an extremal point with function value $f(a) = a - a\ln a \ge 0$ since $\ln a \le 1$. The second derivation yields $f''(x) = 2a/x$ which is strictly positive for all values of $a, x \ge 0$. Thus, the extremal point is a minimum. □

Observation 5.8. *For $D, x, y \ge 0$ with $Dy > 1$, the inequality $D\ln(Dy) > x$ is fulfilled if*

$$D > \frac{2x}{\ln(2xy)}\,.$$

Proof. Substituting D yields

$$D\ln(Dy) > \frac{2x}{\ln(2xy)}\ln\left(y\frac{2x}{\ln(2xy)}\right) \geq \frac{2x}{\ln(2xy)}\ln\sqrt{2xy} = x$$

where we use $\sqrt{2xy} \geq 2\ln\sqrt{2xy}$ given by Observation 5.7. □

Observation 5.9. *For $n \geq k \geq a \geq 1$ it holds*

$$\binom{n}{k} \geq \left(\frac{n-k}{k}\right)^a \binom{n}{k-a}.$$

Proof. By definition of binomial coefficients

$$\binom{n}{k}\cdot\binom{n}{k-a}^{-1} = \frac{n!(n-k+a)!(k-a)!}{(n-k)!k!n!} = \prod_{i=1}^{a}\frac{n-k+i}{k-a+i} \geq \left(\frac{n-k}{k}\right)^a.$$

□

Lemma 5.10. *Let $\mathcal{G}$ be the family of bipartite graphs $G = (U \cup V, E)$ with $|U| = N_x$, $|V| = N_y$ and $|E| = H$ for $H \leq N_xN_y/2$.*

For every $M \leq H$ there is a graph $G \in \mathcal{G}$ such that G contains no subgraph $G_S = (U_S \cup V_S, E_S)$ with $|E_S| = M$ and average degree

$$D'_M > \max\left\{\frac{8\ln\frac{N_x+N_y}{2M}}{\overline{\ln}\left(\frac{16N_xN_y}{HM}\ln^2\frac{N_x+N_y}{2M}\right)}, e^4\cdot\sqrt{\frac{HM}{N_xN_y}}\right\}. \tag{5.5}$$

Proof. We will show this by upper bounding the number of graphs containing at least one such dense subgraph and compare this to the cardinality of $\mathcal{G}$. The upper bound is given by the number of possibilities to choose $2M/D'_M$ vertices from $U \cup V$ and the number of possibilities to insert M edges between the selected vertices. Furthermore, the remaining $H - M$ edges are chosen arbitrarily within the graph. The former presumes $2M/D'_M \leq N_x + N_y$. However, since $M \leq H$ and $D'_M > \sqrt{\frac{HM}{N_xN_y}}$ this is implied. Furthermore, we can assume $D'_M \leq \sqrt{M}$ since this is the maximum average degree of a subgraph consisting of M edges. Hence, if the inequality

$$\binom{N_x+N_y}{2M/D'_M}\binom{(M/D'_M)^2}{M}\binom{N_xN_y}{H-M} < \binom{N_xN_y}{H}$$

holds for the parameters given, Lemma 5.10 is proven. Observation 5.9 yields

$$\binom{N_x+N_y}{2M/D'_M}\binom{(M/D'_M)^2}{M} < \left(\frac{N_xN_y-H}{H}\right)^M.$$

Estimating binomial coefficients according to Observation 2.1, taking logarithms and multiplying by D'_M/M, we obtain

$$2\ln\frac{eD'_M(N_\mathsf{x}+N_\mathsf{y})}{2M}+D'_M\ln\frac{eM}{{D'_M}^2}<D'_M\ln\frac{N_\mathsf{x}N_\mathsf{y}}{H}+D'_M\ln\left(1-\frac{H}{N_\mathsf{x}N_\mathsf{y}}\right).$$

The last term can be estimated for $H\le N_\mathsf{x}N_\mathsf{y}/2$ by using $\ln(1-x)\ge -2x$ from Observation 5.4, resulting in

$$2\ln\frac{eD'_M(N_\mathsf{x}+N_\mathsf{y})}{2M}+D'_M\ln\frac{eM}{{D'_M}^2}<D'_M\ln\frac{N_\mathsf{x}N_\mathsf{y}}{H}-D'_M\frac{2H}{N_\mathsf{x}N_\mathsf{y}}.$$

And by simple equivalence transformations, we obtain

$$D'_M\ln\frac{{D'_M}^2N_\mathsf{x}N_\mathsf{y}}{HM}>\underbrace{2\ln\frac{N_\mathsf{x}+N_\mathsf{y}}{2M}}_{\text{Term 1}}+\underbrace{2\ln eD'_M+D'_M\left(1+2\frac{H}{N_\mathsf{x}N_\mathsf{y}}\right)}_{\text{Term 2}}. \tag{5.6}$$

Equation 5.6 is implied if Terms 1 and 2 are both bounded by $\frac{1}{2}D'_M\ln\frac{{D'_M}^2N_\mathsf{x}N_\mathsf{y}}{HM}$. We first check this for Term 2 only. By Observation 5.7, it holds $\ln(eD'_M)\le D'_M$. Thus,

$$\frac{1}{2}D'_M\ln\frac{{D'_M}^2N_\mathsf{x}N_\mathsf{y}}{HM}>2\ln(eD'_M)+2D'_M$$

is implied by $D'_M>e^4\cdot\sqrt{\frac{HM}{N_\mathsf{x}N_\mathsf{y}}}$ yielding the second term of the maximum in the final inequality (5.8). For any such D'_M, Inequality (5.6) holds if

$$D'_M\ln\frac{D'_M\sqrt{N_\mathsf{x}N_\mathsf{y}}}{\sqrt{HM}}>2\ln\frac{N_\mathsf{x}+N_\mathsf{y}}{2M}. \tag{5.7}$$

By substitution of D'_M by $e^4\cdot\sqrt{\frac{HM}{N_\mathsf{x}N_\mathsf{y}}}$, Inequality (5.7) already holds if $\sqrt{H}>\frac{1}{2e^4}\sqrt{\frac{N_\mathsf{x}N_\mathsf{y}}{M}}\ln\frac{N_\mathsf{x}+N_\mathsf{y}}{2M}$. For $\sqrt{H}\le\frac{1}{2e^4}\sqrt{\frac{N_\mathsf{x}N_\mathsf{y}}{M}}\ln\frac{N_\mathsf{x}+N_\mathsf{y}}{2M}$, we use Observation 5.8 with $y=\sqrt{\frac{N_\mathsf{x}N_\mathsf{y}}{HM}}$ and $x=2\ln\frac{N_\mathsf{x}+N_\mathsf{x}}{2M}$ yielding the first term of the maximum in (5.8). Altogether, for

$$D'_M>\max\left\{\frac{8\ln\frac{N_\mathsf{x}+N_\mathsf{y}}{2M}}{\overline{\ln}\left(\frac{16N_\mathsf{x}N_\mathsf{y}}{HM}\ln^2\frac{N_\mathsf{x}+N_\mathsf{y}}{2M}\right)},e^4\cdot\sqrt{\frac{HM}{N_\mathsf{x}N_\mathsf{y}}}\right\} \tag{5.8}$$

not all possible graphs in $\mathcal{G}$ are covered and therefore, Lemma 5.10 holds. Since the second term is a sufficient bound for any $\sqrt{H}>\frac{1}{2e^4}\sqrt{\frac{N_\mathsf{x}N_\mathsf{y}}{M}}\ln\frac{N_\mathsf{x}+N_\mathsf{y}}{2M}$, we use $\overline{\ln}$ – which is at least 1 – instead of $\ln$ to derive a closed formula by bounding the first term. Finally, note that $D'_M > 4$ holds for $H \geq \max\{N_\mathrm{x}, N_\mathrm{y}\}$. □

Lemma 5.11. *Let $\mathcal{G}$ be the family of bipartite graphs $G = (U \cup V, E)$ with $|U| = N_x$, $|V| = N_y$ and $|E| = H$ for $H \leq N_x N_y/2$.*

For any $M \leq H$, there is a graph $G \in \mathcal{G}$ such that G contains at most $M-1$ edges in subgraphs $G_S = (U_S \cup V_S, E_S)$ with $|E_S| \leq M$ and average degree $D' \geq 2e^4\Delta$ where Δ is defined according to Theorem 5.6.

Proof. By Lemma 5.10, this holds already for subgraphs consisting of exactly M edges. For smaller subgraphs, we prove the statement by contradiction.

Suppose that there are at least M edges in subgraphs with average degree at least D' consisting of less than M edges. Let $\mathcal{S}$ be the set of such subgraphs. Since each subgraph in $\mathcal{S}$ has less than M edges, there exists a subset S' of subgraphs in $\mathcal{S}$ with a total number of cM edges for $1 \leq c < 2$. The subgraph $G_{S'} = (U_{S'} \cup V_{S'}, E_{S'})$ induced by S' has obviously still average degree at least D'.

W.l.o.g. let $|U_{S'}| \geq |V_{S'}|$ and consider the vertices $U_{S'}$ in $G_{S'}$. Now choose the $\left\lceil \frac{M}{D'} \right\rceil$ vertices in $U_{S'}$ with highest degree, and let $U'_{S'}$ denote the set of these. Since the vertices $U_{S'}$ have average degree at least D' in $G_{S'}$, the subset $U'_{S'}$ cannot have a lower average degree than that. Hence, the subgraph $G'_{S'}$ induced by $U'_{S'}$ and $V_{S'}$ contains at least M edges, but consists of no more than $\frac{M}{D'} + \frac{cM}{D'} + 1$ vertices. Therefore, any subgraph induced by exactly M edges of $G'_{S'}$ has average degree at least $\frac{2MD'}{M+cM+D'}$. Since $D' \leq \sqrt{M}$, the average degree is at least $\frac{2D'}{2+c} \geq \frac{1}{2}D'$. This contradicts Lemma 5.10 for any $D' \geq 2D'_M$. □

With the use of this lemma, we can finally prove Theorem 5.6. To this end, we apply the method by Hong and Kung described in Section 2.1. Recall that Lemma 5.10, and thus, 5.11 fails for $D'_M > \sqrt{M}$. However, the maximum average degree of a subgraph with M edges is $\sqrt{M}$. The total number of elementary products necessary for SDM is HN_z. Thus, we use Lemma 2.4 with the potential Φ describing the number of elementary products such that the final potential is $\Phi(T) = HN_\mathrm{z}$. By Lemma 5.11, there are at most $N_\mathrm{z}(M-1)$ elementary products which might be calculated faster than the rest. Hence, we ignore the computation of these elementary products and let $\Phi(0) = N_\mathrm{z}(M-1)$. For the remaining $HN_\mathrm{z} - N_\mathrm{z}M + N_\mathrm{z}$ elementary products, the following holds. Within each sequence of M/B I/Os, there are at most $2M$ records of $\mathbf{B}$ and $\mathbf{C}$ loaded. Let s_{ij}, t_{ij} be the number of records from the jth column of $\mathbf{B}$, $\mathbf{C}$ respectively, loaded during the ith sequence. By Lemma 5.11 and the observation that any subgraph has degree at most $\sqrt{M}$, there can be made no more than $\sum_{j=1}^{N_z} \min\{D', \sqrt{M}\} \cdot s_{ij}t_{ij} = 2M \cdot \min\{D', \sqrt{M}\}$ elementary products in each sequence. This implies a potential change by each sequence of $\Delta(M) \le 2M \cdot \min\{D', \sqrt{M}\}$. Hence, there have to be at least

$$\frac{HN_z - N_zM + N_z}{2M \cdot \min\left\{D', \sqrt{M}\right\} P}$$

sequences per processor which yields a lower bound of

$$\frac{M}{B}\left(\frac{HN_z - N_zM + N_z}{2M \cdot \min\left\{D', \sqrt{M}\right\} P} - 1\right) = \Omega\left(\max\left\{\frac{HN_z}{PB\Delta}, \frac{HN_z}{PB\sqrt{M}}\right\}\right)$$

I/Os for SDM given that $M \le H/2$.

Note that in our lower bound, we also considered matrices that contain empty columns or rows, i.e. without any non-zero entries, because we did not require that each vertex has degree at least 1. However, by adding $\max\{N_x, N_y\}$ to any matrix, a matrix without any empty columns or rows can be obtained. Hence, the lower bounds hold for matrices with H non-zero entries that have no empty columns or rows, by using $H - \max\{N_x, N_y\}$ as the number of edges. This does not change the statement asymptotically for any $H \ge 2 \cdot \max\{N_x, N_y\}$.

5.4.1 Closing the Parameter Range

Recall that Lemma 5.11 only holds for $H \le N_xN_y/2$, and we assumed $H \ge 2M$. However, $\Omega\left(\max\left\{\frac{HN_z}{PB\Delta}, \frac{HN_z}{PB\sqrt{M}}\right\}\right)$ is a lower bound for $N_xN_y/2 \le H \le N_xN_y$ as well since increasing the number of non-zero entries in $\mathbf{A}$ cannot decrease the number of I/Os. For $H \le M$ a scanning bound of $\Omega\left(\frac{H+N_xN_z+N_yN_z}{PB}\right)$ holds for reading the inputs $\mathbf{A}$, $\mathbf{B}$, and writing the output $\mathbf{C}$. As argued in Section 5.1, the scanning bound dominates the term $\max\left\{\frac{HN_z}{PB\Delta}, \frac{HN_z}{PB\sqrt{M}}\right\}$ for $H \le M$.

5.5 Conclusion

We presented upper and lower bounds for the task of multiplying a sparse matrix with a dense matrix, for arbitrary dimensions $N_x, N_y, N_z \ge B$ and number of non-zero entries H in the sparse matrix. Assuming a tall cache

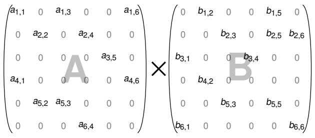
$M \geq B^{1+\varepsilon}$, upper and lower bounds are matching up to constant factors. In order to show that a program for the non-uniform algorithm can be constructed in polynomial (preprocessing) time, we presented a deterministic strategy to identify a subgraph, limited by the number of edges, with a guaranteed degree above the average degree for every bipartite graph. Together with Chapter 4, this yields lower bounds, and asymptotically optimal algorithms, for multiplying a sparse matrix with an arbitrary number of dense vectors – or equivalently multiplying a sparse matrix with a dense matrix with arbitrary dimensions. However, the lower bounds of both chapters rely on significantly different arguments.

For the results presented in this chapter, we restricted ourselves to the case $P \leq \frac{HN_z}{M^{3/2}}$. However, all the algorithms can be extended to larger P as long as $\frac{HN_z}{P} \geq B^{3/2+\varepsilon}$. This allows for transposing $\mathbf{B}$ and $\mathbf{C}$ in scanning time. To this end, internal memory can simply be restricted to a smaller virtual internal memory of size M' such that $\frac{HN_z}{\sqrt{M'}P} = M'$, i.e. $M' = \left(\frac{HN_z}{P}\right)^{2/3}$. The algorithms can obviously be executed for PEM parameters B, P, and $M' < M$.

For a lower bound on the number of I/Os, we can argue as follows. After ℓ I/Os, internal memory of each processor can contain at most ℓB records. Hence, for $\ell B < M$, we consider a sequence of ℓ I/Os and apply the arguments for Hong-Kung sequences from Section 2.1 similar to Section 5.4. Assuming $\ell B \leq \frac{1}{2}H$, at least $H/2$ non-zero entries cannot be assigned to a subgraph with average degree more than $\min\left\{D'_{\ell B}, \sqrt{\ell B}\right\}$ by Lemma 5.11. The number elementary products involving these $H/2$ non-zero entries that can be created in the sequence of ℓ I/Os is thus bounded above by $2\ell BP\min\left\{D'_{\ell B}, \sqrt{\ell B}\right\}$. Calling for this term to be at least $HN_z/2$ leads to a lower bound on the number of I/Os of

$$\Omega\left(\max\left\{\frac{HN_z}{PBD'_{\ell B}}, \frac{(HN_z)^{3/2}}{P^{3/2}B}, \frac{H + N_xN_z + N_yN_z}{PB}\right\}\right) \tag{5.9}$$

for SDM. Finally, for $M > \ell B > \frac{1}{2}H$, a scanning lower bound is sufficient for matching upper and lower bounds. However, this case is included in (5.9) as explained in the last paragraph of the introduction when applying $M' = \ell B < M$. These complexities are obviously matched by the presented algorithms restricted to smaller virtual internal memory M'.



6 Sparse × Sparse

6.1 Introduction

Having considered the multiplication of a sparse matrix with a dense matrix, the natural question arises which I/O complexities are implied when multiplying a sparse matrix with another sparse matrix (SSM). This is also an interesting problem in database queries [ACP10]. Multiplying two sparse matrices reveals some similarities to the join operation in relational database systems in that the two matrices are joined by the column index of **A** and the row index of **B**. However, if not combined with other operations, a join operation does not involve any process similar to the summation of elementary products. When requiring the output in column major layout, a join operation that is similar to matrix multiplication could be expressed by the following mySQL statement.

```
SELECT t1.value, t2.value FROM t1 JOIN t2 ON t1.column=t2.row
ORDER BY t2.column, t1.row
```

Of course, this similarity holds for dense matrix multiplication as well. Nevertheless, tables in database applications are usually rather sparse.

Multiplying a sparse matrix **A** with a single sparse vector changes the considerations in Chapter 4 only in that some columns of **A** become irrelevant. Ignoring the non-zero entries of **A** within the not required columns only changes the layout of **A**. Especially, if $N_x > H/B$, i.e. the average column spans more than one block, the asymptotic complexities correspond to the multiplication with the matrix $\mathbf{A}'$ where the irrelevant columns are removed. However, when considering the multiplication of **A** with another sparse matrix $\mathbf{B}$ with arbitrary conformation, all the non-zero entries of $\mathbf{A}$ can be important again.

For simplicity of exposition, we restrict ourselves in this chapter to the case of multiplying square matrices. The number of elementary products that have to be created for the matrix $\mathbf{C} = \mathbf{A} \cdot \mathbf{B}$ strongly depends on the conformations of $\mathbf{A}$ and $\mathbf{B}$, and so does the number of non-zero entries in $\mathbf{C}$. For the simple task of multiplying an $N \times N$ matrix $\mathbf{A}$ that contains one dense column (w.l.o.g. let the first column contain only non-zero entries) with the $N \times N$ matrix $\mathbf{B}$ having a dense row in matching dimension (the first row contains only non-zero entries), N^2 elementary products and N^2 entries of $\mathbf{C}$ have to be created. In this case, the I/O complexity is dominated by the output of the dense matrix $\mathbf{C}$ which requires $\Theta\left(\frac{N^2}{PB}\right)$ I/Os. The efficient estimation of the size of the result matrix $\mathbf{C}$ is a problem of its own (see e.g. [ACP10]) that we will not tackle here.

Hence, we restrict ourselves to matrices that are regular in their columns or rows, i.e. that have a fixed number of non-zero entries per column or row. The product can be written as $\mathbf{C} = \sum_{i=1}^{N} \mathbf{a}_i \mathbf{b}_i$ where $\mathbf{a}_i$ is the ith column vector of $\mathbf{A}$ and $\mathbf{b}_i$ is the ith row vector of $\mathbf{B}$. Observe that a matrix $\mathbf{C}$ can have at most $k_1 k_2 N$ entries in the following two cases: If $\mathbf{A}$ is k_1-column regular, i.e. has k_1 non-zero entries in each column, the number of elementary products is $\sum_{i=1}^{N} k_1 \cdot b_i = k_1 k_2 N$ where b_i is the number of non-zero entries in the ith row of $\mathbf{B}$. Similarly, if $\mathbf{B}$ is k_2-row regular, i.e. contains k_2 non-zero entries per row, each non-zero entry in $\mathbf{A}$ is multiplied with k_2 records of $\mathbf{B}$ such that there are $k_1 k_2 N$ elementary products that have to be evaluated. Obviously, the number of elementary products bounds the number of non-zero entries in $\mathbf{C}$ from above. However, only for $k_1 k_2 < N$ this yields a non-trivial upper bound.

In the worst-case, for any $k_1 k_2 \leq N/14$, the result matrix $\mathbf{C}$ contains at least $k_1 k_2 N/4$ non-zero entries. To this end, consider $\mathbf{A}$ as the adjacency matrix of a bipartite graph with degree k_1. We aim to find $\mathbf{A}$ such that any subset of k_2 nodes on one side has at least $k_1 k_2/4$ neighbours. In Lemma 6.1, it is shown that such a bipartite (expander) graph exists. Then, any sum of k_2 column vectors of $\mathbf{A}$ contains at least $k_1 k_2/4$ non-zero entries. Since each column vector in $\mathbf{C}$ is a (weighted) sum of column vectors of $\mathbf{A}$, $\mathbf{C}$ contains at least $k_1 k_2 N/4$ non-zero entries for such $\mathbf{A}$ if $\mathbf{B}$ is k_2-column regular.

Lemma 6.1. *For $k_1 k_2 \leq \frac{N}{14}$, there is a bipartite graph $G = (U \cup V, E)$ with $|U| = |V| = N$ where every node in U has degree k_1, and any set $S \subseteq U$ of k_2 nodes has at least $\frac{k_1 k_2}{4}$ neighbours.*

Proof. Several considerations of expanders are presented in Chapter 8. Thus, we refer the reader kindly forward to Lemma 8.17 presented in Chapter 8. There, the class of bipartite graphs $G = (U \cup V, E)$ with $|U| = N_1$, $|V| = N_2$, degree D for each node in U and degree $N_1 D/N_2$ for each node in V is considered. For this class, the existence of a graph in the class is shown where each subset $S \subseteq U$ of size $|S| \leq N_2/((1-\varepsilon)e^{3/\varepsilon}D)$ has a neighbourhood of size $(1-\varepsilon)D|S|$. Setting $\varepsilon = \frac{3}{4}$ and using $k_1 k_2 \leq 4N/e^4$, there is a graph for $N_1 = N_2 = N$ and $D = k_1$ such that each set $S \subseteq U$ of size $|S| \leq k_2$ has at least $\frac{k_1 k_2}{4}$ neighbours in V. $\square$

Unfortunately, SSM seems more difficult to analyse than the previous tasks. The lower bounds derived from the techniques in Chapter 2 are matching our algorithms only for few parameter ranges. With the counting technique applied in Chapter 4 for a lower bound on SPMV, the impact of the block structure of external memory on SSM can be exploited. However, it turns out that upper and lower bounds are matching only for the case that k_1 and k_2 are smaller than B, and smaller than $N/(2B)$. As we will see, especially the complexity of the direct algorithm cannot be obtained by this technique.

The fact that we have matching upper and lower bounds only if $k_1, k_2 \leq B$ seem to indicate a similar separation of the complexities as for SPMV with multiple vectors, where different techniques where required for $w < B$ and $w > B$ (which we considered as SDM). Hence, we apply a technique similar to Chapter 5 which ignores the block structure. These bounds are usually most useful for the case $B = 1$, but naturally imply a lower bound for arbitrary block size, by a factor $1/B$ weaker. On the algorithmic side, an extension of the tile-base algorithm to SSM can be shown to perform well. However, its optimality can only be shown within a special class of algorithms. An algorithm in this class creates in each sequence of M/B I/Os only partial results for $\mathbf{C}$ that lie within a submatrix, consisting of the intersection of a set of columns and a set of rows, with at most M records.

Similar to Chapter 5, the existence of denser than average parts of the matrices can be shown. However, different than for SDM, the existence of a single denser zone cannot be used directly to obtain a fast (non-uniform) algorithm. Nevertheless, we state the proof of the existence to show what bounds on the density can be achieved.

We make use of a probabilistic argument based on the graph representation of the matrix product of $\mathbf{A}$ and $\mathbf{B}$. Consider each matrix as the adjacency matrix of a bipartite graph, where each non-zero entry a_{ij} in row i and column j induces an edge from U_i to V_j. Recall that in matrix matrix multiplication, the number of columns of $\mathbf{A}$ and the number of rows of $\mathbf{B}$ correspond. Let $G_A = (U \uplus V, E_A)$ be the bipartite graph described by $\mathbf{A}$ and

$G_B = (V \uplus W, E_B)$ be the bipartite graph of $\mathbf{B}$. Then, the combined graph is $G = (U \uplus V \uplus W, E_A \uplus E_B)$ (cf. Figure 6.1). This is a common representation used for sparse matrix multiplication (see e.g. [Coh98]). Any path of length 2 from a node $u_i \in U$ over $v_j \in V$ to a node $w_k \in W$ can now be considered as an elementary product $a_{ij} \cdot b_{jk}$, which is thus part of the resulting entry c_{ik}. Consequently, if there is no path of length 2 from some $u_i \in U$ to $w_k \in W$, the position c_{ik} in the result matrix will be zero. On the other hand, note that if a subgraph contains multiple paths (u_i, v_j, w_k) for fixed i, k including several nodes $v_{j_1}, \dots, v_{j_D}$ the partial sum for c_{ik} consisting of the D elementary products $a_{ij_1}b_{j_1k} + \dots + a_{ij_D}b_{j_Dk}$ can be computed using only edges (non-zero entries) present in the subgraph.

6.2 Algorithms

6.2.1 Direct Algorithm and its Variants

As usually, for the direct algorithm, each elementary product is created by accessing the required records directly, and adding the result to the partial sum in $\mathbf{C}$. The k_1k_2N elementary products can be created in an arbitrary order, each inducing one access to records of $\mathbf{A}$ and $\mathbf{B}$, and possibly a partial result of $\mathbf{C}$, and one output of the partial sum for $\mathbf{C}$. Thus, the result matrix

$$
\begin{array}{c}
\\ u_1 \\ u_2 \\ u_3 \\ u_4 \\ u_5 \\ u_6
\end{array}
\begin{array}{c}
\\
\begin{pmatrix}
0 & 1 & 0 & 0 & 1 & 0 \\
1 & 0 & 0 & 1 & 0 & 0 \\
0 & 1 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 & 1 \\
1 & 0 & 0 & 0 & 0 & 1 \\
0 & 0 & 1 & 0 & 1 & 0
\end{pmatrix}
\end{array}
\cdot
\begin{array}{c}
w_1 w_2 w_3 w_4 w_5 w_6 \\
\begin{pmatrix}
0 & 0 & 1 & 0 & 1 & 0 \\
0 & 1 & 0 & 0 & 1 & 0 \\
0 & 0 & 1 & 1 & 0 & 0 \\
1 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 & 1 \\
1 & 0 & 0 & 0 & 1 & 0
\end{pmatrix}
\end{array}
=
\begin{array}{c}
w_1 w_2 w_3 w_4 w_5 w_6 \\
\begin{pmatrix}
0 & 1 & 1 & 0 & 1 & 1 \\
1 & 1 & 1 & 0 & 1 & 0 \\
0 & 1 & 1 & 1 & 1 & 0 \\
2 & 1 & 0 & 0 & 1 & 0 \\
1 & 0 & 1 & 0 & 2 & 0 \\
0 & 0 & 2 & 1 & 0 & 1
\end{pmatrix}
\end{array}
\begin{array}{c}
\\ u_1 \\ u_2 \\ u_3 \\ u_4 \\ u_5 \\ u_6
\end{array}
$$

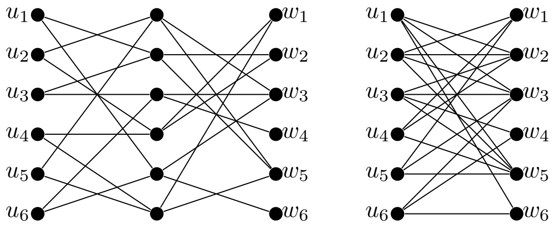


Figure 6.1: Sparse matrix multiplication considered as a graph. Each path of length 2 from left to right stands for an elementary product.

$\mathbf{C}$ can be created with $\mathcal{O}(k_1k_2N)$ I/Os by a single processor. When using $P \leq \frac{N^2}{B}$ processors, each processor can be assigned to compute the results in $\frac{N^2}{PB}$ blocks of $\mathbf{C}$. Then, $\mathcal{O}\left(\frac{k_1k_2N}{P}\right)$ I/Os are sufficient.

Recall that if $\mathbf{B}$ is a dense matrix, the direct algorithm for SDM from Chapter 5 only induces $\mathcal{O}\left(\frac{k_1N^2}{B}\right)$ I/Os. In the same sense, the evaluation of elementary products can be performed for SSM with only $\mathcal{O}\left(\frac{k_1k_2N}{PB}\right)$ I/Os: Multiplying each record of $\mathbf{B}$ with the records in a single column of $\mathbf{A}$ induces $\mathcal{O}\left(\frac{k_2N}{P}\left\lceil\frac{k_1}{B}\right\rceil\right)$ I/Os if $\mathbf{A}$ is in column major layout. However, organising the results in a desired layout for $\mathbf{C}$, and especially summing up elementary products presents a problem.

If a column major layout is desired and $\mathbf{B}$ is k_2-column regular, the following can be done. The k_1k_2N elementary products are evaluated as described above with $\mathcal{O}\left(\max\left\{\frac{k_1k_2N}{PB}, \frac{k_2N}{P}\right\}\right)$ I/Os, where $\mathbf{B}$ is read column-wise. The results are written to disk in the order they are created. Hence, there are k_1k_2 elementary products that belong to the same column of $\mathbf{C}$ written as contiguous records, consisting of k_2 runs of k_1 elementary products are ordered by row index. Then, it is possible to merge the k_2 runs to form a column of $\mathbf{C}$. This takes $\mathcal{O}\left(\frac{k_1k_2N}{PB}\log_{\mathrm{d}} k_2\right)$ I/Os for $\mathrm{d} = \max\left\{2, \min\left\{\frac{M}{B}, \frac{k_1k_2N}{PB}\right\}\right\}$, and with another scan, elementary products that belong to the same output record c_{ij} can be summed up.

If $\mathbf{A}$ is k_2-row regular, and $\mathbf{B}$ is in row major layout, $\mathbf{C}$ can be created in a row major layout with $\mathcal{O}\left(\max\left\{\frac{k_1N}{P}, \frac{k_1k_2N}{PB}\right\} + \frac{k_1k_2N}{PB}\log_{\mathrm{d}} k_1\right)$ I/Os in the same manner.

6.2.2 Sorting-Based Algorithm

In the following, we describe an algorithm for the case that $\mathbf{A}$ and $\mathbf{B}$ are both in column major layout, and $\mathbf{C}$ is required in column major layout as well. Furthermore, we assume again that $\mathbf{B}$ is k_2-column regular. First, the columns of $\mathbf{B}$ are used as pre-sorted runs for the PEM merge sort to form k_2 meta-columns which are internally ordered row-wise, each consisting of N records. According to the description of sorting-based algorithms for SPMV in Section 4.2.2, this merging is possible with $\mathcal{O}\left(\frac{k_2N}{PB}\log_{\mathrm{d}}\frac{N}{\max\{B,k_2\}}\right)$ I/Os.

Now, for each meta-column, the matrix $\mathbf{A}$ is scanned simultaneously with the records of the meta-column to create all the elementary products that involve records from the current meta-column. Because the meta-columns are in row-major layout and $\mathbf{A}$ is in column major layout, one scan of $\mathbf{A}$ per meta-column in $\mathbf{B}$ is sufficient. Thus, this step requires $\mathcal{O}\left(\frac{k_1k_2N}{PB}\right)$ I/Os. Note that each record of $\mathbf{B}$ is involved in k_1 elementary products. After scanning

A and creating elementary products in a meta-column, there are hence k_1N elementary products created for this meta-column which are still in some arbitrary ordering.

Sorting the elementary products that result from a meta-column of **B** by their column index (and row index within a column) in the output **C** takes $\mathcal{O}\left(\frac{k_1N}{PB}\log_{\mathrm{d}}\frac{k_1N}{B}\right)$ I/Os. Performing the sorting for all of the k_2 meta-columns yields a column major layout of all elementary products of **C**. With another scan, elementary products that belong to the same position in **C** are summed together. A possible gather step induces $\mathcal{O}(\log N)$ I/Os and is justified for column major layout by the lower bound in Section 2.4. Altogether, this algorithm has an I/O complexity of $\mathcal{O}\left(\frac{k_1k_2N}{PB}\log_{\mathrm{d}}\frac{k_1N}{B}+\log N\right)$ which can be shown to be optimal for $k_1, k_2 \leq B$.

6.2.3 Tile-Based Algorithm

This algorithm is an extension of the tile-based algorithm in Section 5.2.3, which in turn was based on an algorithm for dense matrix multiplication (see [KW03]). The output matrix **C** is generated, partitioned into $x \times y$ tiles with $x \cdot y = M$ As described in Lemma 6.1, for $k_1k_2 \leq N/14$ there are graphs where the number of elementary products in a tile of **C** is only a constant factor higher than the number of matrix positions in the tile. Hence, this algorithm only becomes interesting when $k_1k_2 > N$. The layout of the input matrices is assumed to be given in meta-rows and meta-columns according to the tiles in **C**. We assume that **A** is given in meta-rows of x rows which are internally ordered in a column major layout. The matrix **B** is given in meta-columns of y columns that are each ordered row-wise. For the ease of notation, we assume an internal memory size of $3M$ for this algorithm. This will, however, not change the asymptotic I/O complexity.

Let $x = \sqrt{Mk_2/k_1}$ and $y = \sqrt{Mk_1/k_2}$. For each $x \times y$ tile in **C**, we compute its content by scanning simultaneously through the meta-row of **A** consisting of the x corresponding rows and meta-column of **B** that consists of the y corresponding columns. This yields all the required records to create all elementary products for the tile. In this scanning, the (at most) M partial results of the tile are kept in memory, and for each $1 \leq i \leq N$ the ith column of the meta-row in **A** is read together with the ith row of the meta-column in **B**. Since the number of rows in a meta-column of **A** is less than M, all the records of a column from this area can be kept in memory. The same holds for **B**.

The number of tiles in a row of **C** is $\lceil N/y \rceil$. Similarly, we have $\lceil N/x \rceil$ tiles per column in **C**. Recall again that a record in **A** is only required for tiles of $\mathbf{C}$ that lie in the same row. Hence, each of the k_1N records of $\mathbf{A}$ is read at most $\lceil N/y \rceil$ times. Similarly, each of the k_2N records of $\mathbf{B}$ is read at most $\lceil N/x \rceil$ times.

Hence, in the serial case, the number of accesses to $\mathbf{A}$ is bounded by $\frac{k_1N}{B}\lceil N/y \rceil = \left\lceil \frac{N^2\sqrt{k_1k_2}}{B\sqrt{M}} \right\rceil$ I/Os. The same number of I/Os are sufficient for reading $\mathbf{B}$. For writing the results, $\mathcal{O}\left(N^2/B\right)$ I/Os suffice. Altogether, the algorithm requires $\mathcal{O}\left(\frac{N^2}{B}\left(\sqrt{\frac{k_1k_2}{M}}+1\right)\right)$ I/Os in the serial case. Thus, the tile-based algorithm can asymptotically outperform the direct algorithm for $k_1k_2 \geq N^2/M$ by a factor of $\frac{N}{\sqrt{k_1k_2M}}$.

For any number of processors $P \leq N^2/B$, the tiles can be computed separately by different processors. This reduces the number of parallel I/Os to $\mathcal{O}\left(\frac{N^2}{PB}\left\lceil\sqrt{\frac{k_1k_2}{M}}\right\rceil\right)$.

6.3 Lower Bounds

6.3.1 Lower Bound for By Counting Arguments

From Chapter 4, we can derive the following simple reduction. If $\mathbf{B}$ is a matrix with $k_2/2 \leq B$ dense columns, we can apply Theorem 4.5 which is stated for SPMV with w vectors, by setting $w = k_2/2$, $N_{\mathrm{x}} = N_{\mathrm{y}} = N$, and $H = k_1N$. Note that such a $\mathbf{B}$ can especially be k_2-row regular, and recall that in the proof of Theorem 4.5, column-regular matrices $\mathbf{A}$ are investigated. Hence, the considered case of matrix multiplication are covered by the theorem. This yields a lower bound of

$$\Omega\left(\min\left\{\frac{k_1N\log\left(\frac{k_2N}{k_1B\min\{M,k_1k_2N/P\}}\log k_1N\right)}{P\log k_1N}, \frac{k_1k_2N}{PB}\log_{\mathrm{d}}\frac{N}{k_1B}\right\}\right).$$

Similarly, by exchanging the roles of $\mathbf{A}$ and $\mathbf{B}$ in the reduction, assume that $\mathbf{A}$ is a (k_1-column regular) matrix with $k_1/2 \leq B$ dense rows. Thus, we obtain a lower bound of

$$\Omega\left(\frac{k_1k_2N}{PB}\log_{\mathrm{d}}\frac{N}{\min\{k_1,k_2\}B}\right)$$

for the case that $\min\{k_1,k_2\} \leq B\log_{\mathrm{d}} N$. This comes close to the complexity of the sorting-based algorithm and is asymptotically matching if furthermore $k_1 \leq N/(2B)$ holds.

The reduction is most obvious for the case that $\mathbf{B}$, or $\mathbf{A}$, contains dense columns, or rows respectively. However, the bounds hold basically in any setting where $\mathbf{B}$ contains at least k_2/c records in each row for some constant $c \geq 1$, or $\mathbf{A}$ contains at least k_1/c records per column.

6.3.2 Lower Bound for a Class of Algorithms

In the following, we construct a lower bound on the number of I/Os any program P for SSM induces if each sequence of M/B I/Os in P writes only partial results from $\mathbf{C}$ to disk that lie within a (pseudo rectangular) submatrix consisting of at most M positions. We refer to this class of algorithms as **pseudo rectangular algorithms**. Similar to Chapter 5, a pseudo rectangular submatrix consists of the matrix entries that belong to the intersection of a set of columns and a set of rows (permuting rows and columns in the matrix yields a rectangular submatrix). Hence, we require that the product of the number of selected columns and the number of selected rows is at most M. In other words, the bound holds for any program P where within any sequence of M/B I/Os in P, at least a constant fraction of the elementary results that can be created, are created and are predecessors of $\mathbf{C}$.

While this seems to be a somewhat arbitrary assumption, any such algorithm reveals spacial locality when writing and summing partial results of $\mathbf{C}$. For block size $B > 1$, it seems desirable to write partial results in a structured layout which enables fast summation of partial results for the same output-record. We conjecture that the presented bound holds for the worst-case I/O complexity of any algorithm for SSM. Unfortunately, we where not able to prove the general case.

Theorem 6.2. *There are two $N \times N$ matrices $\mathbf{A}$ and $\mathbf{B}$ where $\mathbf{A}$ contains $k_1 N$ non-zero entries and $\mathbf{B}$ contains $k_2 N$ non-zero entries such that any pseudo rectangular algorithm requires*

$$\Omega\left(\frac{k_1 k_2 N}{PB \min\left\{D, \sqrt{M}\right\}}\right)$$

I/Os for

$$D = \max\left\{\frac{(7-4\varepsilon)\log N}{\log\left(\frac{N^2}{8k_1k_2M}\log^2 N\right)}, 2e\frac{\sqrt{k_1k_2M}}{N}\right\}. \tag{6.1}$$

if $M \leq N^{1-\varepsilon}$ with constant $\varepsilon > 0$. Furthermore, $\mathbf{A}$ and $\mathbf{B}$ are regular in one dimension such that at most $k_1 k_2 N$ elementary products have to be created.

We prove the theorem in the remainder of this section. Similar to the lower bound in the previous chapter, we make use of the technique of considering the progress within sequences of M/B I/Os, like introduced in [HK81] by Hong and Kung and described for the PEM model in Section 2.1. Recall that within one sequence of M/B I/Os, each single processor can perform computations involving at most $2M$ records: M records that are read throughout the sequence and another M records that resided in internal memory at the beginning of the sequence. Similarly, at most $2M$ results can be written to external memory or reside in internal memory at the end of the sequence. Hence, only $2M$ results can serve as predecessors of the output.

Since there are k_1k_2N elementary products that have to be created for SSM, it is sufficient to bound the number elementary products that can be created within a sequence of M/B I/Os by a single processor. For a given upper bound D, we can apply Lemma 2.4 where the number of elementary products created so far serves as the potential Φ. Hence, we have $\Phi(0) = 0$, $\Phi(\ell) = k_1k_2N$ and an upper bound on the change of the potential cause by a single processor within a sequence of M/B I/Os $\Delta \leq D$. Lemma 2.4 then yields Theorem 6.2 if the bound on D is shown.

Recall that in Chapter 5 it was sufficient to consider denser submatrices with at most M records in $\mathbf{A}$ only. In these considerations, the bound was stated for submatrices with exactly M records because too many smaller dense submatrices imply that there is a dense submatrix with exactly M records of $\mathbf{A}$. The number of records from $\mathbf{B}$ that are loaded and the number of partial results written to external memory for $\mathbf{C}$ only influence the argument of Hong Kung sequences. If both $\mathbf{A}$ and $\mathbf{B}$ are sparse matrices, we have to consider denser submatrices in $\mathbf{A}$ and $\mathbf{B}$ that consist together of at most M records. In this case, it could be possible that one of the submatrices always consists of strictly less than M records, even if both submatrices together contain M records. To account for these asymmetric situations, we parameterise the number of records that are accessed during one sequence of M/B I/Os.

To lower bound the number of I/Os of a program that creates all the required k_1k_2N elementary products (each of them being a predecessor of an output-record), we upper bound the progress of an arbitrary round, i.e. the number of elementary products created as a predecessor of the output. To this end, let $\mathcal{M}_{\mathbf{A}}$ and $\mathcal{M}_{\mathbf{B}}$ be the records of $\mathbf{A}$ and $\mathbf{B}$ that are involved in computations within the considered sequence, and let $\mathcal{M}_{\mathbf{C}}$ be the results serving as predecessors of the output $\mathbf{C}$. Obviously, we have $|\mathcal{M}_{\mathbf{A}}| + |\mathcal{M}_{\mathbf{B}}| \leq 2M$ and $|\mathcal{M}_{\mathbf{C}}| \leq 2M$. Let us now consider possible distributions of records in $\mathcal{M}_{\mathbf{A}}$, $\mathcal{M}_{\mathbf{B}}$ and $\mathcal{M}_{\mathbf{C}}$ within the matrices $\mathbf{A}$, $\mathbf{B}$ and $\mathbf{C}$ that lead to many elementary products. Similar to the previous chapters, we assume that programs are normalised as described in Chapter 2 such that only elementary products are created which are a predecessor of an output-record (i.e. no product is created which is deleted afterwards). We show that, for any $M \leq N^{1-\varepsilon}$, the overall number of elementary products that can be created by a single processor within a sequence of M/B I/Os is bounded from above by $5DM$. To this end, we first carve out interesting parts of $\mathcal{M}_\mathbf{A}$, $\mathcal{M}_\mathbf{B}$ and $\mathcal{M}_\mathbf{C}$ as follows.

For the ease of notation, let $M_i = |\mathcal{M}_i|$ for $i \in \{\mathbf{A}, \mathbf{B}, \mathbf{C}\}$ from now on. Furthermore, we assume an internal memory of $M/2$ such that $M_i \leq M$, $i \in \{\mathbf{A}, \mathbf{B}, \mathbf{C}\}$, holds. Following the proof of the lower bound for dense matrix multiplication in [HK81], the following observations can be made: Firstly, note that each record in $\mathbf{B}$ can only be multiplied with records from a single column in $\mathbf{A}$. Consider the set of columns in $\mathbf{A}$ with at most $D_\mathbf{A} := DM/M_\mathbf{B}$ records in $\mathcal{M}_\mathbf{A}$ so that each record from $\mathbf{B}$ can be used to create only $D_\mathbf{A}$ elementary products. Since only $M_\mathbf{B}$ records from $\mathbf{B}$ can be involved in computations, at most $D_\mathbf{A} M_\mathbf{B} = DM$ elementary products can be created involving columns of $\mathbf{A}$ with no more than $D_\mathbf{A}$ records in $\mathcal{M}_\mathbf{A}$. Note that there are at most $M_\mathbf{A}/D_\mathbf{A} = M_\mathbf{A} M_\mathbf{B}/DM$ columns of $\mathbf{A}$ with more than $D_\mathbf{A}$ records in $\mathcal{M}_\mathbf{A}$. Similarly, each record in $\mathbf{A}$ can be multiplied with records from one row in $\mathbf{B}$. Any row in $\mathbf{B}$ with no more than $D_\mathbf{B} := DM/M_\mathbf{A}$ records in $\mathcal{M}_\mathbf{B}$ can lead to at most $D_\mathbf{B} M_\mathbf{A} = DM$ elementary products. It remains to consider indices $i \in \{1, \ldots, N\}$ such that $\mathcal{M}_\mathbf{A}$ contains at least $D_\mathbf{A}$ records from column i and $\mathcal{M}_\mathbf{B}$ contains at least $D_\mathbf{B}$ records from row i. The set of such indices has size at most $M_\mathbf{A} M_\mathbf{B}/DM$, which yields a bound on the interesting parts of $\mathcal{M}_\mathbf{A}$ and $\mathcal{M}_\mathbf{B}$ that do not trivially lead to at most $2DM$ elementary products.

By similar arguments, the number of interesting rows in $\mathbf{A}$ and $\mathbf{C}$ is limited to $\frac{M_\mathbf{A} M_\mathbf{C}}{DM}$, and the number of interesting columns in $\mathbf{B}$ and $\mathbf{C}$ is at most $\frac{M_\mathbf{B} M_\mathbf{C}}{DM}$: Each partial result in $\mathbf{C}$ contains only elementary products involving

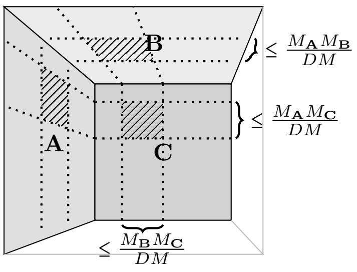

B
A
C
$\leq \frac{M_\mathbf{A} M_\mathbf{B}}{DM}$
$\leq \frac{M_\mathbf{A} M_\mathbf{C}}{DM}$
$\leq \frac{M_\mathbf{B} M_\mathbf{C}}{DM}$

Figure 6.2: Parts of the matrices that can lead to a faster computation. In this case, the pseudo rectangular submatrices are rectangles, which can be achieved by permuting rows and columns in $\mathbf{A}$, $\mathbf{B}$ and $\mathbf{C}$.

a single row of $\mathbf{A}$ (and a single column of $\mathbf{B}$). Hence, all rows of $\mathbf{A}$ with less than $DM/M_\mathbf{C}$ records in $\mathcal{M}_\mathbf{A}$ contribute to at most DM elementary products, and there are at most $M_\mathbf{A}M_\mathbf{C}/DM$ rows with more records in $\mathcal{M}_\mathbf{A}$. The number of columns in $\mathbf{B}$ with more than $DM/M_\mathbf{C}$ records in $\mathcal{M}_\mathbf{B}$ is limited to $\frac{M_\mathbf{B}M_\mathbf{C}}{DM}$. Altogether, the overall number of elementary products that can be created with records (i) from columns of $\mathbf{A}$ with at most $D_\mathbf{A}$ records in $\mathcal{M}_\mathbf{A}$, (ii) from rows of $\mathbf{B}$ with few records in $\mathcal{M}_\mathbf{B}$, (iii) from rows of $\mathbf{A}$ and (iv) from columns of $\mathbf{B}$ with few records loaded, is upper bounded by $4DM$.

Only an intersection of $\frac{M_\mathbf{A}M_\mathbf{B}}{DM}$ columns of $\mathbf{A}$ and rows of $\mathbf{B}$, with $\frac{M_\mathbf{A}M_\mathbf{C}}{DM}$ rows in $\mathbf{A}$, or $\frac{M_\mathbf{B}M_\mathbf{C}}{DM}$ columns in $\mathbf{B}$ respectively, can lead to more than DM elementary products in the sequence. Hence, we can assume in the following for an upper bound on D that all records of $\mathcal{M}_\mathbf{A}$, $\mathcal{M}_\mathbf{B}$ and $\mathcal{M}_\mathbf{C}$ lie within such interesting parts. All this holds for any algorithm, not only those that create pseudo rectangular submatrices in $\mathbf{C}$. In the following, we restrict ourselves to algorithms that create a pseudo rectangular submatrix with $M_\mathbf{C} \le M$ records in $\mathbf{C}$. We identify the extend of such a pseudo rectangle by its number of rows $\gamma \le \frac{M_\mathbf{A}M_\mathbf{C}}{DM}$ and the number of columns $M_\mathbf{C}/\gamma \le \frac{M_\mathbf{B}M_\mathbf{C}}{DM}$.

A part in $\mathbf{A}$ that can lead to more than DM elementary products in a pseudo rectangular submatrix of $\mathbf{C}$ has dimensions at most $\gamma \times \frac{M_\mathbf{A}M_\mathbf{B}}{DM}$. For the ease of notation, let $\delta := \frac{M_\mathbf{A}M_\mathbf{B}}{DM}$ in the following. Note that by assumption $\gamma \cdot \delta \ge M_\mathbf{A}$ has to hold in order to have all records of $\mathcal{M}_\mathbf{A}$ in the interesting part. By the above considerations, we have $\gamma \cdot \delta \le \frac{M_\mathbf{A}^2 M_\mathbf{B} M_\mathbf{C}}{D^2M^2}$ and hence require $D \le \frac{\sqrt{M_\mathbf{A}M_\mathbf{B}M_\mathbf{C}}}{M} \le \sqrt{M}$. We aim to estimate the number of $N \times N$ matrices with k_1 non-zero entries per column that contain a $\gamma \times \delta$ submatrix with $M_\mathbf{A}$ non-zero entries. To this end, we fix the δ columns in $\mathbf{A}$ and bound the number of possibilities to choose the γ rows of the submatrix. Together with the number of choices to place k_1 non-zero entries in the remaining columns, the number of choices to fill the $\gamma \times \delta$ submatrix, and the possibilities to fill the δ columns such that each column contains k_1 entries, we have

$$\binom{N}{\gamma}\binom{N}{k_1}^{N-\delta} \cdot \sum_{\mathbf{x}\in\mathbb{X}} \prod_{i=1}^{\delta} \binom{\gamma}{x_i}\binom{N}{k_1 - x_i} \tag{6.2}$$

where $\mathbb{X} \subset \mathbb{N}^\delta$ is the set of vectors of dimension δ such that for each vector $\mathbf{x} \in \mathbb{X}$, $\mathbf{x} = (x_1, \ldots, x_\delta)$, it holds $x_i \le k_1$ and $\sum_{i=1}^{\delta} x_i = M_\mathbf{A}$. Using the following observation, the fraction of matrices that contain $M_\mathbf{A}$ non-zero entries in a submatrix can be bounded from above.

Observation 6.3. *Let $x_1, \ldots, x_s$ be natural numbers with $0 \le x_i \le k \le N/2$,*

$1 \le i \le s$, and $\sum_{i=1}^{s} x_i = M \le N$. Then, it holds

$$\frac{\binom{N}{k}^s}{\prod_{i=1}^{s}\binom{N}{k-x_i}} \ge \left(\frac{N-k}{k}\right)^M . \tag{6.3}$$

Proof. By definition of binomial coefficients, we have

$$\frac{\binom{N}{k}}{\binom{N}{k-x_i}} = \frac{N!(k-x_i)!(N-k+x_i)!}{N!k!(N-k)!} = \prod_{j=0}^{x_i-1} \frac{N-k+x_i-j}{k-j} \ge \prod_{j=0}^{x_i-1} \frac{N-k-j}{k-j} .$$

Observe that for $k \le N/2$, for each $0 \le j < k$ it holds that $\frac{N-k-j}{k-j} \ge \frac{N-k}{k}$. Using $\sum_{i=1}^{s} x_i = M$ establishes the lemma. □

Applying Observation 6.3 to (6.2) shows that a fraction of at most

$$\binom{N}{\gamma} \cdot \binom{\gamma\delta}{\sum_{\mathbf{x}\in\mathbb{X}} x_i} \left(\frac{k_1}{N-k_1}\right)^{M_{\mathbf{A}}} = \binom{N}{\gamma} \cdot \binom{\gamma\delta}{M_{\mathbf{A}}} \left(\frac{k_1}{N-k_1}\right)^{M_{\mathbf{A}}}$$

of all possible k_1 regular $N \times N$ matrices contains a $\gamma \times \delta$ submatrix with $M_{\mathbf{A}}$ non-zero entries for a fixed set of δ columns. By estimating the binomial coefficient using Observation 2.1 and the trivial bound of $\binom{n}{k} \le n^k$, and by substituting δ, we obtain an upper bound of

$$N^{\gamma} \left(\frac{e\gamma M_{\mathbf{B}}}{DM}\right)^{M_{\mathbf{A}}} \left(\frac{k_1}{N-k_1}\right)^{M_{\mathbf{A}}} .$$

Furthermore, this term can be bounded from above by estimating $M_{\mathbf{B}} \le M$, $k_1 \le N/2$ and $\gamma \le \frac{M_{\mathbf{A}} M_{\mathbf{C}}}{DM} \le \frac{M_{\mathbf{A}}}{D}$ yielding

$$\alpha(M_{\mathbf{A}}, \gamma) := \min\left\{1, N^{\frac{M_{\mathbf{A}}}{D}} \left(\frac{2ek_1\gamma}{DN}\right)^{M_{\mathbf{A}}}\right\} . \tag{6.4}$$

where we used that the fraction can be at most 1.

In the same manner, the fraction of k_2 regular $N \times N$ matrices with $M_{\mathbf{B}}$ records in a $\frac{M_{\mathbf{A}} M_{\mathbf{B}}}{DM} \times M_{\mathbf{C}}/\gamma$ submatrix with a fixed row set is upper bounded by

$$\beta(M_{\mathbf{B}}, \gamma) := \min\left\{1, N^{\frac{M_{\mathbf{B}}}{D}} \left(\frac{2ek_2 M}{DN\gamma}\right)^{M_{\mathbf{B}}}\right\} \tag{6.5}$$

where we estimate similarly $M_{\mathbf{A}}, M_{\mathbf{C}} \le M$, $M_{\mathbf{C}}/\gamma \le \frac{M_{\mathbf{B}} M_{\mathbf{C}}}{DM} \le \frac{M_{\mathbf{B}}}{D}$ and $k_2 \le N/2$.

Only if both matrices contain a submatrix with sufficiently many non-zero entries, DM elementary products can be created. Similarly to the proof of the lower bound in Chapter 5, we show that this is not always the case, i.e. not for all combinations of matrices $\mathbf{A}$ and $\mathbf{B}$. To this end, we prove that the product of the fractions of matrices $\mathbf{A}$ and $\mathbf{B}$ that contain the required submatrices is strictly smaller than 1 for D according to Equation 6.1. This holds even for the sum over the fractions of "bad" matrices for all choices of $M_{\mathbf{A}}$, $M_{\mathbf{B}}$ and $M_{\mathbf{C}}$ which is expressed in the following inequality.

$$\sum_{M_{\mathbf{A}}=1}^{M} \sum_{M_{\mathbf{B}}=1}^{M} \sum_{M_{\mathbf{C}}=1}^{M} \sum_{\gamma=D}^{\frac{M_{\mathbf{A}}M_{\mathbf{C}}}{DM}} \binom{N}{\gamma} \alpha(M_{\mathbf{A}},\gamma) \cdot \beta(M_{\mathbf{B}},\gamma) < 1 \tag{6.6}$$

We prove (6.6) throughout this section. To this end, we show that every single summand is upper bounded, i.e.

$$\binom{N}{\frac{M_{\mathbf{A}}M_{\mathbf{B}}}{DM}} \alpha(M_{\mathbf{A}},\gamma) \cdot \beta(M_{\mathbf{B}},\gamma) < \frac{1}{M^4} \tag{6.7}$$

for any choice of $M_{\mathbf{A}}, M_{\mathbf{B}}, M_{\mathbf{C}}$, and γ. Thus, it is sufficient to show that

$$f(M_{\mathbf{A}}, M_{\mathbf{B}}, \gamma) := N^{\frac{M_{\mathbf{A}}M_{\mathbf{B}}}{DM}} \alpha(M_{\mathbf{A}},\gamma) \cdot \beta(M_{\mathbf{B}},\gamma) < \frac{1}{M^4} . \tag{6.8}$$

In the following, we distinguish between three cases depending on $\alpha(M_{\mathbf{A}},\gamma)$ and $\beta(M_{\mathbf{B}},\gamma)$.

Case I Assume that

$$\begin{aligned} \alpha(M_{\mathbf{A}},\gamma) &< \left(\frac{1}{N}\right)^{\frac{M_{\mathbf{A}}}{2D}} \\ \beta(M_{\mathbf{B}},\gamma) &< \left(\frac{1}{N}\right)^{\frac{M_{\mathbf{B}}}{2D}} . \end{aligned}$$

The exponent of N in (6.8) can be bounded $\frac{M_{\mathbf{A}}M_{\mathbf{B}}}{DM} \le \frac{M_{\mathbf{A}}}{2D}\frac{M_{\mathbf{B}}}{2D}$ so that bounding

$$\left(N^{\frac{M_{\mathbf{A}}}{2D}} \alpha(M_{\mathbf{A}},\gamma)\right) \cdot \left(N^{\frac{M_{\mathbf{A}}}{2D}} \beta(M_{\mathbf{B}},\gamma)\right) \tag{6.9}$$

yields the desired result. Note that reducing $M_{\mathbf{A}}$ or $M_{\mathbf{B}}$, which appear only as exponents, can only increase (6.9) in the considered case. Hence, (6.9) can be bounded by substituting $M_{\mathbf{A}}$ and $M_{\mathbf{B}}$ with $M' := \min\{M_{\mathbf{A}}, M_{\mathbf{B}}\}$. Substituting $\alpha(M_{\mathbf{A}}, \gamma)$ and $\beta(M_{\mathbf{B}}, \gamma)$ in (6.9) yields an upper bound of

$$N^{\frac{3M'}{D}} \left(\frac{4e^2 k_1 k_2 M}{D^2 N^2} \right)^{M'} .$$

Taking logarithms, it remains to show that

$$3\frac{M'}{D} \log N + M' \log \frac{4e^2 k_1 k_2 M}{D^2 N^2} < -4 \log M$$

which is equivalent to

$$D \log \frac{D^2 N^2}{4e^2 k_1 k_2 M} > 3 \log N + \frac{4D}{M'} \log M .$$

Note that $M' = \min\{M_{\mathbf{A}}, M_{\mathbf{B}}\} \geq D$ is required to create DM elementary products. Furthermore, we assume $M \leq N^{1-\varepsilon}$ so that

$$D \log \frac{D^2 N^2}{4e^2 k_1 k_2 M} > (7 - 4\varepsilon) \log N$$

needs to be shown. This is the case if the logarithm on the left-hand side is positive, i.e.

$$D > 2e \frac{\sqrt{k_1 k_2 M}}{N}$$

and additionally, by Observation 5.8,

$$D > \frac{(7 - 4\varepsilon) \log N}{\log \left(\frac{N^2}{4e^2 k_1 k_2 M} \cdot (7 - 4\varepsilon)^2 \log^2 N \right)} .$$

To fulfil the last inequality,

$$D > \frac{(7 - 4\varepsilon) \log N}{\log \left(\frac{N^2}{4 k_1 k_2 M} \log^2 N \right)}$$

is a sufficient condition. Thus, for D according to inequality (6.1) there are at least two matrices $\mathbf{A}$ and $\mathbf{B}$ such that no submatrices exist that lead to DM elementary products with only M I/Os by a single processor.

Case II Now, assume that

$$\alpha(M_{\mathbf{A}},\gamma) \geq \left(\frac{1}{N}\right)^{\frac{M_{\mathbf{A}}}{2D}} .$$

Then, we obtain by substituting $\alpha(M_{\mathbf{A}},\gamma)$

$$N^{\frac{3M_{\mathbf{A}}}{2D}} \left(\frac{2ek_1\gamma}{DN}\right)^{M_{\mathbf{A}}} \geq 1$$

which is equivalent to

$$\frac{1}{\gamma} \leq \frac{2ek_1}{DN} \cdot N^{\frac{3}{2D}} . \tag{6.10}$$

Since $\alpha(M_{\mathbf{A}},\gamma)$ is bounded from above by 1, we have

$$f(M_{\mathbf{A}},M_{\mathbf{B}},\gamma) \leq N^{\frac{M_{\mathbf{A}}M_{\mathbf{B}}}{D}} \beta(M_{\mathbf{B}},\gamma).$$

Plugging in $\beta(M_{\mathbf{B}},\gamma)$ and γ from (6.10), and using the estimation $M_{\mathbf{A}} \leq M$ yields

$$f(M_{\mathbf{A}},M_{\mathbf{B}},\gamma) \leq N^{\frac{5M_{\mathbf{B}}}{2D}} \left(\frac{4e^2k_1k_2M}{D^2N^2}\right)^{M_{\mathbf{B}}} .$$

Using this upper bound on $f(M_{\mathbf{A}},M_{\mathbf{B}},\gamma)$ for (6.8), we obtain after taking logarithms

$$\frac{5M_{\mathbf{B}}}{2D} \log N + M_{\mathbf{B}} \log \frac{4e^2k_1k_2M}{D^2N^2} < -4\log M$$

and by multiplying by $D/M_{\mathbf{B}}$, we finally get

$$D \log \frac{D^2N^2}{4e^2k_1k_2M} > \frac{5}{2}\log N + \frac{4D}{M_{\mathbf{B}}}\log M$$

which was already shown for D according to (6.1) in Case I.

Case III Finally, assume that

$$\beta(M_{\mathbf{B}},\gamma) \geq \left(\frac{1}{N}\right)^{\frac{M_{\mathbf{B}}}{2D}}$$

which implies

$$\gamma \leq \frac{2ek_2M}{DN} \cdot N^{\frac{3}{2D}} .$$

Using $\beta(M_{\mathbf{B}}, \gamma) \leq 1$ yields

$$f(M_{\mathbf{A}}, M_{\mathbf{B}}, \gamma) \leq N^{\frac{M_{\mathbf{A}}}{D}} \alpha(M_{\mathbf{A}}, \gamma) \,,$$

and by substituting $\alpha(M_{\mathbf{A}}, \gamma)$ and γ, we obtain

$$f(M_{\mathbf{A}}, M_{\mathbf{B}}, \gamma) \leq N^{\frac{5M_{\mathbf{A}}}{2D}} \left(\frac{4e^2 k_1 k_2 M}{D^2 N^2} \right)^{M_{\mathbf{A}}}$$

which yields the same result as Case II.

The Existence of Dense Structures

While the second term in the maximum of D in (6.1) is matched by the tile-based algorithm in Section 6.2.3, the first term reveals some similarities to the upper bound in Section 5.2.4 obtained by the existence of dense subgraphs. Similar to Section 5.2.4, the existence of denser than average parts can be shown by a probabilistic argument for SSM as well. Following Chapter 5, this argument is based on the graph representation of the matrix product of $\mathbf{A}$ and $\mathbf{B}$. However, other than in Chapter 5, the existence of only a single dense part does not directly imply sufficiently many such structures that would allow for a fast computation. While in Section 5.2.4, all elementary products that involve records from the dense subgraph in $\mathbf{A}$ can be evaluated quickly, once the records are in memory, for SSM only dense zones in $\mathbf{A}$ and $\mathbf{B}$ together lead to a fast computation. Thus, it is not clear if the records in a subgraph, for instance from $\mathbf{A}$, that lead to a fast computation involving records within a certain subgraph in $\mathbf{B}$, can be used to obtain elementary products as fast with other records from $\mathbf{B}$. Referring to the three dimensional view depicted in Figure 6.2, only a small cuboid of elementary products can be evaluated.

In this sense, we only show the existence of a single dense part to match the bounds on denser parts in Section 6.3.2. To this end, we will look for sets of edges $S_{\mathbf{A}} \subseteq E_{\mathbf{A}}$, $S_{\mathbf{B}} \subseteq E_{\mathbf{B}}$, $|S_{\mathbf{A}} \cup S_{\mathbf{B}}| \leq M$ such that as many distinct paths (i.e. elementary products) as possible can be generated for only M pairs of nodes from $U \times W$. If, in this setting, for each of the M pairs from $U \times W$ on average D paths exist, by loading $S_{\mathbf{A}} \cup S_{\mathbf{B}}$ records, DM elementary products can be created and added up to output only M records.

Lemma 6.4. *Let $G = (U \uplus V \uplus W, E)$ be a bipartite graph with $|U| = |V| = |W| = N$, where each node in U has degree $k_1 \leq N/2$ and each node in W has degree $k_2 \leq N/2$. For $k_1 k_2 \leq \frac{N^2}{9M} \ln^2 \frac{N}{M}$ and $2M \leq N$, there is a subgraph $G_S = (X \uplus S \uplus Y, E_S)$ for*

$X \subseteq U$, $S \subseteq V$, $Y \subseteq W$ with M edges, $|X| \cdot |Y| \le M$ and average degree

$$D = \min\left\{ \frac{\ln \frac{N}{M}}{3 \ln \frac{N^2}{9Mk_1k_2} \ln^2 \frac{N}{M}}, \frac{\sqrt{M}}{2}, \frac{\min\{k_1, k_2\}}{2} \right\}$$

for nodes from S into each set X and Y.

Proof. We are aiming to find a set $S \subseteq V$, $|S| = M/D$ such that for two sets $X \subseteq U$, $|X| = M/\gamma$, $Y \subseteq W$, $|Y| = \gamma$ drawn uniformly at random, each vertex $v \in S$ has on average at least D neighbours in both X and Y. To this end, we first estimate the probability for an arbitrary vertex $v \in V$ to have at least D neighbours in X. This probability is lower bounded by the probability that X contains exactly D neighbours of v. Hence, $\Pr[|\Gamma(v) \cap X| \ge D]$ is given by a hypergeometric distribution:

$$\Pr[|\Gamma(v) \cap X| \ge D] \ge \binom{M/\gamma}{D} \left(\frac{k_1 - D}{N}\right)^D \left(1 - \frac{k_1}{N - M/\gamma}\right)^{M/\gamma - D} .$$

Assuming that $M/\gamma \ge 2D$, $D \le k_1/2$, and $N \ge 2M/\gamma$, we obtain

$$\Pr[|\Gamma(v) \cap X| \ge D] \ge \binom{M/\gamma}{D} \left(\frac{k_1}{2N}\right)^D \left(1 - \frac{2k_1}{N}\right)^{\frac{M}{2\gamma}} . \tag{6.11}$$

Similarly, for the neighbourhood of v in Y, we get

$$\Pr[|\Gamma(v) \cap Y| \ge D] \ge \binom{\gamma}{D} \left(\frac{k_2}{2N}\right)^D \left(1 - \frac{2k_2}{N}\right)^{\frac{\gamma}{2}} \tag{6.12}$$

assuming $D \le k_2/2$, $\gamma \ge 2D$, and $N \ge 2\gamma$.

Since X and Y are chosen independently, the probability of v having in both sets at least D neighbours is given by the product of (6.11) and (6.12). Note that there must be a set of $S \subseteq V$ of nodes with at least D neighbours in X and Y, if the expected value of such nodes is at least $|S|$. Hence, we want to choose D and γ such that

$$\binom{M/\gamma}{D} \left(\frac{k_1}{2N}\right)^D \left(1 - \frac{2k_1}{N}\right)^{\frac{M}{2\gamma}} \cdot \binom{\gamma}{D} \left(\frac{k_2}{2N}\right)^D \left(1 - \frac{2k_2}{N}\right)^{\frac{\gamma}{2}} \ge \frac{M}{DN}$$

holds. Taking logarithms and estimating binomial coefficients yields

$$D \ln \frac{M}{D^2} + D \ln \frac{k_1k_2}{4N^2} + \frac{M}{2\gamma} \ln\left(1 - \frac{2k_1}{N}\right) + \frac{\gamma}{2} \ln\left(1 - \frac{2k_2}{N}\right) \ge \ln \frac{M}{DN} .$$

By Lemma 5.4, we have $\log(1-x) \geq -2x$ for $x \leq 1/2$ so that the above is implied for $k_1, k_2 \leq N/2$ if

$$D \ln \frac{M k_1 k_2}{4N^2 D^2} - \frac{2k_1 M}{N\gamma} - \frac{2k_2\gamma}{N} \geq \ln \frac{M}{DN}$$

which is equivalent to

$$D \ln \frac{4N^2 D^2}{M k_1 k_2} + \frac{2k_1 M}{N\gamma} + \frac{2k_2\gamma}{N} \leq \ln \frac{DN}{M}\,. \tag{6.13}$$

In the following, we investigate for which D and γ each summand on the left-hand side of (6.13) is bounded from above by $\frac{1}{3}\ln\frac{N}{M}$ (recall that $D \geq 1$).

For the first term, we want to bound D so that $D \ln \frac{2ND}{\sqrt{Mk_1k_2}} \leq \frac{1}{6}\ln\frac{N}{M}$. Similarly to Chapter 5, we apply Lemma 5.5 which yields

$$D \leq \frac{\ln\frac{N}{M}}{6\,\overline{\ln}\,\frac{2N}{6\sqrt{k_1k_2M}}\ln\frac{N}{M}}\,. \tag{6.14}$$

The second term leads to the inequality $\gamma \geq 6\frac{N}{k_1 M \ln N/M} =: a$, while the third term demands $\gamma \leq \frac{N}{6k_2}\ln N/M =: b$. Furthermore, we required $\gamma \geq 2D$ and $M/\gamma \geq 2D$ throughout our calculations, i.e. $2D \leq \gamma \leq M/2D$. Hence, we have to show that there exists a γ fulfilling all these conditions.

We distinguish between the following cases:

- *$2D \leq a$ and $b \leq M/2D$:* We only have to show that $a \leq b$. This is equivalent to $k_1k_2 \leq \frac{N^2}{9M}\ln^2\frac{N}{M}$ which holds by our initial assumption on k_1k_2.
- *$a \leq 2D$ and $M/2D \leq b$:* There is a γ if $D \leq \sqrt{M}/2$ which holds by definition of D.
- *$a \leq 2D$ and $b \leq M/2D$:* We have to prove that $2D \leq b$ holds. For $k_2 \leq N/2$, we have $b \geq \frac{1}{3}\ln N/M$. Any $D \leq \frac{1}{6}\ln N/M$ allows therefore the existence of an appropriate γ.
- *$2D \leq a$ and $M/2D \leq b$:* We want to show $a \leq M/2D$, i.e. $3\frac{N}{k_1 M \ln N/M} \leq M/2D$. This is fulfilled for any $D \geq \frac{1}{3}\frac{k_1}{N}\ln N/M$ which holds again by definition since $k_1 \leq N/2$.

This proves the existence of a γ such that (6.13) holds. Together with (6.14), this finishes the proof. $\square$

6.4 Conclusion

In this chapter, we applied our techniques – which have been used successfully for SPMV and SDM – to derive bounds on the I/O complexities of multiplying two sparse matrices. With the algorithms presented in Section 6.2 we achieve an I/O complexity of

$$\mathcal{O}\left(\min\left\{\frac{k_1k_2N}{P}, \frac{k_1k_2N}{PB}\log_{\mathrm{d}} k_2, \frac{k_1k_2N}{PB}\log_{\mathrm{d}}\frac{k_1N}{B}, \frac{N^2}{PB}\left\lceil\sqrt{\frac{k_1k_2}{M}}\right\rceil\right\}\right).$$

However, even for k_1 and k_2 being upper bounded by N^{ξ} for sufficiently small $\xi < 1$, our only general lower bound on the number of I/Os becomes no stronger than

$$\Omega\left(\min\left\{\max\left\{\frac{k_1N}{P}, \frac{k_2N}{P}\right\}, \frac{k_1k_2N}{PB}\log_{\mathrm{d}}\frac{N}{\min\{k_1,k_2\}B}\right\}\right).$$

Hence, we only have asymptotically matching upper and lower bounds for the case $k_1, k_2 \le \min\{B, N/(2B)\}$.

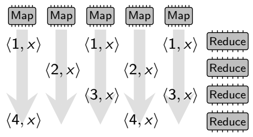
Similar to the lower bound above, a counting argument that is based on comparing the number of programs with the number of tasks can never lead to a lower bound beyond $\Omega((k_1 + k_2)N/P)$: The number of different tasks is bounded above by

$$\binom{N^2}{k_1N}\binom{N^2}{k_2N} \le \left(\frac{eN}{k_1}\right)^{k_1N}\left(\frac{eN}{k_2}\right)^{k_2N}.$$

With $(k_1 + k_2)N/P$ I/Os, any permutation of the $(k_1 + k_2)N$ records can be achieved in a direct manner. Hence, there are $(k_1N + k_2N)!$ configurations reached by programs with $(k_1 + k_2)N/P$ I/Os, which exceeds the number of different SSM tasks. Note that for $k_1, k_2 \ge B$ and $k_1k_2 \le N$, a lower bound for writing the output of $\frac{k_1k_2N}{PB}$ I/Os is already stronger than any lower bound obtained by the classical comparison of the number of different configurations/traces to the number of tasks. Thus, we conjecture that the lower bounds derived by the counting techniques are too weak. Another technique seems to be required instead.

Unfortunately, we could neither derive a lower bound for the general case using density arguments that are similar to Chapter 5. To obtain a lower bound that seems natural - especially because SDM is a special case of SS – we had to restrict ourselves to the class of pseudo rectangular algorithms. However, we believe that a similar lower bound also holds for other algorithms. If the partial results that are created for **C** within a sequence of M/B

I/Os do not lie within a pseudo rectangular submatrix, our bound becomes significantly weaker. Instead, it might be possible to bound the number of output-records in **C**, for which at last D elementary products can be created within a sequence. While the expected number of such records looks promising, the events are highly dependent and a simple tail bound is not sufficient. Hence, it seems that more involved arguments and probabilistic estimations are required to tackle this problem.



7 MapReduce

7.1 Introduction

Since its introduction in 2004 [DG04], the MapReduce framework has become one of the standard approaches in massive distributed and parallel computation. In contrast to its intensive use in practice, theoretical footing is still limited and only little work has been done yet to put MapReduce on a par with the major computational models.

In this chapter, a first step towards a comparison to the PEM model is given. On the one hand, we consider the I/O-efficiency of an algorithm expressed in MapReduce by presenting algorithms for the simulation of the framework. On the other hand, our investigation bounds the complexity that can be "hidden" in the framework of MapReduce in comparison to the PEM model. The main technical contribution is the consideration of the *shuffle step* which is the single communication phase between processors/workers during a MapReduce round. In this step, all information is redistributed among the workers. The insights gained can be helpful when considering the trade-off between a fitted parallel algorithm and the simple expression and communication structure of MapReduce. It also highlights the work that is done automatically by the framework. This chapter is based on results that were published in [GJ11,GJ12].

MapReduce Framework The MapReduce framework [DG04,DG10] can be understood as an interleaved model of parallel and serial computation. It operates in rounds where within one round the user-defined serial functions are executed independently in parallel. Each round consists of the consecutive execution of a map, shuffle and reduce step. The input is a set of $\langle key, value\rangle$ pairs.

A round of the MapReduce framework begins with the parallel execution of independent *map* operations. Each map operation is supplied with a single $\langle key, value\rangle$ pair as input and generates a number of intermediate $\langle key, value\rangle$ pairs. To allow for parallel execution, it is important that map operations are independent from each other and rely on a single input pair only. In the *shuffle* step, the set of all intermediate pairs is redistributed such that lists of pairs with the same key are available for the reduce step. The *reduce* operation for key k is provided with the list of intermediate pairs with key k and generates a new (usually smaller) set of pairs.

The original description in [DG04, DG10], and current implementations, like Hadoop [Whi09], realise this framework by first performing a *split* function to distribute input data to workers. Usually, multiple map and reduce tasks are assigned to a single worker. During the map phase, intermediate pairs are already partitioned according to their keys into sets that will be reduced by the same worker. The intermediate pairs still reside at the worker that performed the map operation and are then pulled by the reduce worker. Sorting the intermediate pairs of one reduce worker by key finalises the shuffle phase. Finally, the reduce operations are executed to complete the round. A common extension of the framework is the introduction of a *combiner* function that is similar in spirit to the reduce function. However, a combine function is already applied during the map execution, as soon as enough intermediate pairs with the same key have been generated.

Typically, a MapReduce program involves several rounds where the output of one round's reduce functions serves as the input of the next round's map functions. Although most examples are simple enough to be solved in one round, there are many tasks that involve several rounds such as computing page rank or prefix sums. In this case, a consideration of the shuffle step becomes most important, especially when map and reduce are I/O-bounded by writing and reading intermediate keys. If map and reduce functions are hard to evaluate and large data sets are reduced in their size by the map function, it is important to find optimised techniques for the evaluation of these functions. However, this shall not be the focus here. Karloff et al. [KSV10] mention that because the shuffle step is a time consuming operation, it is a general aim to reduce the number of MapReduce rounds.

One can see the shuffle step as the transposition of a (sparse) matrix: Considering columns as origin and rows as destination, there is a non-zero entry x_{ij} iff there is a pair $\langle i, x_{ij}\rangle$ emitted by the jth map operation (and hence will be sent to reducer i). Data is first given partitioned by column, and the task of the shuffle step is to reorder non-zero entries row-wise. Since each mapper and reducer is responsible for a certain (known) key, w.l.o.g. we can rename keys to be contiguous and starting with one. Note that there is consensus in current implementations to use a partition operation during the map operation as described above. This can be considered as a first part of the shuffle step.

Related Work Feldman et al. [FMS+10] started a first theoretical comparison of MapReduce and streaming computation. They address the class of symmetric functions (that are invariant under permutation of the input) and restrict communication and space for each worker to be polylogarithmic in the input size N (but mention that results extend to other sublinear functions). In [KSV10], Karloff et al. state a theoretical formulation of the MapReduce model where space restriction and the number of workers is limited by $\mathcal{O}\left(N^{1-\varepsilon}\right)$. Similarly, space restrictions limit the number of records each worker can send or receive. In contrast to other theoretical models, they allow several map and reduce tasks to be run on a single worker. Based on this model, the complexity class $\mathcal{MRC}^i$ is defined to consist of MapReduce algorithms with $\mathcal{O}\left(\log^i N\right)$ rounds. For this model, they present an efficient simulation for a subclass of EREW PRAM algorithms. Goodrich et al. [GSZ11] introduce the parameter M to restrict the number of records sent or received by a machine. Their MapReduce model compares to the BSP model with M-relation, i.e. a restricted message passing degree of M per super-step. The main difference, is that in all the MapReduce models, information cannot reside in the memory of a worker, but the workers have to resent it to itself in order to preserve the data for the next round. A simulation of BSP and CRCW PRAM algorithms is presented based on this model.

The restriction of worker-to-worker communication allows for the number of rounds to be a meaningful performance measure. As noted in [KSV10] and [GSZ11], without restrictions on space/communication there is always a trivial non-parallel one-round algorithm where a single reducer performs a sequential algorithm.

On the experimental side, MapReduce has been applied to multi-core machines with shared memory [RRP+07]. They found several classes of problems that perform well in MapReduce even on a single machine.

Contribution of this Chapter We provide upper and lower bounds on the parallel I/O complexity of the shuffle step. To this end, we revise the algorithms and lower bound techniques from Chapter 4 with a special adaption to the different types of map and reduce functions. With the derived bounds, we can show that current implementations of the MapReduce model as a framework are almost optimal in the sense of worst-case asymptotic parallel I/O complexity. This further yields a simple method to consider the external memory performance of an algorithm expressed in MapReduce.

Following the abstract description of MapReduce [GSZ11, KSV10], the input of each map function is a single $\langle key, value \rangle$ pair. The output of reduce instead can be any finite set of pairs. In terms of I/O complexity, however, it is not important how many pairs are emitted, but rather the size of the input / output matters.

We analyse several different types of map and reduce functions. For map, we first consider an arbitrary order of the emitted intermediate pairs. This is most commonly phrased as the standard shuffle step provided by a framework. Another case is that intermediate pairs are emitted ordered by their key. Moreover, as a last case, we allow evaluations of a map function in parallel by multiple processors. For reduce, we consider the standard implementation which guarantees that a single processor gets data for the reduce operations ordered by intermediate key. Additionally, we consider another type of reduce which is assumed to be associative and parallelisable. This is comparable to the combiner function described before (cf. [DG04]). For the cases where we actually consider the evaluation of map and reduce functions, we assume that input (output) of a single map (reduce) function fits into internal memory. We further assume in these cases that input and output read by a single processor does not exceed the number of intermediate pairs it accesses. Otherwise, the complexity of the task can be dominated by reading the input, or writing the output respectively, which leads to a different character that is strongly influenced by the implementation of map and reduce. For the most general case of MapReduce, we simply assume that intermediate keys have already been generated by the map function, and have to be reordered to be provided as a list to the reduce workers.

We assume similarly to [KSV10] and [GSZ11] that the number of messages sent and received by a processor is restricted. More precisely, for N_M being the number of map operations and N_R the number of reducers, we require that each reducer receives at most $N_M^{1-\gamma}$ intermediate pairs, and each mapper emits at most $N_R^{1-\gamma}$ where γ depends on the type of map operation. However, for the first and the second types of map as described above, any $\gamma > 0$ is sufficient. Only, when considering an associative, parallelisable map function, we require $\gamma \geq \frac{5}{6}$. For N_M and N_R being meaningful parameters, we further assume that each mapper emits at least one intermediate pair and each reducer receives at least one.

7.2 Upper Bounds for the Shuffle Step

For a clearer understanding of the shuffle step and to use the insights developed, we use the analogy of a sparse matrix. Let N_M be the number of distinct input keys, and N_R be the number of distinct intermediate keys (i.e. independent reduce runs). Each pair $\langle i, x_{ij} \rangle$ emitted by map operation j can be considered a triple (i, j, x_{ij}), $j \in [N_M]$, $i \in [N_R]$. Using this notation, one can think of a sparse $N_R \times N_M$ matrix with non-zero entries x_{ij}. This matrix is given in some layout determined by the map function and has to be either reordered into a row-wise ordering, or a layout where rows can be reduced easily. In the following, we consider input keys as column indices and intermediate keys as row indices. The total number of intermediate pairs / non-zero records is denoted by H. Additionally, we have w, the number of records emitted by a reduce function, and v, the size of the input to a map function, where $v, w \leq \min\{M - B, \lceil H/P \rceil\}$ as argued in the introduction. Although we consider a broader class of sparse matrix tasks, the algorithms presented here are similar in spirit to those of Chapter 4. However, they are revised here in order to adapt them to the special cases of map and reduce, we consider. Note that the number of records w emitted by reduce functions does not correspond to the number of SPMV tasks in Chapter 4.

An overview of the algorithmic complexities is given in Table 7.1. Due to space restrictions, terms $\mathcal{O}(\log P)$ for gather tasks and prefix sums are omitted in Table 7.1. Similarly as before, we use $\overline{\log}_b x := \max\{\log_b x, 1\}$. The complexities given in Table 7.1 only differ from the descriptions in the following sections in that we distinguish the special case that a single meta-column is prepared, i.e. the matrix is transposed immediately, and we make use of the observation $\mathcal{O}(\log_d(x/d)) = \mathcal{O}(\log_d x)$. For all our algorithms we assume $H/P \geq B$ similarly to Chapter 4, i.e. there are less processors than blocks in the input and each processor can get a complete block assigned to it.

	Non-parallel reduce	Parallel reduce
Unordered map	$\mathcal{O}\left(\frac{H}{PB}\overline{\log}_{\mathrm{d}} N_R\right)$	$\mathcal{O}\left(\frac{H}{PB}\overline{\log}_{\mathrm{d}} \frac{N_R w}{B}\right)$
Sorted map	$\mathcal{O}\left(\frac{H}{PB}\overline{\log}_{\mathrm{d}} \min\left\{\frac{N_M N_R B}{H}, N_R, N_M\right\}\right)$	$\mathcal{O}\left(\frac{H}{PB}\overline{\log}_{\mathrm{d}} \min\left\{\frac{N_M N_R w}{H}, \frac{N_R w}{B}\right\}\right)$
Parallel map	$\mathcal{O}\left(\frac{H}{PB}\overline{\log}_{\mathrm{d}} \min\left\{\frac{N_M N_R v}{H}, \frac{N_M v}{B}\right\}\right)$	$\mathcal{O}\left(\frac{H}{PB}\overline{\log}_{\mathrm{d}} \frac{N_M N_R v w}{BH}\right)$
Direct shuffling	$\mathcal{O}(H/P)$ (non-uniform)	
Complete merge	$\mathcal{O}\left(\frac{H}{PB}\overline{\log}_{\mathrm{d}} \frac{H}{B}\right)$	

Table 7.1: Overview of the algorithmic complexities $\mathrm{d} = \mathrm{d}(H)$.

7.2.1 Direct Shuffling

Obviously, the shuffle step can be completed by accessing each record once and writing it to its destination. For a non-uniform algorithm, $\mathcal{O}(H/P)$ parallel I/Os are sufficient as described in Chapter 4. To this end, the output can be partitioned into H/P consecutive parts. Since we assume $H/P \geq B$, collisions when writing can be avoided. In contrast, reading the records in order to write them to their destination can be performed concurrently because we consider CREW. We restrict ourselves in this case to a non-uniform algorithm to match the lower bounds in Section 7.3. This shows that our lower bounds are asymptotically tight. Such a direct shuffle approach can be optimal. However, for other cases that are closer to real world parameter settings a more evolved approach is given by the sorting-based algorithms presented in the following.

7.2.2 Map-Dependent Shuffle Part

In this part, we describe for the different types of map functions how to prepare intermediate pairs to be reduced in a next step. To this end, during the first step R meta-runs of non-zero entries from ranges of different columns will be formed. Afterwards, these meta-runs are processed further to obtain the final result. The meta-runs shall be internally ordered row-wise (aka *row major layout*). If intermediate pairs have to be written in sorted order before the reduce operation can be applied, we set $R = \left\lceil \frac{H}{N_R B} \right\rceil$. Otherwise, if the reduce function is associative, it will suffice to set $R = \left\lceil \frac{H}{N_R \max\{w,B\}} \right\rceil$.

Non-Parallel Map, Unordered Intermediate Pairs We first consider the most general (standard) case of MapReduce where we only assume that intermediate pairs from different map execution are written in external memory one after another. The records are ordered by column but within a column no ordering is given. We refer to this as *mixed column layout*. We apply the PEM merge sort to sort records by row index and stop the merging process when the number of runs is less than R. Thus, we get a parallel I/O complexity of $\mathcal{O}\left(\frac{H}{PB} \overline{\log}_{\mathrm{d}} \frac{H}{BR}\right)$ for $\mathrm{d} = \max\left\{2, \min\left\{\frac{M}{B}, \frac{H}{PB}\right\}\right\}$. Note that this ordering can only be exploited algorithmically if a non-parallel reduce functions is applied afterwards so that a row major layout has to be constructed, and only in the sense that intermediate keys are already ordered by input key.

Non-Parallel Map, Sorted Intermediate Pairs Here, we assume that within a column, records are additionally ordered by row index, i.e. intermediate pairs are emitted sorted by their key. This corresponds to the *column major layout*. In the following, we assume $H/N_R \geq B$. Otherwise, the previous algorithm is applied, or simply columns are merged together as described in a later paragraph.

Since columns are ordered internally, each column can serve as a pre-sorted run. Thus, we start the PEM Merge sort with the N_M pre-sorted runs and stop as soon as there are at most R runs left. This leads to an I/O complexity of $\mathcal{O}\left(\frac{H}{PB}\,\overline{\log}_{\mathrm{d}}\,\frac{N_M}{R} + \log\min\{P, B, N_M\}\right)$.

Parallel Map, Sorted Intermediate Pairs In the following, we describe the case with the best possible I/O complexity for the shuffle step when no further restrictions on the distribution of intermediate keys are made. This is the only case where we actually consider the execution of the map function itself. Note that even if a map function emits all intermediate pairs one after another, pairs from a predefined key range can be extracted without inducing more I/Os. This is possible since intermediate pairs are generated in internal memory, and can be deleted immediately, while pairs within the range of interest are kept and written to disk. In a model with considerations of the computational cost, it would be more appropriate to consider a map function which can be parallelised to emit pairs in a predefined intermediate key range.

We first describe the layout of intermediate pairs in external memory that shall be produced. This layout is similar to the best-case layout for the sorting-based algorithm in Chapter 4. In contrast to Chapter 4, where we assume that a best-case layout is given in external memory, we have to construct the layout from the output of the map functions here. Let $m = \min\{M - B, \lceil H/P \rceil\}$. Intermediate pairs shall be written in meta-columns of m/v columns each, which are internally ordered row-wise. Hence, when creating this layout, each processor can keep its memory filled with the m input-records required for each meta-column while writing the intermediate results.

To create this layout efficiently in parallel, the volume of each meta-column has to be determined first by creating and counting intermediate results in parallel without writing them to disk. Since there are vN_M/m meta-columns, it is sufficient to use at most vN_M/m processors for this step. With a parallel prefix sum computation (cf. Section 2.7.2), it can be determined how many, and which range of processor shall be assigned to each meta-column in order to realise an equal load-balancing. The prefix sum computation and the scattering of the processor assignment is possible with $\mathcal{O}(\log P)$ I/Os. After the assignment of processors to meta-columns, each processor reads and keeps input pairs for a whole meta-column in internal memory. Using the map function, intermediate pairs are requested and extracted, and then written to external memory in row-wise order (within a meta-column). If a processor is assigned to multiple meta-columns, it processes each meta-column one after another. Since we assume $H/P \geq B$, multiple processors never have to write to the same block at the same time.

Because we already formed row-wise sorted meta-columns of $\frac{m}{v}$ columns, the number of merge iterations to generate R row-wise sorted meta-runs is reduced. If $\frac{N_M}{m} \leq R$, nothing needs to be done because the number of meta-columns is already less than the desired number of meta-runs. Otherwise, we use the PEM merge sort to reduce the $N_M v/m$ meta-columns into R meta-runs, which induces an I/O complexity of $\mathcal{O}\left(\frac{H}{PB} \overline{\log}_{\mathrm{d}} \frac{N_M v}{\min\{M, H/P\}R} + \log P\right)$.

7.2.3 Reduce-Dependent Shuffle Part

Non-Parallel Reduce Function For the most general case of having a non-parallel reduce function, intermediate keys of the same key have to be provided consecutively to the reduce worker, i.e. a row major layout has to be created. Given the at most $\left\lceil \frac{H}{N_R B} \right\rceil$ meta-runs produced in the previous phase, this can be obtained in a manner similar to the direct algorithm, by moving blocks directly. We describe the current layout in tiles, where one tile consists of the records in one row within a meta-column. The macroscopic ordering of these tiles is currently a column major layout (cf. Figure 7.1). To obtain the desired layout, tiles only need to be rearranged into a row major layout.

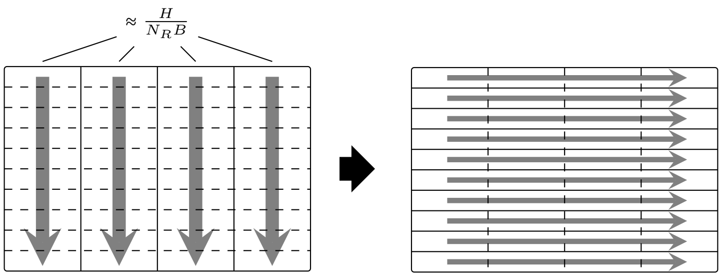


Figure 7.1: Transformation of the tile structure for non-parallel reduce.

Observe that there are $R = \mathcal{O}\left(\frac{H}{N_R B}\right)$ meta-runs with N_R rows each so that there are $\mathcal{O}\left(\frac{H}{B}\right)$ tiles. Each tile covers at most two blocks that contain records from another tile (and need to be accessed separately). However, these are still $\mathcal{O}\left(\frac{H}{PB}\right)$ parallel I/Os to access these blocks. The remaining $\mathcal{O}\left(\frac{H}{B}\right)$ blocks that belong entirely to a tile contribute another $\mathcal{O}\left(\frac{H}{PB}\right)$ I/Os.

To rearrange tiles, we assign records to processors balanced by volume and range-bounded by the tile index (ordered by meta-runs first, and by row within a meta-run). In order to write the output in parallel, the destined positions of each of its assigned records has to be known to the processors. To this end, each processor scans its assigned records, and for each beginning of a new tile, the memory position is stored in a table S (consisting of $\mathcal{O}(H/B)$ entries, one blocks each). Afterwards, using this table, the size of each tile is determined and written to a new table D. With table D a prefix sum computation is started in row major layout (inducing $\mathcal{O}(\log P)$ I/Os) such that D now contains the relative output destination of each row within each meta-run.

In a CRCW model, with the same assignment of records as before, tiles can now be written to their destination to form the output using table D. For CREW, when first creating D, one can ceil the tile sizes to full blocks. The resulting layout will obviously contain blocks that are not entirely filled, but contain empty memory cells. However, using the contraction described in Section 2.7.4, one can extract these empty cells. The whole step to finalise the shuffle step has I/O complexity $\mathcal{O}\left(\frac{H}{PB} + \log P\right)$.

Parallel (Associative) Reduce Function Assuming a parallelisable reduce, each processor can perform multiple reduce functions simultaneously on a subset of records with intermediate key in a certain range. In a final step, the results of these partial reduce executions are then collected and reduced to the final result. By considering addition as the reduce function, this step is completely analogous to the summing process of the elementary products for SpMV which is described in detail in Section 4.2.2.

For an extension to w emitted result vectors, the range of intermediate keys is partitioned into $\lceil N_R P w / H \rceil$ ranges of up to $\lceil H/(Pw) \rceil$ keys. Using the range-bounded load-balancing algorithm from Section 2.7.3, records (still ordered in meta-runs) are assigned to processors such that each processor gets records from at most two pieces of row indices. This can be achieved by using the tuple $(\textit{meta-run index}, \textit{row index})$ as key, and induces $\mathcal{O}\left(\frac{H}{PB} + \log P\right)$ I/Os. If a processor got assigned records that belong to the same reduce function, records can be reduced immediately by the processor.

Afterwards, for each key range, records can be gathered to form the final result of the reduce function. This is possible with $\mathcal{O}\left(\frac{H}{PB} + \log P\right)$ I/Os.

7.2.4 Complete Sorting/Merging

For some choices of parameters, especially for small instances, it can be optimal to simply apply a sorting algorithm to shuffle records row-wise. Using the PEM merge sort, this has I/O complexity $\mathcal{O}\left(\frac{H}{PB} \overline{\log}_{\mathrm{d}} \frac{H}{B}\right)$. Furthermore, if the matrix is given in column major layout, the N_M already sorted columns serve as pre-sorted runs. This results in an I/O complexity of $\mathcal{O}\left(\frac{H}{PB} \overline{\log}_{\mathrm{d}} N_M\right)$.

7.3 Lower Bounds for the Shuffle Step

In order to obtain lower bounds for the shuffle step, we consider the analogy to sparse matrix computations and use the following reduction. A simple task in MapReduce is creating the product of a sparse matrix A with a vector. Assuming that the matrix entries are implicitly given by the map function, the task can be accomplished within one round if the matrix contains at most $N_R^{1-\varepsilon}$ non-zero entries per column and $N_M^{1-\varepsilon}$ per row. To this end, map function j is supplied with input vector record x_j and emits $\langle i, x_j a_{ij} \rangle$. The reduce function simply sums up values of the same intermediate key. Hence, a lower bound for matrix vector multiplication immediately implies a lower bound for the shuffle step. Since reduce can be an arbitrary function, we restrict ourselves to matrix multiplication in a semiring, where the existence of inverse records is not guaranteed. These considerations allow us to revise the lower bounds in [BBF+10] and the extensions to the PEM model in Chapter 2 together with their application in Chapter 4 in order to obtain lower bounds for the shuffle step.

However, we are considering tasks where multiple input- and output-records are associated with each intermediate pair / non-zero entry. More specifically, we have v input and w output vectors. Any intermediate pair – in SpMV an elementary product $a_{jk} x_k^{(i)}$ – can consist of a linear combination $a_{jk} \sum_{i \in I \subseteq [N_M]} x_k^{(i)}$ of the v corresponding vector records, and any output record can be a linear combination of intermediate pairs with corresponding intermediate key. It suffices for a lower bound to consider a simplified version of this task. In the simplified task, each intermediate pair is a copy of one of the v input-records, and it is required for the computation of precisely one output record. The ith coordinate of the lth output vector is then the sum of all intermediate pairs that were associated with vector l. Hence, in contrast to Chapter 4, the task considered here corresponds to multiplying each non-zero entry a_{ij} with only one of the v vector records $x_j^{(1)}, \dots, x_j^{(v)}$. The assignment which vector is used for which non-zero entry is part of the input. Each non-zero entry has, apart from its position in the matrix and its value, two further variables assigned to it, defining origin of the input vector record and destination of the elementary product. We refer to this task as the **combined matrix vector product** of a given matrix, a set of v input vectors, the number of output vectors w and a given assignment of non-zero entries to input/output vectors. The following theorem is proven throughout the next sections.

Theorem 7.1. *Given parameters B, $M \geq 3B$ and $P \leq \frac{H}{B}$. Creating the combined matrix vector product for a sparse $N_R \times N_M$ matrix with H non-zero entries for $H/N_R \leq N_M^{1-\varepsilon}$ and $H/N_M \leq N_R^{1-\varepsilon}$ for $\varepsilon > 0$ from $v \leq H/N_M$ input vectors to $w \leq H/N_R$ output vectors has (parallel) I/O complexity*

- $\Omega\left(\min\left\{\frac{H}{P}, \frac{H}{PB} \log_{\mathrm{d}} \frac{N_R w}{B}\right\}\right)$ *if the matrix is in mixed column layout*
- $\Omega\left(\min\left\{\frac{H}{P}, \frac{H}{PB} \log_{\mathrm{d}} \min\left\{\frac{N_M N_R w}{H}, \frac{N_R w}{B}\right\}\right\}\right)$ *if given in column major layout*
- *and* $\Omega\left(\min\left\{\frac{H}{P}, \frac{H}{PB} \log_{\mathrm{d}} \frac{N_M N_R v w}{H \min\{M, H/P\}}\right\}\right)$ *for the best-case layout if $H/N_R \leq \sqrt[6]{N_M}$ and $H/N_M \leq \sqrt[6]{N_R}$*

where $\mathrm{d} = \max\{2, \min\{M/B, H/(PB)\}\}$.

These lower bounds already match the algorithmic complexities for parallel reduce in Section 7.2. Moreover, a lower bound for creating a matrix in row major layout from v vectors can be obtained (cf. parallel map & non-parallel reduce).

Lemma 7.2. *Given parameters B, $M \geq 3B$ and $P \leq \frac{H}{B}$. Creating a sparse $N_R \times N_M$ matrix with H non-zero entries in row major layout from v vectors $x^{(1)}, \dots, x^{(v)}$ such that for all non-zero entries holds $a_{ij} = x_j^{(k)}$ for some k has (parallel) I/O complexity*

$$\Omega\left(\min\left\{\frac{H}{P}, \frac{H}{PB} \log_{\mathrm{d}} \min\left\{\frac{N_M N_R v}{H}, \frac{N_M v}{B}\right\}\right\}\right)$$

for $H/N_R \leq N_M^{1-\varepsilon}$ and $H/N_M \leq N_R^{1-\varepsilon}$, $\varepsilon > 0$.

Theorem 7.1 and Lemma 7.2 both hold not only in the worst-case, but for a fraction of the possible sparse matrices exponentially close to one. Hence, for distributions over the matrix conformations (position of the non-zero entries), even if not uniform but somehow biased, the lower bounds still hold on average if a constant fraction of the space of matrix conformations has constant probability. Similarly, the bounds hold on average for distributions where a constant fraction of the non-zero entries is drawn with constant probability from a constant fraction of the possible position.

The following technical lemma is required for the proofs in this section.

Lemma 7.3. *Assume* $\log 3H \geq \frac{7}{2}B \log \min\left\{\frac{M}{B}, \frac{2H}{PB}\right\}$, $H \geq \max\{N_1, N_2\} \geq 2$, $H/N_2 \leq N_1^{1-\varepsilon}$, *then*

(i) $H \leq N_2^{1/\varepsilon}$

(ii) $N_2 \geq 2^8$ *implies* $B \leq \frac{1}{e\varepsilon} N_2^{3/8}$.

(iii) $N_2 \geq 2^8$ *and* $H/N_2 \leq N_1^{1/6}$ *implies* $\min\left\{\frac{M}{B}, \frac{2H}{BP}\right\} \leq N_2^{\frac{3}{7B}}$.

Proof. Combining $H/N_2 \leq N_1^{1-\varepsilon}$ with $H \geq N_1$ yields $N_1 \leq N_2 N_1^{1-\varepsilon}$, i.e. $N_2 \leq N_1^{1/\varepsilon}$. Substituting N_1 in $H \leq N_2 N_1^{1-\varepsilon}$ results in (i).

For (ii), we have $B \leq \frac{2}{7} \log 3H \leq \frac{1}{e} \log H$ since $H \geq N_2 \geq 2^8$ so that we can use $3 \cdot 2^8 < 2^{10}$ and note that $\frac{5}{4} \cdot \frac{2}{7} \leq \frac{1}{e}$. Using (i), we get $B \leq \frac{1}{e} \log N_2^{1/\varepsilon} \leq \frac{1}{e\varepsilon} \log N_2$. Finally, we simply use the additional observation $\log x \leq x^{3/8}$ for $x \geq 2^8$.

The last results is derived from the main assumption, which can be rewritten as $3H \geq \min\left\{\frac{M}{B}, \frac{2H}{BP}\right\}^{7B/2}$. Again, using $3H \leq H^{5/4}$ for $H \geq N_2 \geq 2^8$, we get $H \geq \min\left\{\frac{M}{B}, \frac{2H}{BP}\right\}^{14B/5}$. H in turn is bounded from above by $N_2^{6/5}$ so that we have $\min\left\{\frac{M}{B}, \frac{H}{BP}\right\} \leq (N_2^{6/5})^{5/(14B)} \leq N_2^{3/(7B)}$. □

7.3.1 Best-Case to Row Major Layout with Multiple Input Pairs

To begin with, we consider a task that is related to the matrix vector product but seems somewhat simpler. In [BBF+10], a copy task is described, where a matrix in column major layout is created from a vector such that each non-zero entry in row i is a copy of the ith vector record. This allows for a time forward analysis as described in Section 2.2.1, similar to the lower bound for permuting in [AV88] and in Theorem 2.7. In [BBF+10], the copy task is only used as a preliminary task, and is reduced to the matrix vector product to obtain a lower bound. Here, we can actually use the task itself to state a lower bound. Observe that the copy task is equivalent to the creation of a sparse matrix in row major layout where each non-zero entry in column j is a copy of the jth vector record. This corresponds to the special case of MapReduce where a map function simply copies its input value H/N_M times with random intermediate keys which then have to be ordered by intermediate key. Thus, a lower bound for this task states a lower bound for the shuffle step with parallel map functions but non-parallel reduce. We extend this task further to v multiple input vectors, such that in the created matrix each non-zero entry in column j is a copy of of the jth vector record of a specified one of the v vector.

Abstract Configurations In the following, we bound the number of I/Os required for a family of programs for this copy task such that every matrix conformation can be created by a program. To this end, we consider the change of configurations as defined in Chapter 1. Analogously to [BBF$^+$10] and Section 2.2.1, we abstract from the actual configuration in that the abstract configuration at time t refers to the concatenation of non-empty memory cells of external and internal memory between the tth I/O and the $t+1$th I/O. Recall that in this abstraction the ordering and multiplicity of similar records in a block or internal memory is ignored. As usually, we consider programs that are normalised according to the description of normalised programs in Section 2, i.e. every intermediate record is a predecessor of the output.

Similar to [BBF$^+$10], and like in Section 4.3, we abstract furthermore from the actual value of a record to the indices describing its positioning in the task. Instead of the full information of a record, we consider only the set of *(column index, origin vector index)* tuples of the records in a block or in internal memory. Hence, blocks and internal memory are considered subsets of $\{(1,1),\ldots,(N_M,v)\}$ of size at most B and M, respectively. This abstracts especially from the ordering and multiplicity of (distinct) records that have the same tuple of indices.

By Lemma 2.5, a family of normalised programs with ℓ parallel I/Os and a fixed computation trace can lead to

$$\prod_{l=1}^{\ell} 3^P \prod_{p=1}^{P} \binom{M_{p,l}+B}{B} 2^B \cdot 2(\lceil n/B \rceil + P\ell)$$

distinct abstract configurations where n is the input size and $M_{p,l}$ is the number of distinct records in internal memory of processor p before the lth parallel I/O. For a non-trivial bound $\ell > \frac{n}{PB}$ (obtained by reading the input), we obtain a number of distinct abstract configurations after ℓ I/Os of

$$\prod_{l=1}^{\ell} 3^P \prod_{p=1}^{P} \binom{M_{p,l}+B}{B} 2^B \cdot 4P\ell \le \left(3^P \binom{\min\{MP, H\}+PB}{PB} 2^{PB} 4P\ell \right)^{\ell} \quad (7.1)$$

where the last inequality stems from bounding the product of binomial coefficients, similar to Section 4.3.1, by using that the maximal number of records concurrently in internal memories in an abstract conformation is at most H.

Since a normalised program for the copy task does not involve any computation operations such as summing or multiplication, there is only a single (empty) computation trace over the entire set of programs for the copy task. For fixed input sizes N_M, N_R, and H, the initial abstract configuration is unique over all programs. The final abstract configuration in contrast depends on the conformation of the matrix that is generated.

Number of Abstract Matrix Conformations For a family of programs being able to produce all conformations, ℓ needs to be large enough so that (7.1) is at least as large as the number of abstract configurations representing all conformations. There are $\binom{N_M}{H/N_R}^{N_R}$ different conformations of $N_R \times N_M$ matrices with H/N_R non-zero entries per row. For the ease of notation, we assume for a lower bound that H is an integer multiple of N_R. Furthermore, each of the non-zero entries can stem from one of the v input vectors. However, since we consider abstract configurations and ignore the ordering and multiplicity of records within a block, the number of final abstract configurations is less. For an abstract configuration, it is not clear whether a tuple of indices in a block stems from one or multiple rows, neither from which of them. Analogously to [BBF⁺10] and as already considered in Section 4.3, the following three cases have to be distinguished. If $H/N_R = B$, each block corresponds to exactly one row so that each abstract configuration describes only a single conformation. In case $H/N_R > B$, a block contains entries from at most two rows. Hence, a column index in the abstract description of a block can originate either from the first, the second or from both rows. For $H/N_R < B$, a block contains entries from $\lceil BN_R/H \rceil$ different rows. A row can however only contain indices from at most two blocks. Thus, there are $\binom{2B}{H/N_R}^{N_R}$ orderings of records in external memory with the same abstract description.

Theorem 7.4. *Given the block size B, internal memory size $M \geq 3B$ and the number of processors $P \leq \frac{H}{B}$. Creating a sparse $N_R \times N_M$ matrix with H non-zero entries in row major layout from $v \leq H/N_M$ vectors such that $a_{ij} = x_j^{(k)}$ for a $k \in \{1, \ldots, v\}$ for each non-zero entry such that $H/N_R \leq N_M^{1-\varepsilon}$ and $H/N_M \leq N_R^{1-\varepsilon}$ for (constant) $\varepsilon > 0$ with $N_M \geq 9^{1/\varepsilon}$ has (parallel) I/O complexity*

$$\ell \geq \min\left\{\frac{\varepsilon^2}{5}\frac{H}{P}, \frac{H}{7PB}\log_{\min\{\frac{M}{B}, \frac{2H}{PB}\}} \min\left\{\frac{N_M N_R v}{3H}, \frac{N_M v}{eB}\right\}\right\}.$$

Proof. If a family of programs with ℓ I/Os is able to create all conformations of $N_R \times N_M$ matrices in row major layout with H non-zero entries, then

$$\left(3^P \binom{\min\{MP, H\} + PB}{PB} 2^{PB} 4P\ell\right)^\ell \geq \binom{N_R}{H/N_M}^{N_M} v^H / \tau_R \tag{7.2}$$

with

$$\tau_R \leq \begin{cases} 3^H & \text{if } B < H/N_R \\ 1 & \text{if } B = H/N_R \\ (2eBN_R/H)^H & \text{if } B > H/N_R \end{cases}$$

has to hold. For $\ell \geq \frac{1}{5}\frac{H}{P}$, the claim is already proven, so we can assume in the following $\ell < \frac{1}{4}\frac{H}{P}$. Similar, if $N_M \leq 2^8$, then the logarithm in Theorem 7.4 is smaller than 7 so that the theorem is proven with a scanning bound of $\frac{H}{PB}$ which is required for reading the input. Hence, also assume $N_M > 2^8$. Taking logarithms and estimating binomial coefficients in (7.2) yields

$$\ell P \left(B + \log 3H + B \log e \frac{\min\{M, H/P\} + B}{B}\right) \geq H \log \frac{N_M N_R}{H} + H \log v - \log \tau_R \,.$$

After substituting τ and isolating ℓ, we obtain

$$\ell \geq \frac{H}{P} \frac{\log \min\left\{\frac{N_M N_R v}{3H}, \frac{N_M v}{2eB}\right\}}{\log 3H + B \log\left(2e(\min\left\{\frac{M}{B}, \frac{H}{PB}\right\} + 1)\right)}$$

and using the assumptions $M \geq 3B$ and $H/(PB) \geq 1$, we get

$$\ell \geq \frac{H}{P} \frac{\log \min\left\{\frac{N_M N_R v}{3H}, \frac{N_M v}{2eB}\right\}}{\log 3H + \frac{7}{2} B \log \min\left\{\frac{M}{B}, \frac{2H}{PB}\right\}}.$$

where we used that $2e(x+1) \leq (2x)^{7/2}$ for $x \geq 1$.
Case 1: For $\log 3H \leq \frac{7}{2} B \log \min\left\{\frac{M}{B}, \frac{2H}{PB}\right\}$, we have

$$\ell \geq \frac{H}{7PB} \log_{\min\left\{\frac{M}{B}, \frac{2H}{PB}\right\}} \min\left\{\frac{N_M N_R v}{3H}, \frac{N_M v}{2eB}\right\}.$$

Case 2: If $\log 3H \geq \frac{7}{2} B \log \min\left\{\frac{M}{B}, \frac{2H}{PB}\right\}$, we can use $3H \leq H^{5/4}$ for $H \geq N_M \geq 2^8$, and with Lemma 7.3.i, we obtain

$$\ell \geq \frac{H}{P} \frac{\log \min\left\{\frac{N_M N_R v}{3H}, \frac{N_M v}{2eB}\right\}}{\frac{5}{2} \log N_M^{1/\varepsilon}}.$$

Using Lemma 7.3.ii with $N_1 = N_R$ and $N_2 = N_M$, and ignoring v, i.e. $v = 1$, we have

$$\ell \geq \frac{H}{P} \frac{\log \min\left\{\frac{1}{3}N_M^{\varepsilon}, \frac{\varepsilon}{2}N_M^{5/8}\right\}}{\frac{5}{2}\log N_M^{1/\varepsilon}} .$$

For $N_M \geq 3^{2/\varepsilon}$, we obtain

$$\ell \geq \frac{H}{P} \frac{\log \min\left\{N_M^{\varepsilon/2}, \frac{\varepsilon}{2}N_M^{5/8}\right\}}{\frac{5}{2}\log N_M^{1/\varepsilon}} \geq \frac{H}{P} \frac{\log N_M^{\varepsilon/2}}{\frac{5}{2}\log N_M^{1/\varepsilon}} \geq \frac{\varepsilon^2}{5}\frac{H}{P}$$

because $\frac{\varepsilon}{2}N_M^{5/8} \geq N_M^{\varepsilon/2}$ holds for $N_M \geq 2^8$ for all $0 < \varepsilon < 1$: Using $N_M = 2^8$, we have $\varepsilon 2^4 \geq 2^{4\varepsilon}$ which holds obviously for $0 < \varepsilon \leq 1$. Larger N_M can only increase the absolute difference between the terms. □

7.3.2 Mixed Column Layout with Multiple Output Pairs

This case is similar to the lower bound for column major layout in Section 4.3.1. The main differences are that in a mixed column layout, the records within a column are not ordered whereas for column major layout, records are ordered row-wise within a column. Additionally, we consider the combined matrix vector product instead of SPMV. Following [BBF+10] and as described in Section 4.3.1, it suffices for a matrix vector product with the matrix in column major layout to consider the multiplication of a matrix with the all-ones-vector, i.e. the task of creating row sums from the matrix. In the combined matrix vector product, each elementary product is only used for the calculations of one of the w output vectors. Hence, we consider w independent tasks of building subsets of row sums. This task can be seen as a time-inverse variant of the copy task described in Section 7.3.1. Instead of spreading copies of input-records, the scattered matrix records of the same row have to be collected and summed up to w subset sums. Thus, we apply the time-backwards analysis from Lemma 2.6 to identify the number of initial abstract configurations that can reach a single abstract output configuration.

Abstract Configurations Again, we consider normalised programs by their abstract configurations according to Section 2.2.2, i.e. we consider blocks and internal memory as sets of records, we ignore empty blocks, and additionally, records that do not belong to the final output and are not accessed after the considered configuration, are ignored. Furthermore, instead of actual records only the set tuples of *row* indices and *destinations* vector are considered. Note that a sum can only consist of records from the same row having the same destination vector in a normalised program. Hence, internal memory and each block states a subset of $\{(1,1),\dots,(N_R,w)\}$ of size up to M and B, respectively.

Description of Abstract Programs Consider the final configuration after ℓ I/Os. Since all records that do not belong to the output are ignored, the final abstract configuration is unique for all programs that compute the matrix vector product for fixed N_R. In contrast, the initial configuration depends on the conformation of the matrix. Hence, by Lemma 2.6 there are at most

$$\prod_{l=1}^{\ell} 3^P \prod_{p=1}^{P} \binom{M_{p,l}+B}{B} 2^B 2(\lceil n/B \rceil + P\ell) \le \left(3^P \binom{\min\{MP,H\}+PB}{PB} 2^{PB} 4P\ell\right)^{\ell} \tag{7.3}$$

initial abstract configuration that lead to the same abstract output configuration for fixed computation trace, where $M_{p,l}$ is the number of distinct records in internal memory of processor p after the lth I/O. We bounded again the product of binomial coefficients, using that there exist at most H (non-empty) records in an abstract configuration at a time, and assumed $P\ell > \lceil \frac{H}{B} \rceil$ in (7.3) justified by a lower bound for reading the input.

In the considered task, no multiplication operation is required. Furthermore, sum operations and their following deletion operations are not visible in our view of abstract configurations. The computation trace does hence not influence the sequence of abstract configuration.

Number of Abstract Matrix Conformations To obtain a lower bound on the I/O complexity of the mixed column layout, (7.3) is lower bounded by the number of different matrix conformations in mixed column layout expressed by an abstract configuration. We consider matrices with exactly H/N_M records per column, hence, we assume that H is an integer multiple of N_M Think of drawing the non-zero entries for each column one after another. For the number of non-zero entries per column $H/N_M \le N_R/2$, there are at least $(N_R/2)$ possibilities to draw the position of a non-zero entry. Furthermore, each record can be involved in the computation of one of the w output vectors. In total, there are at least $(N_R/2)^H w^H$ different matrix conformations. However, in an abstract configuration, the ordering of records within a block gets hidden. Additionally, if a block contains records from several columns, in an abstract configuration it is not clear from which column(s) a tuple may stem. The number of different conformation that correspond to the same abstract conformation can be bounded from above by B^H since each record can be one of the at most B row indices in the set describing its block (if it is the ith record in the layout, its block is the $\lceil i/B \rceil$th).

Theorem 7.5. *Given block size B, internal memory $M \geq 3B$ and the number of processors $P \leq \frac{H}{B}$. Creating the combined matrix vector product for a sparse $N_R \times N_M$ matrix in mixed column layout with H non-zero entries for $w \leq H/N_R$ output vectors and $H/N_R \leq N_M^{1-\varepsilon}$, $H/N_M \leq N_R^{1-\varepsilon}$, and $N_R \geq 1/\varepsilon^{8/3}$ for $\varepsilon > 0$ has (parallel) I/O complexity*

$$\ell \geq \min\left\{\frac{\varepsilon}{10}\frac{H}{P}, \frac{H}{7PB}\log_{\min\{\frac{M}{B},\frac{2H}{PB}\}}\frac{N_R w}{2B}\right\}.$$

Proof. A lower bound on the minimal number of I/Os ℓ required for a family of programs that create the matrix vector product for $N_R \times N_M$ matrices with H entries in mixed column layout is given from (7.3) by

$$\left(3^P\binom{\min\{MP, H\} + PB}{PB}2^{PB}4P\ell\right)^{\ell} \geq \left(\frac{N_R}{2B}\right)^H w^H. \tag{7.4}$$

With similar arguments as in the proof of Theorem 7.4, we can assume $\ell < \frac{1}{4}H/P$, and $N_R \geq 2^8$. Otherwise the theorem holds trivially. Taking logarithms and estimating binomial coefficients yields

$$\ell P\left(B + \log 3H + B\log e\frac{\min\{M, H/P\} + B}{B}\right) \geq H\log\frac{N_R w}{2B} + H\log w.$$

Isolating ℓ, we can obtain

$$\ell \geq \frac{H}{P}\frac{\log\frac{N_R w}{2B}}{\log 3H + \frac{7}{2}B\log\min\left\{\frac{M}{B}, \frac{2H}{PB}\right\}}.$$

where we used again $M \geq 3B$ and $H/(PB) \geq 1$.

Case 1: For $\log 3H \leq \frac{7}{2}B\log\min\left\{\frac{M}{B}, \frac{2H}{PB}\right\}$, the lower bound matches the sorting algorithm:

$$\ell \geq \frac{H}{7PB}\log_{\min\{\frac{M}{B},\frac{2H}{PB}\}}\frac{N_R w}{2B}$$

Case 2: For $\log 3H > \frac{7}{2}B\log\min\left\{\frac{M}{B}, \frac{2H}{PB}\right\}$, we get

$$\ell \geq \frac{H}{P}\frac{\log\frac{N_R w}{2B}}{2\log 3H}.$$

Using Lemma 7.3.i, and $3H \geq H^{5/4}$ for $H \geq 2^8$, we have

$$\ell \geq \frac{H}{P} \frac{\log \frac{N_R w}{2B}}{\frac{5}{2} \log N_R^{1/\varepsilon}}$$

and with Lemma 7.3.ii, we get

$$\ell \geq \frac{H}{P} \frac{\log \varepsilon N_R^{5/8} w}{\frac{5}{2} \log N_R^{1/\varepsilon}} \geq \frac{\varepsilon}{10} \frac{H}{P}$$

for $N_R \geq 1/\varepsilon^{8/3}$ which is matched by the direct algorithm. □

7.3.3 Column Major Layout with Multiple Output Pairs

For column major layout, the number of different abstract matrix conformations corresponds to the number of abstract conformation described in Section 7.3.1 but with N_M and N_R exchanged.

If a family of programs with ℓ I/Os is able to create the matrix vector product with each $N_R \times N_M$ matrix with H non-zero entries to obtain $w \leq H/N_R$ vectors of row sum subsets, then

$$\left(3^P \binom{\min\{MP, H\} + PB}{PB} 2^{PB} 4P\ell\right)^{\ell} \geq \binom{N_M}{H/N_R}^{N_R} w^H / \tau_M \qquad (7.5)$$

with

$$\tau_M \leq \begin{cases} 3^H & \text{if } B < H/N_M \\ 1 & \text{if } B = H/N_M \\ (2eBN_M/H)^H & \text{if } B > H/N_M \end{cases}$$

has to hold. This yields the following theorem.

Theorem 7.6. *Given the block size B, internal memory $M \geq 3B$ and the number of processors $P \leq \frac{H}{B}$. Creating the combined matrix vector product for a sparse $N_R \times N_M$ matrix in column major layout with H non-zero entries for w output vectors and $H/N_R \leq N_M^{1-\varepsilon}$, $H/N_M \leq N_R^{1-\varepsilon}$, and $N_M \geq 9^{1/\varepsilon}$ for $\varepsilon > 0$ has (parallel) I/O complexity*

$$\ell \geq \min\left\{\frac{\varepsilon^2}{5} \frac{H}{P}, \frac{H}{7PB} \log_{\min\{\frac{M}{B}, \frac{2H}{PB}\}} \min\left\{\frac{N_M N_R w}{3H}, \frac{N_R w}{eB}\right\}\right\}.$$

7.3.4 Best-Case Layout with Multiple Input and Output Pairs

For the best-case layout, the algorithm is allowed to choose the layout of the matrix as described in Chapter 1. Recall that this makes the task of building row sums become trivial by setting the layout of the matrix to row major layout. Hence, we follow the movement and copying of input vector records as well, like in Section 4.3.2.

In contrast to Section 4.3.2, we investigate bounds for the combined matrix product, where we consider both, multiple input and multiple output vectors. Recall that any intermediate pair stems from exactly one input-record, and it is required for the computation of only a single output record. For each of the H non-zero entries in our matrix there is not only a choice of its position but also the choice from which of the v records it stems from and which of the w records is its destination. This results in a total number of $\binom{N_M N_R}{H} v^H w^H$ different tasks. We assume that $v \geq H/N_M$ and $w \geq H/N_R$ so that all input- and output-records can be useful.

The task of creating this type of matrix vector product can be seen as first spreading the input-records to create elementary products $a_{ij}x_j^{(k_{ij})}$ – where k_{ij} denotes the index vector that is required for the elementary product involving a_{ij} – and then collecting these elementary products to form the output vectors. To describe these two main tasks, we distinguish between matrix entries $a_{i,j}$, elementary products $a_{ij}x_j^{(k_{ij})}$ and partial sums $\sum_{j\in\mathcal{S}} a_{ij}x_j^{(k_{ij})}$ which we trace as described in Section 7.3.3, and input vector records which will be followed as described below. To the first group (matrix entries, elementary products, partial sums), we also refer as row records and we call the row index their index.

Abstraction As before, we consider an abstraction of the current configuration according to Section 2.2. As usually, the considered programs are normalised as described in Chapter 2. Additionally, we normalise programs in that a multiplication is performed as soon as both records are in internal memory. Since a non-zero entry $a_{i,j}$ is used for exactly one multiplication in the task, it can be replaced by the product. Hence, this normalisation requires no further space in internal memory, and the number of I/Os does not change.

Similar to Section 4.3.2, we distinguish between two abstract configurations for each configuration, the abstract row configuration and the abstract column configuration. The abstract column configuration is analogous to Section 7.3.1: Only records that are a copy of one of the $N_M v$ input vector records are considered, all other records are ignored. The records are further reduced to the tuple describing their column index and the index of the vector they stem from. Hence, the internal memory of each processor and each block are considered subsets of $\{(1,1),\dots,(N_M,v)\}$. In the abstract row configuration only partial sums $\sum_{j\in\mathcal{S}} a_{ij}x_j^{(k_{ij})}$ are considered, analogously to Section 7.3.2. Abstracting from records to row and destination vector indices, internal memories and blocks are considered subsets of $\{(1,1),\dots,(N_R,w)\}$.

Description of Programs Again, the final abstract row configuration over all programs that create the matrix vector product is unique for fixed N_R. Hence, we can apply our time-backwards analysis from Section 2.2.2 which we also used in Section 7.3.3 to describe the I/O trace of the abstract row configurations for programs with ℓ I/Os. To describe the I/O trace of abstract input configurations, we use the time-forward analysis from Section 2.2.1 which we used in Section 7.3.1.

It remains to bound the number of computation traces. Since we abstract from multiplicity and actual records to indices, copy and sum operations do not change the trace of abstract configurations. There is only one operation that performs an interaction, and that is multiplying an input-record with a non-zero entry which creates a row record. For each non-zero entry a_{ij}, exactly one multiplication with a $x_j^{(k_{ij})}$ is performed during the whole execution of the program. Hence, if in our abstraction, a row record with index i was created by multiplication from an input variable with index j, this fixes a_{ij} to be non-zero. We normalised programs such that an elementary product is created immediately when both records appear in internal memory together for the first time. This can only happen after an input. Thus, for each input there are at most B new records in internal memory and some $M_{p,l}$ records that are already in internal memory. Hence, there are at most $BM_{p,l}$ possibilities where a multiplication can be done for each processor after each I/O. In total, we get $Y = \sum_{l=1}^{\ell}\sum_{p=1}^{P} BM_{p,l} \le \ell B \cdot \min\{PM, H\}$ possibilities where a multiplication can be performed. After each multiplication, the operands can be deleted, yielding an additional factor of 4^H. Together with the abstract I/O traces for input and row records, this gives a unique description of the abstract matrix conformation.

Theorem 7.7. *Given the block size B, internal memory $M \ge 3B$, and the number of processors $P \le \frac{H}{B}$. Creating the combined matrix vector product for a sparse $N_R \times N_M$ matrix in best-case layout with H non-zero entries for $v \le H/N_M$ input and $w \le H/N_R$ output vectors, with $H/N_R \le N_M^{1/6}$ and $H/N_M \le N_R^{1/6}$ has (parallel)*

I/O complexity

$$\ell \geq \min\left\{\frac{1}{24}\frac{H}{P}, \frac{H}{14PB}\log_{\min\{\frac{M}{B},\frac{2H}{PB}\}}\frac{N_M N_R v w}{4H\min\{M, H/P\}}\right\}.$$

Proof. With the above observations, for a family of programs with ℓ I/Os that create the matrix vector product for any $N_R \times N_M$ matrix with H non-zero entries where the layout of the matrix can be chosen by the program it has to hold

$$4^H\binom{Y}{H}\cdot\left[\left(3^P\binom{\min\{MP,H\}+PB}{PB}2^{PB}4P\ell\right)^{\ell}\right]^2 \geq \binom{N_R}{H/N_M}^{N_M} v^H w^H.$$

Like in the proofs before, we argue that $\ell < \frac{H}{4P}$ and $N_R \geq 2^8$, otherwise the claim holds trivially. Estimating binomial coefficients and taking logarithms, we have

$$2\ell P\left(\log 3H + \frac{7}{2}B\log\min\left\{\frac{M}{B},\frac{2H}{PB}\right\}\right) \geq H\left(\log\frac{N_M N_R v w/4H}{e\ell B\min\{PM, H\}/H}\right)$$

where we followed the calculations given for the other layouts. Reordering terms, we obtain

$$\ell \geq \frac{H}{P}\frac{\log\frac{N_M N_R v w}{4e\ell B\min\{PM,H\}}}{\log 3H + \frac{7}{2}B\log \mathrm{d}}$$

with $\mathrm{d} = \min\{M/B, 2H/(PB)\}$. By Lemma 4.8, $x \geq \frac{\log_b(s/x)}{t}$ implies $x \geq \frac{\log_b(s\cdot t)}{2t}$. For $x = \ell$, $s = \frac{N_M N_R}{4eB\min\{PM,H\}}$ and $t = \frac{P}{H}\left(\log 3H + \frac{7}{2}B\log\mathrm{d}\right)$ this results in

$$\ell \geq \frac{H}{2P}\frac{\log\frac{N_M N_R v w P\left(\log 3H + \frac{7}{2}B\log\mathrm{d}\right)}{4eBH\min\{PM,H\}}}{\log 3H + \frac{7}{2}B\log\mathrm{d}}.$$

Case 1: For $\log 3H \leq \frac{7}{2}B\log\mathrm{d}$, we get a lower bound of

$$\ell \geq \frac{H}{14P}\frac{\log\frac{N_M N_R v w}{4H\min\{M,H/P\}}}{B\log\mathrm{d}}.$$

Case 2: For $\log 3H > \frac{7}{2}B\log\mathrm{d}$, we get

$$\ell \geq \frac{H}{4P}\frac{\log\frac{N_M N_R v w\log 3H}{4eBH\min\{M,H/P\}}}{\log 3H} \geq \frac{H}{4P}\frac{\log\frac{N_M N_R 7\log d}{8eHdB}}{\frac{5}{4}\log H}$$

using $H \geq 2^8$. Assuming $H/N_M \leq N_R^{\frac{1}{6}}$ and using Lemma 7.3.i with $\varepsilon = \frac{5}{6}$, we get

$$\ell \geq \frac{H}{5P} \frac{\log \frac{7N_R^{5/6}}{8eBd}}{\log N_R^{6/5}}$$

for $N_R \geq 9$. Using Lemma 7.3.ii and 7.3.iii, with $N_1 = N_M$ and $N_2 = N_R$, we finally get

$$\ell \geq \frac{H}{6P} \frac{\log \frac{7 \cdot \frac{5}{6} N_R^{5/6}}{8N_R^{3/8} N_R^{3/(7B)}}}{\log N_R} > \frac{H}{6P} \frac{\log \frac{2}{3} N_R^{1/3}}{\log N_R} \geq \frac{H}{24P}$$

for $B \geq 4$, by using $N_R \geq 2^8 > (\frac{3}{2})^{12}$ so that $\frac{5}{6} N_R^{1/3} \geq N_R^{1/3-1/12} \geq N_R^{1/4}$. □

7.3.5 Transposing Bound

Another method is presented in [AV88] to obtain lower bounds on the I/O complexity of dense matrix transposition. This potential-based approach was extended to the PEM in Section 2.3. The bound can also be applied to sparse matrix transposition if the matrix is given in column major layout.

Theorem 7.8. *For $B > 4$, the transposition of a sparse $N_R \times N_M$ matrix with H non-zero entries has worst-case parallel I/O complexity*

$$\Omega\left(\frac{H}{PB} \log_{\mathrm{d}} \min\left\{B, N_M, N_R, \frac{H}{B}\right\}\right)$$

where $\mathrm{d} = \max\left\{2, \min\left\{\frac{M}{B}, \frac{H}{PB}\right\}\right\}$.

Proof. Note that this task does not require any computation operations. Now consider the potential defined by the togetherness ratings in Section 2.3. A normalised program has obviously a final potential of $\Phi(\ell) = H \log B$. The initial potential in contrast has to be considered for each matrix separately. Note that the task is symmetric, i.e. transposing from column to row major layout has the same I/O complexity as the other way around. W.l.o.g. let $N_M \geq N_R$ in the following.

For $H/N_M \geq B$, any matrix that is row-wise H/N_M-regular and column-wise H/N_R-regular has an initial potential of $\Phi(0) \leq H \log 4$: Observe that in this case each input block intersects with an output block in at most four records: Each block in the initial layout covers at most two columns, and each block in the output layout covers at most two rows. By Lemma 2.8, the increase of the potential during one parallel I/O is bounded above by

$PB \log 2e + PB \log \frac{\min\{M,H/P\}}{B}$. This yields a lower bound for the number of I/Os of

$$\ell \geq \frac{H \log B - 2H}{PB \log 2e + PB \log \frac{\min\{M,H/P\}}{B}}$$

so that $\ell = \Omega\left(\frac{H}{PB} \log_d B\right)$ holds.

For $H/N_M < B$, we consider the following matrix. All non-zero entry a_{ij} are located at coordinates where $(i-1)H/N_R+1 \leq j \leq iH/N_R \mod N_M$ holds. This yields a sparse matrix that is regular in both dimensions (cf. Figure 7.2). Any matrix with such a conformation stored in column or row major layout corresponds directly to a dense $H/N_M \times N_M$ matrix in column major layout, row major respectively. A block of the output covers hence $\lceil B/N_M \rceil$ rows. The blocks of the initial layout cover at most $\lceil BN_M/H \rceil$ columns. Conform with [AV88] an initial potential of $\Phi(0) \leq H \log \max\left\{1, \left\lceil \frac{BN_M}{H} \right\rceil, \left\lceil \frac{B}{N_M} \right\rceil, \left\lceil \frac{B^2}{H} \right\rceil\right\}$ is obtained. Hence, we get a lower bound of

$$\Omega\left(\frac{H}{PB} \log_{\mathrm{d}} \min\left\{B, \frac{H}{N_M}, N_M, \frac{H}{B}\right\}\right) . \tag{7.6}$$

□

Combining the Bounds

A combination of the previous bound and Theorem 7.1 matches the algorithmic complexities given in Section 7.2 for non-parallel reduce. For the scenario $N_M > B > H/N_M$, the minimum breaks down to the term H/N_M. Combining the results from Theorem 7.6 with the above bound, we get

$$\Omega\left(\frac{H}{PB}\left(\log_{\mathrm{d}} \frac{H}{N_M} + \log_{\mathrm{d}} \frac{N_M N_R}{H}\right)\right)$$

which is bound from below by $\Omega\left(\frac{H}{PB} \log_{\mathrm{d}} N_R\right)$. Similar observations hold for $N_R > B > H/N_R$. Considering the other cases for the minimum in (7.6), we get a lower bound of

$$\Omega\left(\min\left\{\frac{H}{P}, \frac{H}{PB} \log_{\mathrm{d}} \min\left\{\frac{N_M N_R B}{H}, N_M, N_R, \frac{H}{B}\right\}\right\}\right)$$

I/Os for matrices in column major layout.

Given a matrix in mixed column layout, we can apply (7.6) as well since column major is a special case of the mixed column layout. Hence, with similar considerations, we obtain a lower bound of

$$\Omega\left(\min\left\{\frac{H}{P}, \frac{H}{PB} \log_{\mathrm{d}} \min\left\{N_R, \frac{H}{B}\right\}\right\}\right)$$

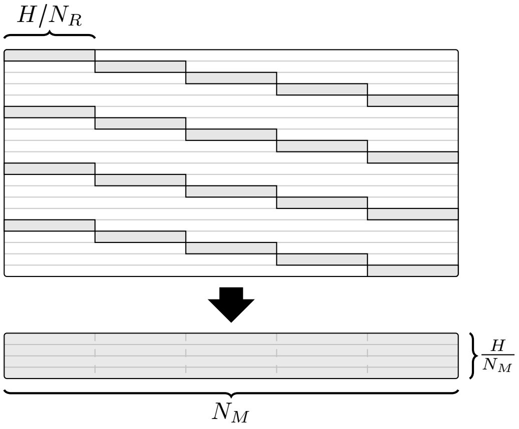


Figure 7.2: The matrix conformation to obtain a lower bound for transposing in the case $H/N_M \leq B$.

I/Os for matrices in mixed column layout.

Lower Bound for Scatter/Gather and Prefix Sums

To cover all the algorithmic complexities, it remains to consider complexities induced by the scatter and gather tasks and the prefix sum computation that are required for the exclusive write policy, and the load-balancing. A matching lower bound for some of the tasks can be obtained from Section 2.4. From there, we derive a lower bound of $\log \min\{H/B, N_M\}$ for matrices in mixed column and column major layout, and $\log(N_M/B)$ for the best-case layout. Since we have $H \leq N_M^{1/\varepsilon}$ from Lemma 7.3.i using $N_2 = N_M$, we can estimate

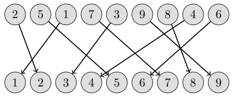
$$\log N_M \geq \varepsilon \log H \geq \varepsilon \log H/B \geq \varepsilon \log P$$

which yields asymptotically matching complexities for mixed column and column major layout. For the best-case layout, we have to assume $N_M \geq B^{1+\varepsilon}$, implying $\log(N_M/B) \geq \varepsilon \log N_M$, which then yields a matching lower bound as before. Otherwise, at most an additive term $\mathcal{O}(\log P)$ differs between our upper and lower bounds.

7.4 Conclusion

We determined the parallel worst-case I/O complexity of the shuffle step for most meaningful parameter settings. All our upper and lower bounds for the considered variants of map and reduce functions match up to constant factors. Although worst-case complexities are considered, most of the lower bounds hold with probability exponentially close to 1 over uniformly drawn shuffle tasks. We considered several types of map and reduce operations, depending on the ordering in which intermediate pairs are emitted and the ability to parallelise the map and reduce operations. All our results hold especially for the case where the internal memory of the processors is never exceeded but (block) communication is required.

Our results show that for parameters that are comparable to real world settings, sorting in parallel is optimal for the shuffle step. This is met by current implementations of the MapReduce framework where the shuffle step consists of several sorting steps, instead of directly sending each record to its destination. In practice one can observe that a merge sort usually does not perform well, but rather a distribution sort does. The partition step and the network communication in current implementations to realise the shuffle step can be seen as iterations of a distribution sort. Still, our bounds suggest a slightly better performance when in knowledge of the block size. If block and memory size are unknown to the algorithm, which corresponds to the so called cache-oblivious model, it is known that already permuting ($N_M = N_R = H$) cannot be performed optimally. Sorting instead can be achieved optimally, but only if $M \geq B^2$ [BF03]. However, when assuming that the naïve algorithm with $\mathcal{O}\left(\frac{H}{P}\right)$ I/Os is not optimal, and $M \geq B^2$, all the considered variants have the same complexity and reduce to sorting all intermediate pairs in parallel.



8 Permutations

We consider the problem of permuting N records in external memory where each record i in the input contains its future position $\pi(i)$ as one of its values. This involves the exchange of records from the $\lceil N/B \rceil$ input to the $\lceil N/B \rceil$ output blocks. In this chapter, we introduce a new notion to describe permutations of records in a block structure. The **block graph** relates input and output blocks that share a record by an edge in a bipartite graph (cf. Figure 8.1). This abstracts from the ordering of records within a block. However, with one scan of the output blocks, every block can be permuted internally. In case that each output block differs from the input block at the corresponding position, i.e. it needs to be written at some point in time, the required block internal permutation can be generated before the final output.

Definition 8.1 (Block graph)**.** *For a permutation $\pi : [N] \to [N]$, we define the **block graph** of π to be the bipartite graph $G_\pi = (V_{in} \cup V_{out}, E)$ with $V_{in} = \{l_1, \dots, l_{N/B}\}$ and $V_{out} = \{r_1, \dots, r_{N/B}\}$ where for each l_i, r_j there is an edge if and only if the jth output block contains a record from the ith input block. Nodes in V_{in} are referred to as **input nodes**, those of V_{out} as **output nodes**.*

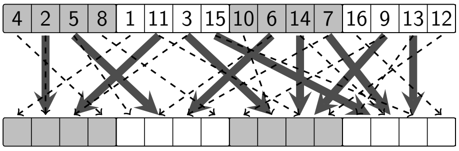


Figure 8.1: Extracting the block graph from a permutation

For simplicity, we assume that B divides N, and w.l.o.g. input and output start at the border of a block. Hence, both input and output consist of exactly N/B entirely filled blocks. By construction, a block graph has maximum degree B, and when allowing parallel edges, its degree is exactly B. Some examples of permutations and their block graphs are given in Figure 8.2.

Many permutations that reveal a very structured block graph – with many similar subgraphs, or with small connected components – imply a simple I/O complexity. This includes dense matrix transposition and the more general bit-matrix-multiply/complement (BMMC) permutations. The class of BMMC permutations maps source addresses to their target by an affine transformation of the bit vectors. This is realised by multiplying the address bit vector with a nonsingular matrix and adding a complement vector. These permutations were investigated in the I/O-model by Cormen et al. in [CW93] who gave a rather complicated though asymptotically optimal algorithm to perform BMMC permutations with $\mathcal{O}\left(\frac{N}{B}\log_{M/B}\operatorname{rank}\gamma\right)$ I/Os, where γ is the lower left $\log(N/B) \times \log B$ submatrix of the bit matrix as described later on. In their algorithm, once the BMMC permutation is identified, the bit matrix is factored into simpler bit matrix permutations (e.g. using Gaussian elimination). These can be of four kinds. For one of them, which appears at most once as a factor, $\mathcal{O}\left(\frac{N}{B}\log_{M/B}\operatorname{rank}\gamma\right)$ I/Os are shown to be sufficient. From the other kinds, there can be at most $\mathcal{O}\left(\log_{M/B}\operatorname{rank}\gamma\right)$, each of which can be realised with $\mathcal{O}(N/B)$ I/Os.

The remainder of this chapter is twofold. We consider the block graph of BMMC matrices, and show that BMMC permutations induce small connected components. Our way to describe permutations by the block graph leads to a more intuitive understanding of why BMMC permutations form an easy subclass of permutations in the I/O- and the PEM model. Building upon this, we present a new simple and parallel algorithm for BMMC

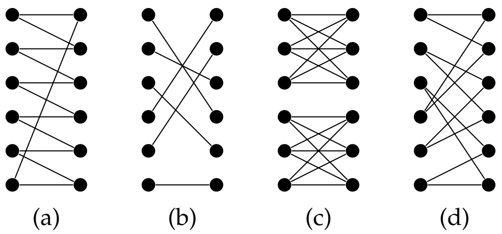


Figure 8.2: Block graphs for (a) cyclic shifting records by $i \in \{1, \ldots, B\}$ positions, (b) moving blocks and permuting each block internally, (c) transposing a 3×6 matrix for block size $B = 3$, (d) $\pi(i) = 2i \mod 12$ for $B = 2$.

permutations.

Secondly, we investigate graph expansion as an indicator for hard permutations. Graphs with good expansion properties reveal a good connectivity which might oppose the simplicity of small connected components in the context of permuting. In contrast to this assumption, we can exclude this property up to some extend by showing that there are simple permutations that describe an expander with nearly asymptotically maximal expansion properties, only a factor $\mathcal{O}(\log B)$ away from maximal expansion. Hence, we conclude that neither connectivity nor expansion lead to an understanding of what makes permuting in the I/O-model nearly as difficult as sorting.

8.1 BMMC-Permutations

Considering bit operations in this section, computations are made over the field $\mathbb{F}_2$ over $\{0,1\}$ with addition $\oplus$ (XOR), and multiplication $\wedge$ (AND). To perform calculations on bit strings, the vector space $\mathbb{F}_2^n$ is used. In addressing a record by a vector, the first entries correspond to the least significant bits. For convenience reasons, we shall assume that N, M and B are exact powers of 2. Like in Chapter 1, we use the notation $[N]$ to refer to the set $\{1,\ldots,N\}$.

Definition 8.2. *Let $\pi : [N] \to [N]$ be a permutation on N records and $n = \log N$. Further let $\boldsymbol{i}$ denote the bit vector representing $i \in [N]$ with least significant positions first.*

A permutation is called BMMC permutation *if there is $\boldsymbol{A} \in \mathbb{F}_2^n \times \mathbb{F}_2^n$ and $\boldsymbol{c} \in \mathbb{F}_2^n$ such that for all $i \in [N]$, $\pi(i) = j$ is equivalent to $\boldsymbol{A}\boldsymbol{i} \oplus \boldsymbol{c} = \boldsymbol{j}$.*

Note that since π is a permutation and hence bijective, $\boldsymbol{A}$ needs to have full rank. Moreover, every bit matrix with full rank describes a permutation. The following observation was exposed in [Cor93] where the membership for bit-permute/complement permutations, a subclass of BMMC permutations, is demonstrated. This yields a lower bound on the I/O complexity of performing a BMMC permutations, also in the PEM model, by Theorem 2.9

Observation 8.3. *Dense matrix transposition is a BMMC permutation. Given an $N \times M$ matrix in column major layout with N and M being exact powers of 2. The bit matrix corresponding to the transposition of the matrix is a cyclic shift of the $\log N + \log M$ bits by $\log N$ positions to the left (bit vectors are again in little endian).*

Proof. The first $\log N$ positions of the bit vector determine the row index of a record within its column. The remaining $\log M$ positions describe the index of the column. Hence, a swap of the first $\log N$ positions with the last $\log M$, which corresponds to a cyclic left-shift by $\log N$ bits, yields a row major layout. □

In the remainder of this section, we show that a BMMC permutation induces a block graph with small connected components of similar structure and describe a simple algorithm to detect and perform BMMC permutations.

Theorem 8.4. *The block graph of a BMMC permutation with bit matrix $\boldsymbol{A}$ and complement vector $\boldsymbol{c}$ consists of disconnected subgraphs of size $2^{1+\operatorname{rank}\gamma}$ with γ being the lower left $n-b \times b$ submatrix of A.*

Proof. We first define the $n-b\times n$ matrix $\boldsymbol{P}$ which removes the first (least significant) b bits of any bit vector and leaves all the other bits as they are (removing the first b rows of the identity matrix $\boldsymbol{I}$ with dimension n yields $\boldsymbol{P}$). Thus for any $i \in [N]$, $\boldsymbol{Pi}$ yields the bit vector indexing the block of x. Similarly, $\boldsymbol{PAi}$ yields the index of the target/output block of i. Now let $U = \operatorname{kern}\boldsymbol{P} + \operatorname{kern}\boldsymbol{PA}$ and $\mathbb{F}_2^n/U$ be the quotient space of $\mathbb{F}_2^n$ modulo U with equivalence classes (cosets) $[\boldsymbol{x}] = \boldsymbol{x} + U$ for any $\boldsymbol{x} \in \mathbb{F}_2^n$.

For any two records $i, j \in [N]$ that are located in the same input block it follows by definition of $\boldsymbol{P}$ that $\boldsymbol{Pi} = \boldsymbol{Pj}$, and hence, $\boldsymbol{P}(\boldsymbol{i}-\boldsymbol{j}) = \boldsymbol{0}$. This means that $\boldsymbol{i}-\boldsymbol{j} \in \operatorname{kern}\boldsymbol{P}$ and there is $\boldsymbol{u} \in \operatorname{kern}\boldsymbol{P} \subseteq U$ such that $\boldsymbol{i} = \boldsymbol{j}+\boldsymbol{u}$. Consequently, $[\boldsymbol{i}] = [\boldsymbol{j}+\boldsymbol{u}] = [\boldsymbol{j}]$. In other words, all records within an input block belong to the same coset defined by $\mathbb{F}_2^n/U$. By the same argument, $\boldsymbol{P}(\boldsymbol{Ai} \oplus \boldsymbol{c}) = \boldsymbol{P}(\boldsymbol{Aj} \oplus \boldsymbol{c})$ holds if $\boldsymbol{i}-\boldsymbol{j} \in \operatorname{kern}\boldsymbol{PA} \subseteq U$ (observe that the complement vector $\boldsymbol{c}$ is irrelevant), and thus, implies $[\boldsymbol{i}] = [\boldsymbol{j}]$. Hence, any two records that share a target block belong to the same coset as well.

As argued above, an input block contains only records from the same coset. Similarly, an output block consists of records within a single coset. Since an edge in the block graph describes a record that belongs to the adjacent input and output blocks, the records of both blocks all belong to the same coset. Hence, any connected component of the block graph consists of blocks whose records are in the same coset. Since each coset has (finite) cardinality $|U|$, we conclude that all the connected components of the block graph are of the same size, namely $2\cdot|U|/B$ (recall that in the block graph each record is involved in 2 blocks).

U is a subspace of $\mathbb{F}_2^n$, so we know $|U| = 2^{\dim U}$ (the number of linear combinations for a vector basis, and the maximal number of linearly independent vectors respectively). By definition, $\dim U = \dim(\operatorname{kern}\boldsymbol{P} + \operatorname{kern}\boldsymbol{PA}) = \dim\operatorname{kern}\boldsymbol{P} + \dim\operatorname{kern}\boldsymbol{PA} - \dim(\operatorname{kern}\boldsymbol{P}\cap\operatorname{kern}\boldsymbol{PA})$ where $\operatorname{kern}\boldsymbol{P}\cap\operatorname{kern}\boldsymbol{PA}$ can be expressed as $\operatorname{kern}\boldsymbol{PA}\,|_{\operatorname{kern}\boldsymbol{P}}$ in which $\boldsymbol{PA}\,|_{\operatorname{kern}\boldsymbol{P}}$ is the restriction of $\boldsymbol{PA}$ to $\operatorname{kern}\boldsymbol{P}$. By the rank theorem, $\dim\operatorname{kern}\boldsymbol{PA}\,|_{\operatorname{kern}\boldsymbol{P}} = \dim\operatorname{kern}\boldsymbol{P} - \operatorname{rank}\boldsymbol{PA}\,|_{\operatorname{kern}\boldsymbol{P}}$ so that $\dim U = \dim\operatorname{kern}\boldsymbol{PA} + \operatorname{rank}\boldsymbol{PA}\,|_{\operatorname{kern}\boldsymbol{P}}$. Since $\operatorname{kern}\boldsymbol{P}$

is spanned by the unit vectors $\boldsymbol{e_1}, \ldots, \boldsymbol{e_b}$, $\boldsymbol{PA} \mid_{\operatorname{kern} \boldsymbol{P}}$ consists exactly of the first b columns of $\boldsymbol{A}$. The transformation $\boldsymbol{P}$ annihilates the first b rows, so that $\operatorname{rank} \boldsymbol{PA} \mid_{\operatorname{kern} \boldsymbol{P}} = \operatorname{rank} \gamma$ where γ is the lower left $n - b \times b$ submatrix of $\boldsymbol{A}$. Finally, because we have $\dim \operatorname{kern} \boldsymbol{PA} = b$, the dimension of U is $b + \operatorname{rank} \gamma$ and every connected component of the block graph has size at most $2^{1+\operatorname{rank} \gamma} \leq 2B$. □

A lower bound using the potential from [AV88] (which we extended to the PEM model in Section 2.3) is presented in [CW93]. This lower bound matches the observation that each block contains records from $2^{\operatorname{rank} \gamma}$ different target blocks. Hence, the initial potential is $\Phi(0) = B/2^{\operatorname{rank} \gamma}$, yielding the following lower bound.

Lemma 8.5. *The number of I/Os required to perform a BMMC permutation with bit matrix $\boldsymbol{A}$ is bounded below by $\Omega\left(\frac{N}{B} \log_{\mathrm{d}} 2^{\operatorname{rank} \gamma}\right)$ for $\mathrm{d} = \max\left\{2, \min\left\{\frac{M}{B}, \frac{N}{PB}\right\}\right\}$.*

8.1.1 A Simple Algorithm for BMMC Permutations

An algorithm with optimal I/O complexity for the single processor case is given in [CW93]. However, as described above, this algorithm operates with rather complex matrix multiplications arguments to yield a factorisation of the permutation matrix $\boldsymbol{A}$. Having shown that a BMMC permutation induces small connected components, one can apply a permuting algorithm to each of the connected components.

For any number of processors $P \leq \frac{N}{B}$, the following algorithm can be used once the connected components are known. A description on how to identify the connected components, or to refuse the input if not a BMMC permutation, follows below. Given the $\frac{N}{2^{\operatorname{rank} \gamma}}$ groups of blocks that describe a connected component, each group can be considered a separate permutation. The P processors are then assign evenly among the $\frac{N}{2^{\operatorname{rank} \gamma}}$ permutation tasks. Since each tasks consists of $2^{\operatorname{rank} \gamma}$ blocks, a reordering of the records within each group takes $\mathcal{O}\left(\frac{N}{PB} \frac{\operatorname{rank} \gamma}{\log \mathrm{d}}\right)$ I/Os. Note that tasks can consist of non-consecutive blocks. The output blocks of each permutation task are then written to the block position specified by the overall permutation to form the output.

Now, we discuss how the connected components are identified. Recall that U is the combination of $\operatorname{kern} \boldsymbol{P}$ and $\operatorname{kern} \boldsymbol{PA}$. While $\operatorname{kern} \boldsymbol{P}$ is trivially given, $\operatorname{kern} \boldsymbol{PA}$ can be obtained by identifying all blocks that contain records that are to be permuted into the same output block. I.e. it suffices to identify a single connected component to determine U. In the following, we consider records that are permuted into the *first* output block. Using U, for each block it can be determined whether it has the lowest block id of the coset, i.e. in the connected component. This yields one block for each connected component (and the set U to address all other blocks of the connected component) so that the permutation tasks can be assigned among the processors.

The connected component including the first input block can be determined with $\mathcal{O}\left(\frac{N}{PB} + \log \frac{N}{B}\right)$ (parallel) I/Os as follows. First, the $\frac{N}{B}$ blocks are assigned evenly among the processors such that at most $\lceil \frac{N}{PB} \rceil$ records are assigned to each processor. Scanning the assigned blocks, it is determined whether a record in the block is to be permuted into the first output block. A block that is – in terms of the block graph – connected to the first output block is marked (marking blocks can be achieved by reserving an additional $\frac{N}{B}$ records in external memory to contain the marking). Recall that the ids of marked blocks are sufficient to describe U. Since $\operatorname{kern} \boldsymbol{PA} \leq B$, and each record can hold a number in $[N]$, the ids of input blocks within the first coset can be saved in one block. Let the set of these block ids be U'. In order to identify one block for each coset, U is provided to each processor. To this end, a parallel prefix sum computation is invoked after reading (and possibly marking) all blocks to assign each marked block with a unique, consecutive index (with $\mathcal{O}(\log P)$ I/Os). Afterwards, the block ids are written into a table which takes $\mathcal{O}\left(\frac{N}{PB}\right)$ I/Os so that U is provided. With the same number of I/Os, the table can be read in parallel to provide each processor with U'.

In a second step, for each block in the assigned area of a processor, it is checked whether it has lowest id within its connected component. This can certainly be done by exoring the block ids with each of the ids in U'. Each block with lowest block id in its connected component is marked, and, with a prefix sum computation, a consecutive indexing is generated. Then, a table containing one block id for each coset is built. Using this table, the $\frac{N}{2^{\operatorname{rank}\gamma}}$ groups can be assigned to processors. Before starting the permutation tasks, each processor checks if all blocks within a group permute to the same output blocks. Otherwise, the permutation is not a BMMC permutation and is refused. Finally note that the permutations successfully performed with this algorithm are not necessarily BMMC permutation. However, the class of BMMC permutations is contained.

8.1.2 Extension to Other Fields

The proof of Theorem 8.4 can also be extended to permutations defined by a matrix $\mathbf{A}$ on indices from a vector space over any galois field $GF(Z)$ for N and B being exact powers of Z. For this, the projection matrix $\boldsymbol{P}$ is defined such that the first $\log_Z B$ positions are removed. The cardinality of the cosets is then $|U| = Z^{\dim U} = Z^{\operatorname{rank}\gamma_Z} \leq B$ where γ_Z is the lower left $\log_Z \frac{N}{B} \times \log_Z B$ submatrix of $\boldsymbol{A}$. Thus, the connected components still have size no more than $2B$.

8.2 Expander Block Graphs

Considering the block graph of permutations, especially in the context of BMMC permutations, leads to the intuition that low connectivity and a structured neighbourhood function implies an "easy" permutation. For the class of BMMC permutations studied before, "easy" refers to an I/O complexity of $\mathcal{O}\left(\frac{N}{B}\log_{M/B} B\right)$ – which for $M \geq B^{1+\varepsilon}$ (tall cache) reduces to scanning time – opposing a lower bound for general permuting of $\min\left\{N, \frac{N}{B}\log_{M/B} N\right\}$. This leads to the conjecture that high connectivity might be an indicator for hard permutation tasks. However, we disprove this conjecture by presenting a construction of a class of expander block graphs which describe easy permutations. The proposed expanders guarantee a vertex expansion of $\mathcal{O}(B)$ for sets of size up to $\mathcal{O}\left(\frac{N}{B^2}\log B\right)$. This is almost optimal in the sense that the maximum size of sets which still expand can only be improved by a factor $\log B$. Our construction is inspired by the construction of the zig-zag product in [RVW00].

8.2.1 Definitions

Throughout this section, we consider discrete random variables on finite sets, i.e. a variable $X : S \to [0,1]$ with finite sample set S. The **probability mass function** (pmf) of X is $p_X(s) = \Pr[X = s]$ for $s \in S$.

Definition 8.6 (Support). *For a random variable X, the **support** of X (**Supp**(X)), is the support the pmf $p_X : S \to [0,1]$ underlying X, i.e. the set $\{s \in S \mid p_X(s) > 0\}$.*

Definition 8.7. *For a function $f : A \to B$ and a random variable X on A with distribution $p_X : A \to [0,1]$, the random variable $Y = f(X)$ on B has probability distribution $p_Y : B \to [0,1]$ with $p_Y(b) = \sum\limits_{a \in A \mid f(a) = b} p_X(a)$ for all $b \in B$.*

Definition 8.8 (ε-close). *Two random variables X and Y are called ε-**close** if for the probability distributions p_X and p_Y holds $\sum_{s \in Supp(X)} |p_X(s) - p_Y(s)|^+ \leq \varepsilon$ where*

$$|x|^+ = \begin{cases} x & \text{if } x > 0, \\ 0 & \text{otw.} \end{cases}$$

Note that this is equivalent to l_1 distance $\leq 2\varepsilon$, and is hence symmetric.

Definition 8.9 (Min-entropy). *The* ***min-entropy*** *of a finite random variable X with range S is given by $H_\infty(X) = -\log \max_{s\in S} \Pr[X = s]$.*

Definition 8.10 (Sources). *A* ***k-source*** *is a random variable with min-entropy at least k. A random variable is called* ***(k,ε)-source*** *if there is a k-**source** ε-close to it. A* ***flat source*** *is a random variable whose possible values (with positive probability) all have the same probability.*

A well-known relation between flat and arbitrary k-sources is given in the following lemma.

Lemma 8.11. *Every distribution of a k-source over a finite set is a convex combination of distributions of flat k-sources.*

Proof. A random variable X on a finite set A is a k-source if $0 \le p_X(a) \le 2^{-k}$ for all $a \in A$. We can describe a probability distribution as a vector $p \in [0,1]^{|A|}$ where $p_X(a_i) = p_i$ for some ordering $a_1, \dots, a_{|A|}$ of the elements in A. This defines the space of probability distributions of k-sources as the intersection of the hypercube $[0, 2^{-k}]^{|A|}$ and the hyperplane defined by $\sum_{i=1}^{|S|} p_i = 1$. Because this intersection yields a convex polytope, any point of the polytope can be written as a convex combination of the vertices of the polytope. Observe that the hyperplane cannot intersect the hypercube at a pure edge of its boundary, i.e. a point with one $0 > p_i > 2^{-k}$ and all other $p_i = 2^{-k}$ or $p_i = 0$. Hence, the vertices of the polytope are exactly the vertices of the hypercube that lie on the hyperplane, i.e. $p_i = 2^{-k}$ for 2^k i's and $p_i = 0$ for the rest. □

For the sake of readability, we denote the set of n-bit strings by (n) in the following.

Definition 8.12 (Definition 10.2 from [HLW06]). *A function $E : (n) \times (d) \to (m)$ is a $(k_{max}, a, \varepsilon)$-**conductor** if for any $k \le k_{max}$ and any k-source X over (n), the random variable $E(X, U_d)$ is a $(k+a, \varepsilon)$-source (where U_d is the random variable described by the uniform distribution over (d)).*

*E is called (k_{max}, ε)-**lossless conductor** if it is a $(k_{max}, d, \varepsilon)$-conductor. A pair of functions $\langle E, C\rangle : (n) \times (d) \to (m) \times (b)$, where $n + d = m + b$ is a* ***(lossless) permutation conductor*** *if E is a (lossless) conductor and $\langle E, C\rangle$ is a permutation over $(n + d)$.*

Definition 8.13 (Modification of Definition 10.3 from [HLW06]). *Let $G = (U \uplus V, E)$ be a bipartite graph where $|U| = N$, $|V| = M$, and all vertices in U have degree D. The graph G is a (K_{max}, δ)-**expander** if every set of $K \le K_{max}$ left vertices has at least δK neighbours.*

8.2.2 An Non-Trivial Easy Expander

Theorem 8.14. *For $\varepsilon > 0$, let $B \geq 8/\varepsilon$ and $\log B \mid B$. Given an $(\frac{N}{B^3 C}, (1-\varepsilon)\frac{B}{\log B})$-expander $G_1 = (V_1 \uplus U_1, E_1)$ with degree $\frac{B}{\log B}$ and $|V_1| = |U_1| = \frac{N}{B^2}$, one can construct a $(\frac{N}{B^2 C}, (1-\varepsilon)e^{3/\varepsilon}\frac{B}{\log B})$-expander $G = (V \uplus U, E)$ with degree B and $|V| = |U| = \frac{N}{B}$ such that a permutation described by the block graph G can be performed with $\mathcal{O}\left(\frac{N}{PB}\log_{\mathrm{d}} B\right)$ I/Os for $\mathrm{d} = \max\left\{2, \min\left\{\frac{M}{B}, \frac{N}{PB}\right\}\right\}$.*

To prove the theorem, we change our view to conductors. First observe the well-known relationship between lossless conductors and expanders:

Lemma 8.15. *A (k_{max}, ε)-lossless conductor $C : (n) \times (d) \to (m)$ induces a (K_{max}, δ) expander $G = (U \uplus V, E)$, and vice versa, where $U = (n)$, $V = (m)$, $K_{max} = 2^{k_{max}}$, $\delta = (1-\varepsilon)2^d$, and left-degree $D = 2^d$ in the following way*

$$\{x, y\} \in E \text{ iff } \exists a : C(x,a) = y\,.$$

Proof. "conductor $\Rightarrow$ expander":

For any vertex set $S \subseteq V_L$ with $|S| = 2^k \leq 2^{k_{max}}$ the flat source U_S with support S has min-entropy k. By definition of C, there is $\tilde{Y}$ with min-entropy $k+d$ which is ε-close to $Y = C(U_S, U_d)$. Considering the definition of ε-close, when summing $p_{\tilde{Y}}$ over the support of Y, we get $\sum_{i \in Supp(Y)} p_{\tilde{Y}}(i) \geq 1-\varepsilon$. Since $p_{\tilde{Y}}(i) \leq 2^{-(k+d)}$ for all $i \in (m)$, the support of Y has to have size at least $(1-\varepsilon)2^{k+d}$, i.e. the neighbourhood of S has size at least $(1-\varepsilon)2^{k+d}$.

"expander $\Rightarrow$ conductor": To this end, we will show that for an arbitrary k-source X, $C(X, U_d)$ as induced by G is a $(k+d, \varepsilon)$-source. We start by showing this for flat k-sources X only. Let $S \subseteq V$ be the vertices corresponding to $Supp(X)$. Since G is a (K_{max}, δ)-expander S has at least $\delta|S| = (1-\varepsilon)2^{k+d}$ neighbours. Each vertex in the neighbourhood of S has at least one incoming edge. Thus, for all $i \in Supp(C(X, U_d))$, it holds $\Pr[C(X, U_d) = i] \geq 2^{-k} \cdot 2^{-d}$. The additional probability above $2^{-k} \cdot 2^{-d}$ for each i which has to be shifted to form a flat $(k+d)$-source is bounded from above by $1 - \delta|S| \cdot 2^{-(k+d)} = \varepsilon$. Therefore, $C(X, U_d)$ is a $(k+d, \varepsilon)$-source.

Now let us generalise this to all kinds of k-sources. By Lemma 8.11, we can write the probability distribution of X as a convex combination $p_X(x) = \sum_i \alpha_i p_{X_i}(x)$ with $\sum_i \alpha_i = 1$, $\alpha_i > 0$ where X_i are flat k-sources. Recall that $Y = C(X, U_d)$ has probability distribution $p_Y(j) = \sum_i \alpha_i p_{Y_i}(j)$. For each i, let $\tilde{Y}_i$ be a $(k+d)$-source which is ε-close to $Y_i = C(X_i, U_d)$. It is easy to see that by definition of ε-close, also $\tilde{Y}$ with $p_{\tilde{Y}}(j) = \sum_i \alpha_i p_{\tilde{Y}_i}(j)$ for all $j \in \mathrm{image}(C)$ is ε-close to Y because

$$\sum_{j \in Supp(Y)} |p_Y(j) - p_{\tilde{Y}}(j)|^+ \leq \sum_{j \in Supp(Y)} \sum_i \alpha_i |p_{Y_i}(j) - p_{\tilde{Y}_i}(j)|^+ \leq \varepsilon$$

since $\sum_i \alpha_i = 1$. □

With this, we can reduce the proof of the theorem to conductors. In a first step, we will explain the general construction of the desired conductor and prove its expansion properties. This is followed by a proof of existence of the required parts including a precise parametrisation. Finally, the degree of the corresponding graph is considered to show that it is a block graph of a valid permutation. By presenting an algorithm, we demonstrate that in terms of asymptotic I/O complexity the described permutations are no harder than BMMC permutations.

Lemma 8.16. *Let $\langle E_1, C_1\rangle : (n-2b)\times(b-a) \to (n-2b)\times(b-a)$ be an $(n-3b+a-c_1,\varepsilon)$-lossless permutation conductor and let $E_2 : (2b-a)\times(a)\to(b)$ be a $(b-a-c_2,\varepsilon)$-lossless conductor. Then the function $E : (n-b)\times(b)\to(n-b)$ with*

- $y_1 = E_1(x_1, r_2)$
- $z = C_1(x_1, r_1)$
- $y_2 = E_2(zx_2, r_2)$
- $E(x_1x_2, r_1r_2) = y_1y_2$

is an $(n-2b+a-c_1, b-a-c_2, 2\varepsilon)$-conductor (see Figure 8.3).

Proof. Let k be the total min-entropy in (X_1X_2). Because it is unknown how much entropy remains on which bits, we partition $Supp(Y_1)$ into

$$\mathcal{A} = \{i \in Supp(Y_1) \mid H_\infty(ZX_2 \mid Y_1 = i) \le b-a-c_2\}$$

and $\mathcal{B} = Supp(Y_1)\setminus\mathcal{A}$.

For any $i\in\mathcal{A}$, since E_2 is a $(b-a-c_2,\varepsilon)$-lossless conductor, we find that $(Y_2 \mid Y_1 = i)$ is a $(k'+a,\varepsilon)$-source with $k' = H_\infty(ZX_2 \mid Y_1 = i)$, i.e. there is a $\tilde{Y}_{2,i}$ ε-close to $(Y_2 \mid Y_1 = i)$ with

$$\max_{j\in(b)} \Pr\left[\tilde{Y}_{2,i} = j\right] \le \max_{j\in(2b-a)} \Pr\left[ZX_2 = j \mid Y_1 = i\right]\cdot 2^{-a}. \tag{8.1}$$

Furthermore, $\langle E_1, C_1\rangle$ is a permutation conductor so that $H_\infty(X_1X_2R_1) = H_\infty(Y_1ZX_2) = k+b-a$. By definition of the min-entropy, this is equivalent to

$$\max_{i\in(n-2b)} \Pr\left[Y_1 = i\right] \max_{j\in(2b-a)} \Pr\left[ZX_2 = j \mid Y_1 = i\right] = 2^{-(k+b-a)}.$$

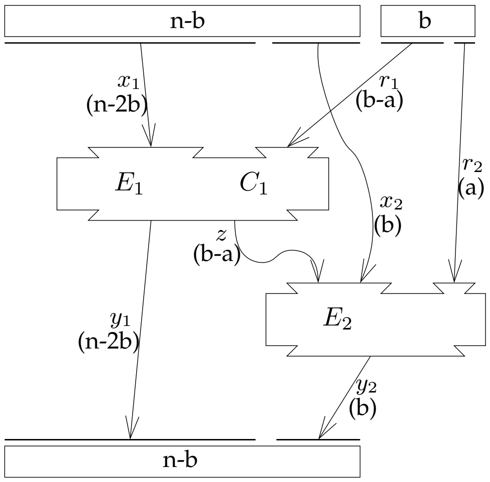


Figure 8.3: The composition of simpler expanders towards a bigger expansion. The upper and lower rectangles correspond to bit positions of the overall input/output ($n - b$ bits specify vertices/blocks, the additional b bits of the input specify edges/records within a block). The two inner shapes correspond to conductors with funnels being input, and output respectively. Each edge is annotated with the variable name and the number of bits in brackets.

Together with (8.1) we get

$$\max_{i\in\mathcal{A}} \Pr\left[Y_1 = i\right] \max_{j\in(b)} \Pr\left[\tilde{Y}_{2,i} = j\right] \le 2^{-(k+b)}. \tag{8.2}$$

For the case $Y_1 = i$, $i \in \mathcal{B}$, the random variable $(Y_2 \mid Y_1 = i)$ is ε-close to a source with min-entropy $b - c_2$. Furthermore, since E_1 is an $(n-3b+a-c_1, \varepsilon)$-lossless conductor, $Y_1 = E_1(X_1, R_1)$ is a $(\min\{k'' + b - a, n - 2b - c_1\}, \varepsilon)$-source with $k'' = H_\infty(X_1) \ge k - b$. Thus, there exists a $\tilde{Y}_1$ which is ε-close to Y_1 and has min-entropy at least $\min\{k - a, n - 2b - c_1\}$. Together with the $(b - c_2)$-sources $\tilde{Y}_{2,i}$ that are ε-close to $(Y_2 \mid Y_1 = i)$ for each $i \in \mathcal{B}$, we obtain

$$\max_{i\in\mathcal{B}} \Pr\left[\tilde{Y}_1 = i\right] \max_{j\in(b)} \Pr\left[\tilde{Y}_{2,i} = j\right] \le \max\{2^{-(k-a)}, 2^{-(n-2b-c_1)}\} \cdot 2^{-(b-c_2)} \tag{8.3}$$

where we chose $\tilde{Y}_{2,i}$ uniformly distributed over (b) for all $i \in \mathit{Supp}(\tilde{Y}_1) \setminus \mathit{Supp}(Y_1)$.

Observe that (8.2) is a stronger bound than (8.2), and hence (8.3) is a general upper bound for $i \in \mathcal{A} \cup \mathcal{B}$. Thus, using (8.3) we can define $(\tilde{Y}_1\tilde{Y}_2)$ by

$\Pr\left[\tilde{Y}_2 = j \mid \tilde{Y}_1 = i\right] = \Pr\left[\tilde{Y}_{2,i} = j\right]$ so that we get

$$\max_{i \in (n-2b)} \Pr\left[\tilde{Y}_1 = i\right] \max_{j \in (b)} \Pr\left[\tilde{Y}_2 = j \mid \tilde{Y}_1 = i\right] \le \max\{2^{-(k+b-a-c_2)}, 2^{-(n-b-c_1-c_2)}\}, \tag{8.4}$$

i.e. $(\tilde{Y}_1\tilde{Y}_2)$ is a $\min\{k+b-a-c_2, n-b-c_1-c_2\}$-source.

Finally, observe that $(\tilde{Y}_1\tilde{Y}_2)$ is 2ε-close to (Y_1Y_2): For any $i \in Supp(Y_1)$, we know that there exists $\tilde{Y}_{2,i}$ with the desired min-entropy which is ε-close to $(Y_2 \mid Y_1 = i)$, i.e.

$$\sum_{j \in Supp(Y_2 \mid Y_1 = i)} |p_{(Y_2|Y_1=i)}(j) - p_{\tilde{Y}_{2,i}}(j)|^+ \le \varepsilon .$$

The distance of $(Y_1\tilde{Y}_2)$ to (Y_1Y_2) is given by

$$\sum_{ij \in Supp(Y_1Y_2)} |p_{Y_1}(i)p_{(Y_2|Y_1=i)}(j) - p_{Y_1}(i)p_{\tilde{Y}_{2,i}}(j)|^+$$

which is obviously upper bounded by ε because $\sum_i p_{Y_1}(i) = 1$. Similarly, for Y_1 and $\tilde{Y}_1$ holds

$$\sum_{i \in Supp(Y_1)} |p_{Y_1}(i) - p_{\tilde{Y}_1}(i)|^+ \le \varepsilon$$

so that by the same argument $(\tilde{Y}_1\tilde{Y}_2)$ is 2ε-close to (Y_1Y_2). □

To apply Lemma 8.16 we have to show the existence of the required conductors. This will be shown by the following lemma.

Lemma 8.17. *For every $0 < \varepsilon < 1$, there is a (k_{max}, ε)-lossless permutation conductor $\langle C, D\rangle : (n_1) \times (d) \mapsto (n_2) \times (d')$ with $k_{max} = n_2 - d - \frac{3\log(e)}{\varepsilon} - \log(1-\varepsilon)$ when $d \ge \log(n_1 - n_2 + d + 3/\varepsilon)$.*

Proof. We show the existence of an appropriate expander graph $G = (U \cup V, E)$ which has on each vertex set U and V constant degree (this implies a permutation conductor). Applying Lemma 8.15 proves the lemma. Let again $N_1 = 2^{n_1}$, $N_2 = 2^{n_2}$, $K_{max} = 2^{k_{max}}$ and $D = 2^d$. Hence, we want to prove the existence of a $(K_{max}, (1-\varepsilon)D)$-expander graph $G = (U \uplus V, E)$ where $|U| = N_1$, $|V| = N_2$, with all $u \in U$ having (out-)degree D, and all $v \in V$ having (in-)degree N_1D/N_2. To ensure the expansion of G, we require for each set $S \subseteq U$, $|S| = K \le K_{max}$ that there is no set $T \subseteq V$, $|T| < (1-\varepsilon)DK$ such that all DK edges from S go to T. For a uniformly chosen regular graph, let $X_{S,T}$ be the random variable which is 1 if all edges from S do go to T. Now fix the sets S and T. We can think of drawing the neighbours for each vertex in U one after another. Since we aim for a graph which has constant degree on each side, during this process it can appear that a node in V already has full degree and can not be drawn anymore. However, the probability for a node in S to have a neighbour in T is bounded from above by $|T|/N_2$. This yields an overall probability for $X_{S,T} = 1$ of at most $(|T|/N_2)^{DK}$.

If the expectation of $\sum_{S,T} X_{S,T}$ over all such sets S and T is strictly less than 1, there is a graph which is a $(K_{max}, (1-\varepsilon)D)$-expander. For a fixed set S, it is sufficient to consider only sets T that have size exactly $(1-\varepsilon)DK$. Hence, we want to bound

$$\mathrm{E}\left[\sum_{S,T} X_{S,T}\right] \leq \sum_{K=1}^{K_{max}} \binom{N_1}{K}\binom{N_2}{(1-\varepsilon)DK}\left(\frac{(1-\varepsilon)DK}{N_2}\right)^{DK}$$

from above by a constant smaller 1. Therefore, we show that the choice of D and K_{max} implies an upper bound of e^{-K} on the Kth term of the sum (note that $\sum_{i=1}^{\infty} e^{-i} = \frac{1}{e-1} < 1$).

Taking logarithm and estimating binomial coefficients, we need to show

$$K\left(\ln\frac{N_1}{K}+1\right) + (1-\varepsilon)DK\left(\ln\frac{N_2}{(1-\varepsilon)DK}+1\right) + DK\ln\frac{(1-\varepsilon)DK}{N_2} < -K\,.$$

Rearranging terms and dividing by K yields

$$2 + (1-\varepsilon)D + \ln\frac{N_1}{K} < \varepsilon D\ln\frac{N_2}{(1-\varepsilon)DK}\,, \tag{8.5}$$

and separating K

$$2 + (1-\varepsilon)D + \ln N_1 + (\varepsilon D - 1)\ln K < \varepsilon D\ln\frac{N_2}{(1-\varepsilon)D}\,. \tag{8.6}$$

Recall that we require $D \geq \log\frac{N_1 D}{N_2} + \frac{3}{\varepsilon}$ in the lemma, and thus $D > 1/\varepsilon$. Hence, the left-hand side of (8.6) is monotonically increasing in K and is sufficient to prove (8.5) for $K = K_{max}$.

Furthermore, we claim $K_{max} = N_2/((1-\varepsilon)e^{3/\varepsilon}D)$ so that it holds

$$2 + (1-\varepsilon)D + \ln\frac{N_1}{K_{max}} \leq 2 + (1-\varepsilon)D + \ln\frac{(1-\varepsilon)e^{3/\varepsilon}DN_1}{N_2} \leq 2 + 2D\,. \tag{8.7}$$

Similarly, by substituting K_{max} on the right-hand side of (8.5), we obtain

$$\varepsilon D\ln\frac{N_2}{(1-\varepsilon)DK} \geq \varepsilon D\ln e^{3/\varepsilon} = 3D\,. \tag{8.8}$$

Finally, (8.5) is implied by (8.7) and (8.8) because $D \geq 3/\varepsilon$ and $\varepsilon < 1$. $\square$

Lemma 8.18. *For every* $0 < \varepsilon < 1$*, there is a* $(b - a - c_2, \varepsilon)$*-lossless permutation conductor* $\langle C, D\rangle : (2b - a) \times (a) \mapsto (b) \times (b)$ *with*

- $c_2 = \frac{3\log(e)}{\varepsilon} + \log(1 - \varepsilon)$
- $a = \log(b + 3/\varepsilon)$.

Lemma 8.19. *For every* $0 < \varepsilon < 1$*, there is a* $(n - 3b + a - c_1, \varepsilon)$*-lossless permutation conductor* $\langle C, D\rangle : (n - 2b) \times (b - a) \mapsto (n - 2b) \times (b - a)$ *with*

- $c_1 = \frac{3\log(e)}{\varepsilon} + \log(1 - \varepsilon)$
- $b - a = \log(6/\varepsilon)$.

Proof. Observe that $d \geq \log(6/\varepsilon)$ implies $d \geq \log(d+3/\varepsilon)$ because $3/\varepsilon > \log(6/\varepsilon)$ (which is equivalent to $8 \cdot 2^{1/\varepsilon} > 6 \cdot \frac{1}{\varepsilon}$ and true for $0 < \varepsilon < 1$). □

Hence, there is a function $E_2 : (2b - a) \times (a) \mapsto (b)$ which is a $(b - a - c_2, \varepsilon)$-lossless conductor for $a = \log(2b + 3/\varepsilon)$ and $c_2 = \frac{4}{\varepsilon} + \log(1 - \varepsilon)$ under the setting $B \geq 6/\varepsilon$. The function E has constant-right degree if E_2 has constant right degree because $\langle E_1, C_2\rangle$ is a permutation. Being a permutation implies that the mapping of $X_1X_2R_1R_2$ to $Y_1ZX_2R_2$ is bijective. Having E_2 from Lemma 8.17 with right-degree B, there are B inputs mapped on each value of Y_1Y_2.

It remains to show that the result of the construction of Lemma 8.16 describes easy permutations.

Lemma 8.20. *A permutation with block graph* G_2 *requires* $\mathcal{O}\left(\frac{N}{PB}\log_{\mathrm{d}} B\right)$ *I/Os for* $\mathrm{d} = \max\left\{2, \min\left\{\frac{M}{B}, \frac{N}{PB}\right\}\right\}$.

Proof. To prove the lemma, we consider the block graph after performing an intermediate permutation. The block graph is then used to describe the remaining permutation to obtain the desired output permutation. More detailed, we consider how the edges within the block graph are changed by the intermediate permutation. Our aim is to transform the block graph into a matching (cf. Figure 8.2 (b)). Then, the inter-block permutations have been realised. Only the ordering of the output blocks and their internal order is not necessarily correct. However, during the last output of each block (or with an additional scan), the blocks can be written with the desired order which yields the complete permutation.

First, we partition the index space of blocks $(n - b)$ into sets $P_1, \ldots, P_{N/B^2}$ such that $P_i = \{ij \in (n - b) \mid j \in (b)\}$, i.e. the first $n - 2b$ bits are i. The sets P_i have obviously size $2^b = B$. Note that for any $v, w \in P_i$ and $j \in (b)$, $E_1(v, j) = E_1(w, j)$ because E_1 considers only the first $n - 2b$ bits (and the last $b - a$ bits). Because there are B nodes in each P_i, and we have constant degree B, we can perform the following permutation, leading to a simpler block graph. For each P_i, we choose an arbitrary ordering $v_1, \dots, v_B$ of the blocks in P_i and assign the kth record of each block to v_k. This reordering of the records in a group of B blocks corresponds to a transposition of the $B \times B$ records (considering each block as a column) in P_i, which is obviously possible by sorting with $\mathcal{O}\left(\frac{N}{PB} \log_{\mathrm{d}} B\right)$ I/Os. In means of the block graph, this corresponds to ordering the nodes $v_1, \dots, v_B$ and assigning all the kth edges to v_k, i.e. replacing the edges (v_k, j), (v_j, k) by (v_k, k), (v_j, j) for each $1 \le j, k \le B$. After this transformation, the first $n - 2b$ bits of each node's neighbours are the same, given by E_1 since the first $n - 2b$ bits and the last $b - a$ bits are the same for all records in a block. In other words, for all v there is i such that $\bigcup_k E(v, k) \subseteq P_i$ in the resulting block graph.

In a second step, we define $Q_1, \dots, Q_{N/B^2}$ where $Q_i = \{j \in (n - b) \mid \bigcup_k E(j, k) \subseteq P_i\}$. Note that this is a partition of $(n - b)$ after the first change operations were applied. Furthermore, since $|P_i| = B$ and E is B-regular, there are B^2 records j, k with $E(j, k) \in P_i$. Every $j \in Q_i$ has all its B neighbours in P_i. Hence, there are exactly B nodes in Q_i. Thus, since the neighbourhood of Q_i is P_i, and $|Q_i| = |P_i| = B$, the records in the blocks of each Q_i can be sorted with the PEM merge sort, resulting in another $\mathcal{O}\left(\frac{N}{PB} \log_{\mathrm{d}} B\right)$ I/Os. With this reordering, all the records that cause an edge to node $v_k \in P_i$ are moved into the same block. Hence, in the resulting block graph each node has only one neighbour. The block graph now corresponds to a matching which finalises the permutation with a total number of $\mathcal{O}\left(\frac{N}{PB} \log_{\mathrm{d}} B\right)$ I/Os.

□

8.3 Conclusion

We considered two classes of permutations that are fundamentally easier in their I/O complexity than the worst-case over all permutations. The first one is the class of BMMC permutations for which the (serial) I/O complexity was determined in [CW93]. Their rather complicated proof of an upper bound was simplified here using insights that are gained from the description of the permutation with the block graph. Furthermore, our view leads to an easy parallelisation involving N/B^2 independent tasks.

Having seen that BMMC permutations induce a block graph with small connected components, as a second class, we considered permutations that induce block graphs with high connectivity that are fairly good expanders. In the construction of this class, we started with B independent and similar permutations of N/B records each (used as the expander E_1). Such a permutation can be realised with $\mathcal{O}\left(\frac{N}{PB}\log_{\mathrm{d}(N)} B\right)$ I/Os. It is hence not more difficult than a BMMC permutation. The connected components of the block graph on the contrary have larger size ($\Theta\left(N/B^2\right)$ instead of $\mathcal{O}(B)$ for BMMC permutations). Any structure can be chosen for the B similar permutations, especially expansion within the connected components is possible. However, it is not connected and its connected components are all of the same kind. Since the block graph of the overall permutation consists of at least B connected components, such a graph cannot be a (K_{max}, B)-expander for any $K_{max} > N/B^2$.

With our construction in Section 8.2, we amplified the expansion properties in that we showed the existence of a class of permutations with a block graphs that is an $(\Omega(N/B), \Omega(B/\log B))$-expander. This separates from the trivial setting of having B disconnected expander graphs and shows that expansion of the block graph, even up to this extend, cannot be a property used to identify difficult permutations. Note that with a similar construction, one can easily change the size of x_2 and z to c and $c - a$ respectively, such that an $(\Omega\left(CN/B^2\right), \Omega(B/\log C))$-expander is obtained for arbitrary $C = 2^c \leq B$ and $a = \log c$.

9 Conclusion

Throughout this thesis, we investigated several computational tasks that depend on sparse matrices, in order to gain an understanding of their complexity in the PEM model. Computations were always carried out over a semiring which guarantees the independent evaluation of each elementary product. For all the considered sparse matrix computation tasks, we presented I/O-optimised algorithms and derived lower bounds on the number of I/Os that are required by any program in the worst- and average-case. All our lower bounds hold not only over uniform algorithms, but bound the number of I/Os required for any non-uniform program, i.e. a program that is designed for a certain matrix conformation. On the contrary, many of the (uniform) algorithms we presented can be proven to be asymptotically optimal for parameter ranges that can be considered most relevant in real world settings.

Clearly, the results presented here are theoretical in nature and some of the presented algorithms are stated more to complement the lower bounds than to be implemented. The theoretical nature of our results allows for clearly formulated mathematical theorems, which comes at the price of abstraction in our model of computation. We neglected everything but the memory access patterns of a program, and disallowed Strassen-like algorithms. Furthermore, our focus is on worst-case behaviour whereas in practice the structure of the input can often be exploited. However, our theoretical understandings give important indications on the limits of what can be expected from a practical algorithm. They can also be considered as a reference point for the design of practical algorithms that reduce the worst-case number of cache misses, which is especially relevant if the sparse matrix is not known to have any special distribution of its non-zero entries. In this context, we remark again that the benefit of Strassen-like algorithms is not clear for sparse matrix multiplication (below certain sparsity thresholds, no helpful techniques are known). Furthermore, the well-known fact that memory access patterns can be improved by applying a sorting procedure arises many times throughout our analyses. In practice, a simplified and adjusted sorting procedure often turns out to be efficient, even for structured matrices.

Summary In order to derive upper and lower bounds, we first extended some important known techniques from the I/O-model to the PEM model. In Chapter 3, we then reduced the number of considered tasks by presenting a reduction of the computation of the matrix vector product SpMV to the computation of bilinear products Bil, and vice versa. Hence, it suffices to consider one of the tasks to obtain lower and upper bounds – at least in a non-uniform fashion – for the respective other task. Only the I/Os induced for parallel gather and scatter tasks differ since the evaluated functions rely on a different number of input variables.

This reduction was used in Chapter 4 to obtain algorithms and lower bounds on the parallel I/O complexity of both, SpMV and Bil, where the sparse matrix can be in column major, row major, or best-case layout. We extended previous work by Bender et al. [BBF+10] to the parallel processor case involving non-square matrices that are multiplied with multiple vectors simultaneously. Therein, the number of vectors that are multiplied with the same matrix is upper bounded by $w \leq B$. Given the tall-cache assumption $M \geq B^{1+\varepsilon}$ for constant ε and a number of processors P such that each processor reads at least $B^{1+\varepsilon}$ records in an equal partitioning of the data among the processors, the asymptotic parallel I/O complexity was determined.

For higher numbers of vectors – which corresponds to the multiplication with a dense matrix (SDM) having more than B columns – the parallel I/O complexity was analysed in Chapter 5. In contrast to the previous chapters, the analysis of SDM is based on the consideration of subgraphs consisting of a limited number of edges within random bipartite graphs. By a derandomisation argument, we could show that a denser than average subgraph can be found in time proportional to the size of the matrix $N_x N_y$ (if such a density exists in the worst-case over the considered random bipartite graphs). For matrices that are beyond a certain density threshold ($H \geq (N_x N_y / M) \log^2((N_x + N_y)/M)$), a more practical algorithm was shown to be optimal. This algorithm is an extension of the tile-based cache aware algorithm for dense matrix multiplications. The obtained upper and lower bounds on the I/O complexity are again matching, given the tall-cache assumption and number of processors that allows each processor to fill its entire memory during an (asymptotically optimal) algorithm.

A similar technique was applied to the multiplication of two sparse matrices (SSM). However, we could not exploit denser than average subgraphs algorithmically. Also our lower bound inspired by Chapter 5 eludes generality. Only for a subclass of algorithms – the class of pseudo rectangular algorithms – lower bounds could be obtained by this technique. Using a simple reduction, we derived lower bounds for SSM from Chapter 4 which are matching the presented algorithms for $k_1, k_2 \leq \min\{B, N/(2B)\}$.

Finally, our techniques for sparse matrix computation were applied to the MapReduce framework in Chapter 7. This allows for an analysis of the parallel I/O complexity induced by the shuffle step. Since this step is the only explicit communication phase, which is performed in each round, this yields bounds on the (parallel) I/O-efficiency of the MapReduce framework.

In a last chapter, we investigated the complexity of two classes of permutations. As shown by the I/O complexity of the presented algorithms, both classes reflect rather easy permutations. In the PEM model, the considered permutations are no harder than dense matrix transposition whose I/O complexity is among the easiest, non-trivial ones, and becomes trivial under the tall-cache assumption. This rejects a candidate class for difficult permutations, namely that of permutations with block graphs being good expanders.

Open Problems All our lower bounds rely, similarly to the ones in this context in [AV88] and [BBF+10], on a counting argument. Though this shows the existence of difficult instances and bounds the minimum number of I/Os, there is no characterisation of which (permutation) matrices are difficult to multiply with. In order to obtain algorithms that perform well on instances that appear in practice, it is desirable to gain an understanding of what makes an instance hard. With such knowledge, algorithms can be adapted to classes of instances that have a different worst-case behaviour.

Furthermore, we only considered worst-case or average-case complexities in this thesis. To find an optimal program for a given permutation or matrix vector multiplication is another problem of its own. In [Lie09], it is shown that finding an optimal program is $\mathcal{NP}$-complete, even for $B = 1$ if M is part of the input. They present an algorithm which is fixed-parameter tractable in M and the number of non-compulsory I/O (that are not required to read/write the input/output). Moreover, they showed recently [LJ12] that the problem is even $\mathcal{NP}$-hard for fixed $M \geq 2$, and gave an approximation algorithm for $M = 2$.

The lower bounds presented in this thesis hold for any non-uniform program. However, especially in the parallel case, the matrix conformation is crucial for a proper load-balancing among the processors. This load-balancing which seems required if the matrix conformation is not known to the algorithm cannot be characterised using the techniques in Chapter 2. In this context, it is for instance not clear wether a prefix sum computation is required or not, to transpose a sparse matrix in column major layout directly. Recall that we required a prefix sum computation in order to determine the target position of a record.

In a similar manner, the CREW policy with its asymmetric access policy requires balanced gather tasks. We were not able to derive a uniform parallel algorithm for the direct matrix vector multiplication with a matrix in column major layout. Here, we note that this problem does not arise for BIL, nor if each record is annotated with both, its column rank and its row rank.

In Chapter 2, we improved on the I/O-efficiency of the PEM merge sort from [AGNS08] for sorting N records with $P \geq N/B^2$ processors. A lower bound for permuting indicates that the algorithm is optimal for any number of processors $P \leq N/B$, unless the direct algorithm is optimal. A bound in the comparison model holds even in a setting where the direct algorithm is optimal for permuting. Recall that there have to be sufficient comparisons to determine uniquely which permutation has to be realised by the sorting instance. However, there are settings, when a sufficient number of comparisons have been performed, but there is no global knowledge of what permutation has to be realised. Excluding such cases, we only have matching lower bounds for $P \leq \frac{N}{B \log^{\varepsilon} N}$ for constant $\varepsilon > 0$. It would be interesting to either improve the lower bounds, or construct an algorithm to identify, and then to perform the permutation directly.

Finally, our attempts to identify the asymptotical I/O complexity of SSM did only yield partial results. The I/O complexity for most parameter ranges is not understood yet. It seems that our techniques – especially the counting based techniques – are not strong enough to tackle this problem, at least not for non-trivial block sizes. A stronger technique that is not based on a counting argument could moreover lead to an understanding of the difficult instances. However, even for block size $B = 1$, SSM remains an interesting problem. In this case, we obtained bounds for a special class of algorithms. For a bound on the I/O complexity of any algorithm, we lack of more involved probabilistic estimations.

List of Figures

List of Tables

Index

Bibliography

[ACP10] Rasmus Resen Amossen, Andrea Campagna, and Rasmus Pagh. Better size estimation for sparse matrix products. In *Proceedings of the 13th international conference on Approximation, and 14 the International conference on Randomization, and combinatorial optimization: algorithms and techniques*, APPROX/RANDOM'10, pages 406–419, Berlin, Heidelberg, 2010. Springer-Verlag.

[AGNS08] Lars Arge, Michael T. Goodrich, Michael Nelson, and Nodari Sitchinava. Fundamental parallel algorithms for private-cache chip multiprocessors. In *Proceedings of SPAA '08*. ACM, 2008.

[AGS10] Lars Arge, Michael T. Goodrich, and Nodari Sitchinava. Parallel external memory graph algorithms. In *IPDPS*, pages 1–11. IEEE, 2010.

[ASU11] Noga Alon, Amir Shpilka, and Chris Umans. On sunflowers and matrix multiplication. *Electronic Colloquium on Computational Complexity (ECCC)*, 18:67, 2011.

[AV88] Alok Aggarwal and Jeffrey S. Vitter. The input/output complexity of sorting and related problems. *Communications of the ACM*, 31(9):1116–1127, 1988.

[BBF$^+$07] Michael A. Bender, Gerth Stølting Brodal, Rolf Fagerberg, Riko Jacob, and Elias Vicari. Optimal sparse matrix dense vector multiplication in the I/O-model. In *Proceedings of SPAA '07*, pages 61–70, New York, NY, USA, 2007. ACM.

[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*, 47:934–962, 2010.

[BCRL79] Dario Bini, Milvio Capovani, Francesco Romani, and Grazia Lotti. $O(n^{2.7799})$ complexity for $n \times n$ approximate matrix multiplication. *Information Processing Letters*, 8:234–235, 1979.

[BDH95] Armin Bäumker, Wolfgang Dittrich, and Friedhelm Meyer auf der Heide. Truly efficient parallel algorithms: c-optimal multisearch for an extension of the BSP model (extended abstract). In *Proceedings of ESA'95*, pages 17–30. Springer, 1995.

[BDHS10] Grey Ballard, James Demmel, Olga Holtz, and Oded Schwartz. Communication-optimal parallel and sequential cholesky decomposition. *SIAM J. Scientific Computing*, 32(6):3495–3523, 2010.

[BDHS11a] G. Ballard, J. Demmel, O. Holtz, and O. Schwartz. Graph expansion and communication costs of fast matrix multiplication. In *Proceedings of the 23rd ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '11, pages 1–12, New York, NY, USA, 2011. ACM.

[BDHS11b] G. Ballard, J. Demmel, O. Holtz, and O. Schwartz. Minimizing communication in numerical linear algebra. *SIAM Journal on Matrix Analysis and Applications*, 32(3):866–901, 2011.

[BF03] Gerth Stølting Brodal and Rolf Fagerberg. On the limits of cache-obliviousness. In *STOC '03: Proceedings of the thirty-fifth annual ACM symposium on Theory of computing*, pages 307–315, New York, NY, USA, 2003. ACM.

[Blä99] Markus Bläser. A 5/2 n^2-lower bound for the rank of n×n matrix multiplication over arbitrary fields. In *Proceedings of FOCS '99*, pages 45–50. IEEE Computer Society, 1999.

[CDR86] Stephen Cook, Cynthia Dwork, and Ru&duml;ger Reischuk. Upper and lower time bounds for parallel random access machines without simultaneous writes. *SIAM Journal on Computing*, 15(1):87–97, February 1986.

[CKSU05] Henry Cohn, Robert Kleinberg, Balázs Szegedy, and Christopher Umans. Group-theoretic algorithms for matrix multiplication. In *In Foundations of Computer Science. 46th Annual IEEE Symposium on 23–25 Oct 2005*, pages 379–388, 2005.

[Coh98] Edith Cohen. Structure prediction and computation of sparse matrix products. *Journal of Combinatorial Optimization*, 2(4):307–332, 1998.

[Col88] Richard Cole. Parallel merge sort. *SIAM Journal on Computing*, 17(4):770–785, 1988.

[Col93] Richard Cole. Correction: Parallel merge sort. *SIAM Journal on Computing*, 22(6):1349, 1993.

[Cor93] Thomas H. Cormen. Fast permuting on disk arrays. *Journal of Parallel and Distributed Computing*, 17(1-2):41–57, 1993.

[CW90] Don Coppersmith and Shmuel Winograd. Matrix multiplication via arithmetic progressions. *Journal of Symbolic Computation*, 9(3):251 – 280, 1990. ¡ce:title¿Computational algebraic complexity editorial¡/ce:title¿.

[CW93] Thomas H. Cormen and Leonard F. Wisniewski. Asymptotically tight bounds for performing bmmc permutations on parallel disk systems. In *Proceedings of SPAA '93*, SPAA '93, pages 130–139, New York, NY, USA, 1993. ACM.

[DDE^{+}05] J. Demmel, J. Dongarra, V. Eijkhout, E. Fuentes, R. Vuduc Antoine Petitet, R. C. Whaley, and K. Yelick. Self-adapting linear algebra algorithms and software. *Proc. of the IEEE, Special Issue on Program Generation, Optimization, and Adaptation*, 93(2), 2005.

[DG04] Jeffrey Dean and Sanjay Ghemawat. MapReduce: simplified data processing on large clusters. *Proceedings OSDI'04*, pages 137–150, 2004.

[DG10] Jeffrey Dean and Sanjay Ghemawat. MapReduce: A flexible data processing tool. *Communications of the ACM*, 53(1):72–77, 2010.

[DK03] Martin Dietzfelbinger and Manfred Kunde. A case against using stirling's formula (unless you really need it). *Bulletin of the EATCS*, 80:153–158, 2003.

[DS] David J. DeWitt and Michael Stonebraker. MapReduce: A major step backwards.

[ER60] P. Erdős and R. Rado. Intersection theorems for systems of sets. *Journal of the London Mathematical Society*, page 85–90, 1960.

[FC00] Salvatore Filippone and Michele Colajanni. PSBLAS: A library for parallel linear algebra computation on sparse matrices. *ACM Transactions on Mathematical Software*, 26(4):527–550, December 2000.

[FLPR99] Matteo Frigo, Charles E. Leiserson, Harald Prokop, and Sridhar Ramachandran. Cache-oblivious algorithms. In *Proceedings of FOCS '99*, pages 285–297, New York, NY, 1999. "IEEE Computer Society".

[FMS$^+$10] Jon Feldman, S. Muthukrishnan, Anastasios Sidiropoulos, Cliff Stein, and Zoya Svitkina. On distributing symmetric streaming computations. *ACM Transactions on Algorithms*, 6:1–19, September 2010.

[GJ10a] Gero Greiner and Riko Jacob. Evaluating non-square sparse bilinear forms on multiple vector pairs in the I/O-model. In *Proceedings of MFCS '10*, pages 393–404. Springer, 2010.

[GJ10b] Gero Greiner and Riko Jacob. Evaluating non-square sparse bilinear forms on multiple vector pairs in the I/O-model. Technical Report TUM-I1015, Technische Universität München, September 2010.

[GJ10c] Gero Greiner and Riko Jacob. The I/O complexity of sparse matrix dense matrix multiplication. In *Proceedings of LATIN '10*, pages 143–156. Springer, 2010.

[GJ11] Gero Greiner and Riko Jacob. The efficiency of mapreduce in parallel external memory. *CoRR*, abs/1112.3765, 2011.

[GJ12] Gero Greiner and Riko Jacob. The efficiency of mapreduce in parallel external memory. In *Proceedings of LATIN '12*, pages 433–445, 2012.

[Goo99] Michael T. Goodrich. Communication-efficient parallel sorting. *SIAM Journal on Computing*, 29(2):416–432, 1999.

[GSZ11] Michael T. Goodrich, Nodari Sitchinava, and Qin Zhang. Sorting, searching, and simulation in the mapreduce framework. *CoRR*, abs/1101.1902, 2011.

[HK81] Hong, Jia-Wei and H. T. Kung. I/O complexity: The red-blue pebble game. In *Proceedings of STOC '81*, pages 326–333. ACM, 1981.

[HLW06] Shlomo Hoory, Nathan Linial, and Avi Wigderson. Expander graphs and their applications. *Bulletin of the American Mathematical Society*, 43:439–561, 2006.

[ITT04] Dror Irony, Sivan Toledo, and Alexandre Tiskin. Communication lower bounds for distributed-memory matrix multiplication. *Journal of Parallel and Distributed Computing*, 64(9):1017–1026, 2004.

[jIY00] Eun jin Im and Katherine Yelick. Optimization of sparse matrix kernels for data mining. In *First SIAM Conference on Data Mining*, 2000.

[JS10] Riko Jacob and Michael Schnupp. Experimental performance of I/O-optimal sparse matrix dense vector multiplication algorithms within main memory. Technical Report TUM-I1017, Technische Universität München, 2010.

[KL51] S. Kullback and R. A. Leibler. On information and sufficiency. *Annals of Mathematical Statistics*, 22(1):79–86, 1951.

[KR90] R.M. Karp and V.L. Ramachandran. Parallel algorithms for shared-memory machines. *Handbook of Theoretical Computer Science*, pages 869–941, 1990.

[KSV10] Howard Karloff, Siddharth Suri, and Sergei Vassilvitskii. A model of computation for MapReduce. In *Proceedings of SODA'10*, pages 938–948. SIAM, 2010.

[KW03] Markus Kowarschik and Christian Weiß. An overview of cache optimization techniques and cache-aware numerical algorithms. *Algorithms for Memory Hierarchies*, pages 213–232, 2003.

[Lie09] Tobias Lieber. Combinatorial approaches to optimizing sparse matrix dense vector multiplication in the I/O-model. Master's thesis, Informatik Technische Universität München, 2009.

[LJ12] Tobias Lieber and Riko Jacob. personal communication, 2012.

[MS04] G. I. Malaschonok and E. S. Satina. Fast multiplication and sparse structures. *Program. Comput. Softw.*, 30(2):105–109, March 2004.

[Pan78] V. Ya. Pan. Strassen's algorithm is not optimal trilinear 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*, SFCS '78, pages 166–176, Washington, DC, USA, 1978. IEEE Computer Society.

[PPR$^+$09] Andrew Pavlo, Erik Paulson, Alexander Rasin, Daniel J. Abadi, David J. DeWitt, Samuel Madden, and Michael Stonebraker. A comparison of approaches to large-scale data analysis. In *Proceedings of SIGMOD '09*, pages 165–178. ACM, 2009.

[RJG$^+$07] Franz F. Roos, Riko Jacob, Jonas Grossmann, Bernd Fischer, Joachim M. Buhmann, Wilhelm Gruissem, Sacha Baginsky, and Peter Widmayer. Pepsplice: cache-efficient search algorithms for comprehensive identification of tandem mass spectra. *Bioinformatics*, 23(22):3016–3023, 2007.

[RP96] K. Remington and R. Pozo. NIST sparse BLAS user's guide. Technical report, National Institute of Standards and Technology, Gaithersburg, Maryland, 1996.

[RRP$^+$07] Colby Ranger, Ramanan Raghuraman, Arun Penmetsa, Gary Bradski, and Christos Kozyrakis. Evaluating mapreduce for multi-core and multiprocessor systems. In *Proceedings of HPCA '07*, pages 13–24. IEEE, February 2007.

[RVW00] Omer Reingold, Salil P. Vadhan, and Avi Wigderson. Entropy waves, the zig-zag graph product, and new constant-degree expanders and extractors. In *Proceedings of FOCS 2000*, pages 3–13, 2000.

[SAD$^+$10] Michael Stonebraker, Daniel Abadi, David J. DeWitt, Sam Madden, Erik Paulson, Andrew Pavlo, and Alexander Rasin. MapReduce and parallel DBMSs: Friends or foes? *Communications of the ACM*, 53(1):64–71, 2010.

[Sch81] Arnold Schönhage. Partial and total matrix multiplication. *SIAM J. Comput.*, 10(3):434–455, 1981.

[Shp01] Amir Shpilka. Lower bounds for matrix product. In *Proceedings of FOCS '01*, pages 358–367. IEEE Computer Society, 2001.

[Sil07] Francesco Silvestri. On the limits of cache-oblivious matrix transposition. In *Proceedings of the 2nd international conference on Trustworthy global computing*, TGC'06, pages 233–243, Berlin, Heidelberg, 2007. Springer-Verlag.

[Sit09] Nodari Sitchinava. *Parallel external memory model and algorithms for multicore architectures*. PhD thesis, University of California, Irvine, California, September 2009.

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

[Str69] Volker Strassen. Gaussian elimination is not optimal. *Numerische Mathematik*, 13:354–356, 1969. 10.1007/BF02165411.

[Str86] V. Strassen. The asymptotic spectrum of tensors and the exponent of matrix multiplication. In *Proceedings of the 27th Annual Symposium on Foundations of Computer Science*, SFCS '86, pages 49–54, Washington, DC, USA, 1986. IEEE Computer Society.

[Val90] Leslie G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, August 1990.

[Vas11] Virginia Vassilevska Williams. Breaking the Coppersmith-Winograd barrier. 2011.

[VDY05] Richard Vuduc, James W. Demmel, and Katherine A. Yelick. Oski: A library of automatically tuned sparse matrix kernels. *Journal of Physics: Conference Series*, 16(1):521+, 2005.

[VDY06] Richard Vudac, James W. Demmel, and Katherine A. Yelick. *The Optimized Sparse Kernel Interface (OSKI) Library: User's Guide for Version 1.0.1b*. Berkeley Benchmarking and OPtimization (BeBOP) Group, March 15 2006.

[Vud03] Richard Wilson Vuduc. *Automatic Performance Tuning of Sparse Matrix Kernels*. PhD thesis, University of California, Berkeley, Fall 2003.

[WF81] Chuan-Lin Wu and Tse-Yun Feng. The universality of the shuffle-exchange network. *IEEE Trans. Comput.*, 30(5):324–332, May 1981.

[Whi09] Tom White. *Hadoop: The Definitive Guide*. O'Reilly, first edition edition, June 2009.

[Wil07] Ryan Williams. Matrix-vector multiplication in sub-quadratic time (some preprocessing required. In *Proceedings of SODA '07*, pages 1–11. ACM Press, 2007.

[Win71] Shmuel Winograd. On multiplication of 2x2 matrices. *Linear Algebra and Application*, 4:381–388, 1971.

[YZ04] Raphael Yuster and Uri Zwick. Fast sparse matrix multiplication. In *Proceedings of ESA '04*, volume 3221 of *Lecture Notes in Computer Science*, pages 604–615, 2004.