Communication-Efficient Probabilistic Algorithms: Selection, Sampling, and Checking

Zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften

von der KIT-Fakultät für Informatik des
Karlsruher Instituts für Technologie (KIT)

genehmigte

Dissertation

von

Lorenz Albert Hübschle-Schneider

Tag der mündlichen Prüfung: 26. November 2020

1. Referent: Prof. Dr. Peter Sanders
Karlsruher Institut für Technologie
Deutschland

2. Referent: Univ.Prof. Dr. Scient. Jesper Larsson Träff
Technische Universität Wien
Österreich

Dedicated to my mother
Susanne Hübschle (1959 – 2017)

Abstract

This dissertation focuses on three fundamental problem families in big data systems, for which we develop communication-efficient probabilistic algorithms. In the first part, we consider various *selection* problems, in the second, *weighted sampling* problems, and the third, *checking* of basic operations in big data systems. This is motivated by a growing need for communication efficiency as the network and its usage increasingly dominate supercomputer system cost and energy consumption as well as running times of distributed applications. Surprisingly few communication-efficient algorithms are known for fundamental big data problems, and we close several of these gaps.

We first consider different selection problems, starting with selecting the element with rank k from a large, distributed input. There, we show how this can be achieved without assuming a random distribution of the input by redesigning the pivot selection step. Next, we show that selection from locally sorted sequences—also known as *multisequence selection*—becomes considerably easier if we allow the precise rank of the output to vary in some range. We then describe how this can be used to construct a bulk priority queue, which we later use to construct an algorithm for weighted reservoir sampling. Lastly, we consider finding the globally most frequent objects as well as those whose associated values add up to the highest sums with a sample-based approach.

In the weighted sampling chapter, we begin by giving new construction algorithms for a classical data structure for weighted sampling, *alias tables*. After presenting the first linear-time construction algorithm with only constant memory overhead, we proceed to parallelise this algorithm for shared memory, obtaining the first parallel construction algorithm for alias tables. We then show how to approach the problem in distributed memory with a two-level algorithm. Next, we present an output-sensitive algorithm for weighted sampling with replacement, meaning that it requires time linear in the number of *unique* items in the sample. This algorithm works sequentially as well as in shared and distributed memory and is the first such algorithm in all three categories. We subsequently adapt it to weighted sampling without replacement by combining it with an estimator for the number of unique items in a sample with replacement. Poisson sampling, a generalisation of Bernoulli sampling to weighted items, can be reduced to integer sorting, and we show how an existing approach can be parallelised. For sampling from data streams, we adapt a sequential approach and show how it can be parallelised in a mini-batch model using our bulk priority queue introduced in the selection chapter. The chapter is concluded by an extensive evaluation of our alias table construction algorithms, our output-sensitive algorithm for weighted sampling with replacement, and our weighted reservoir sampling algorithm.

To probabilistically verify the correctness of distributed algorithms, we propose *checkers* for fundamental operations of big data systems. We show that checking numerous operations can be reduced onto two 'core' checkers, namely checking aggregations and whether one sequence is a permutation of another. While multiple approaches exist for the latter and can be easily parallelised, our sum aggregation checker is a novel application of the same data structure that underpins counting Bloom filters and count-min sketches. We implemented both checkers in Thrill, a big data framework. Experiments with deliberately introduced errors confirm the detection accuracy predicted by theory, even when using real-world hash functions with non-ideal properties. Scaling experiments on a supercomputer show that our checkers have very little runtime overhead, in the area of 2 % for near-certainty in the correctness of the result.

Deutsche Zusammenfassung

Diese Dissertation behandelt drei grundlegende Klassen von Problemen in Big-Data-Systemen, für die wir kommunikationseffiziente probabilistische Algorithmen entwickeln. Im ersten Teil betrachten wir verschiedene *Selektionsprobleme*, im zweiten Teil das Ziehen gewichteter Stichproben (*Weighted Sampling*) und im dritten Teil die probabilistische Korrektheitsprüfung von Basisoperationen in Big-Data-Frameworks (*Checking*). Diese Arbeit ist durch einen wachsenden Bedarf an Kommunikationseffizienz motiviert, der daher rührt, dass der auf das Netzwerk und seine Nutzung zurückzuführende Anteil sowohl der Anschaffungskosten als auch des Energieverbrauchs von Supercomputern und der Laufzeit verteilter Anwendungen immer weiter wächst. Überraschend wenige kommunikationseffiziente Algorithmen sind für grundlegende Big-Data-Probleme bekannt. In dieser Arbeit schließen wir einige dieser Lücken.

Zunächst betrachten wir verschiedene Selektionsprobleme, beginnend mit der verteilten Version des klassischen Selektionsproblems, d. h. dem Auffinden des Elements von Rang k in einer großen verteilten Eingabe. Wir zeigen, wie dieses Problem kommunikationseffizient gelöst werden kann, ohne anzunehmen, dass die Elemente der Eingabe zufällig verteilt seien. Hierzu ersetzen wir die Methode zur Pivotwahl in einem schon lange bekannten Algorithmus und zeigen, dass dies hinreichend ist. Anschließend zeigen wir, dass die Selektion aus lokal sortierten Folgen—*multisequence selection*—wesentlich schneller lösbar ist, wenn der genaue Rang des Ausgabeelements in einem gewissen Bereich variieren darf. Dies benutzen wir anschließend, um eine verteilte Prioritätswarteschlange mit Bulk-Operationen zu konstruieren. Später werden wir diese verwenden, um gewichtete Stichproben aus Datenströmen zu ziehen (*Reservoir Sampling*). Schließlich betrachten wir das Problem, die global häufigsten Objekte sowie die, deren zugehörige Werte die größten Summen ergeben, mit einem stichprobenbasierten Ansatz zu identifizieren.

Im Kapitel über gewichtete Stichproben werden zunächst neue Konstruktionsalgorithmen für eine klassische Datenstruktur für dieses Problem, sogenannte *Alias-Tabellen*, vorgestellt. Zu Beginn stellen wir den ersten Linearzeit-Konstruktionsalgorithmus für diese Datenstruktur vor, der mit konstant viel Zusatzspeicher auskommt. Anschließend parallelisieren wir diesen Algorithmus für Shared Memory und erhalten so den ersten parallelen Konstruktionsalgorithmus für Aliastabellen. Hiernach zeigen wir, wie das Problem für verteilte Systeme mit einem zweistufigen Algorithmus angegangen werden kann. Anschließend stellen wir einen ausgabesensitiven Algorithmus für gewichtete Stichproben mit Zurücklegen vor. Ausgabesensitiv bedeutet, dass die Laufzeit des Algorithmus sich auf die Anzahl der eindeutigen Elemente in der Ausgabe bezieht und nicht auf die Größe der Stichprobe. Dieser Algorithmus kann sowohl sequentiell als auch auf Shared-Memory-Maschinen und verteilten Systemen eingesetzt werden und ist der erste derartige Algorithmus in allen drei Kategorien. Wir passen ihn anschließend

an das Ziehen gewichteter Stichproben *ohne* Zurücklegen an, indem wir ihn mit einem Schätzer für die Anzahl der eindeutigen Elemente in einer Stichprobe *mit* Zurücklegen kombinieren. Poisson-Sampling, eine Verallgemeinerung des Bernoulli-Sampling auf gewichtete Elemente, kann auf ganzzahlige Sortierung zurückgeführt werden, und wir zeigen, wie ein bestehender Ansatz parallelisiert werden kann. Für das Sampling aus Datenströmen passen wir einen sequentiellen Algorithmus an und zeigen, wie er in einem Mini-Batch-Modell unter Verwendung unserer im Selektionskapitel eingeführten Bulk-Prioritätswarteschlange parallelisiert werden kann. Das Kapitel endet mit einer ausführlichen Evaluierung unserer Aliastabellen-Konstruktionsalgorithmen, unseres ausgabesensitiven Algorithmus für gewichtete Stichproben mit Zurücklegen und unseres Algorithmus für gewichtetes Reservoir-Sampling.

Um die Korrektheit verteilter Algorithmen probabilistisch zu verifizieren, schlagen wir *Checker* für grundlegende Operationen von Big-Data-Frameworks vor. Wir zeigen, dass die Überprüfung zahlreicher Operationen auf zwei „Kern"-Checker reduziert werden kann, nämlich die Prüfung von Aggregationen und ob eine Folge eine Permutation einer anderen Folge ist. Während mehrere Ansätze für letzteres Problem seit geraumer Zeit bekannt sind und sich auch einfach parallelisieren lassen, ist unser Summenaggregations-Checker eine neuartige Anwendung der gleichen Datenstruktur, die auch zählenden Bloom-Filtern und dem Count-Min-Sketch zugrunde liegt. Wir haben beide Checker in Thrill, einem Big-Data-Framework, implementiert. Experimente mit absichtlich herbeigeführten Fehlern bestätigen die von unserer theoretischen Analyse vorhergesagte Erkennungsgenauigkeit. Dies gilt selbst dann, wenn wir häufig verwendete schnelle Hash-Funktionen mit in der Theorie suboptimalen Eigenschaften verwenden. Skalierungsexperimente auf einem Supercomputer zeigen, dass unsere Checker nur sehr geringen Laufzeit-Overhead haben, welcher im Bereich von 2 % liegt und dabei die Korrektheit des Ergebnisses nahezu garantiert wird.

Acknowledgements

First and foremost I would like to thank Peter Sanders for giving me the opportunity to work in his research group, having a good nose for where it might prove worthwhile to look—and vast knowledge of what others have already tried—, and providing his guidance when needed. Thank you, Jesper Träff, for agreeing to review this dissertation.

Thanks go also to my coauthors Moritz Baum, Carsten Dachsbacher, Julian Dibbelt, Sebastian Lamm, Ingo Müller, Thomas Pajor, Rajeev Raman, Peter Sanders, Emanuel Schrade, and Dorothea Wagner for all the fruitful cooperations, as well as my students Mischa Carl, Jan Ellmers, Hans-Peter Lehmann, Collin Lorbeer, and Julian Wiesler.

A big thank you also goes out to my fantastic colleagues for interesting discussions and making work at the institute fun: Yaroslav Akhremtsev for his enthusiasm for mathematics and willingness to lend it, Julian Arz for leading on sketchy climbs, Michael Axtmann for his attention to detail in benchmarking, Tomáš Balyo for his dry sense of humour, Timo Bingmann for letting me nap on his office sofa, being an excellent partner in pizza ventures, support with Thrill, and providing useful open source codes like tlx and SqlPlotTools, Daniel Funke for putting up with me as his office mate, sharing his fondness of the outdoors, and great discussions over coffee, Simon Gog for his ideas on top trees, Demian Hespe for being an utterly dependable ice cream buddy, Tobias Heuer for always lightening up the mood, Moritz Kobitzsch for planting the seed for going my own way, coffee-wise, in the office, many years later, Sebastian Lamm for dragging me up (and down) Half Dome, Tobias Maier for sharing his passions of board games and cooking as well as his exemplary calm, even during sheer endless debugging sessions, Dennis Schieferdecker for sharing his fruit basket, Sebastian Schlag for a good dram and the occasional reminder to get back to work, Dominik Schreiber for sounds out of bounds and his image editing skills, Christian Schulz for the cups, Daniel Seemaier for taking a dying plant off my hands, Jochen Speck for sharing his vast trove of knowledge, Darren Strash for being a fantastic office mate, and Sascha Witt for quietly walking the walk while others talk the talk. Last but not least, I would like to thank Anja Blancani and Norbert Berger for dealing with all the bureaucratic and technical rubbish that comes with office work.

Many thanks also to Timo Bingmann, Sebastian Lamm, and Tobias Maier for proof-reading parts of this work, as well as all the proof-readers of the papers it builds on, whose names I cannot claim to remember, and the anonymous reviewers whose names I never knew in the first place.

I thank bwHPC, the Steinbuch Centre for Computing, and the Gauss Centre for Supercomputing for providing computing time on their supercomputers, and the Internet Archive for storing archived copies of web resources and implementations that should hopefully lessen the effects of link rot.

Finally, I would like to thank my family and my dear Ljuba for their love and support, as well as the friends who kept me sane in these last six years. Had I met Ljuba earlier, I might have done it in five.

Table of Contents

Chapter 1

1 Introduction and Overview

In this dissertation, we consider how to scale algorithms for three basic problem families: selection problems, checking of fundamental operations in big data processing systems, and random sampling. We study how to solve these problems in distributed-memory settings with a focus on reducing the amount of communication. To achieve this, we leverage the power of randomisation. Sometimes, this also yields improvements over the state of the art in shared-memory parallel or even sequential algorithms.

Before diving into these challenges in the following chapters, we introduce the reader to our motivation and basic concepts used throughout this dissertation. We define what makes an algorithm communication-efficient and discuss why developing such algorithms is an important current and future research topic. The chapter concludes with a summary of the contributions of this dissertation.

In 1965, Gordon Moore, who would go on to become co-founder and CEO of Intel Corporation, observed that the number of components in integrated circuits that yields minimum component costs had doubled every year and predicted that this development would continue for at least another decade [Moo65]. This was six years after the invention of the MOSFET, the basic building block of modern electronics, at Bell Labs. At the time, the integrated circuits that were most cost-effective to produce had around 50 components (today, that number is measured in the billions). In a 1975 speech, he revised the future predicted growth rate to a doubling once every two years [Moo75]. This prediction is commonly known as *Moore's Law*. Like all exponential progressions, Moore's Law must end someday. Yet while its demise has been predicted many times, exponential growth continues to this day.

For decades, the exponential growth in transistor densities was accompanied by similar growth in clock speeds and instruction throughput. Unlike the former, exponential growth in the other two has indeed ended. In 2005, Sutter claimed that this had come to an end the previous year and that manufacturers would instead produce chips with multiple *cores* [Sut05]. He also (correctly) observed that 'concurrency [would be] the next major revolution in how we write software.'[1]

Hence, parallelism is now the only way to extract significant performance gains from Moore's Law. Current CPUs have multiple cores that operate concurrently on *shared memory*. Yet still, many theoreticians and practitioners alike continue to think sequentially. That is because parallelism is *hard* [McK17], both in theory and in practice, and introduces a plethora of

[1] Note that concurrency is a distinct concept from parallelism, but concurrent computations may be executed in parallel by assigning the concurrent processes to separate cores of a CPU or nodes of a cluster computer.

new challenges and pitfalls. Fifteen years after the 'free lunch' (Sutter 2005, Ref. [Sut05]) of exponentially improving sequential performance ended, parallelism is still treated as an extension of (sequential) algorithmics and programming, and not the default. Still, change is afoot: the tooling is improving, new languages are lowering the barrier of entry, and parallelism is gaining prominence in university curricula.

But the story does not end here. With the rapid growth of data sets observed in the last decades, even the most powerful individual parallel machines are insufficient for solving many real-world problems. Increasingly, *distributed computation* is necessary to achieve the required performance. This has multiple reasons. For one, CPUs with multiple cores must provide a coherent view of the shared memory despite concurrent modifications from multiple cores, requiring elaborate cache coherence protocols. With an increasing number of cores, these require increasingly complex interconnect architectures (eg, Intel's *Mesh Interconnect Architecture* [Wikb] or AMD's *Infinity Fabric* [Wika]). These automated general-purpose implicit communication protocols have limited scalability, even though cache coherence across several thousands of cores could still be possible [MHS12]. Additionally, producing ever-larger chips with more and more cores is hard and costly as production yields fall with growing chip sizes. This makes distributed computation the only way to achieve truly scalable performance.

Distributed machines—eg, cluster computers and supercomputers—are typically *shared-nothing* computers, that is, there is no shared memory resource which all processors can access. Rather, communication is *explicit* and occurs over an *interconnection network*. This differs significantly from the *implicit* communication of *shared-memory* machines (usually, that means multi-core CPUs).

On Notation. When we speak of *parallelism*, we refer to both shared-memory and distributed-memory machines. When discussing parallelism with shared memory, we will often abbreviate this as *SM-parallel* for conciseness. Algorithms targeting distributed memory without a particular focus on shared memory are referred to as *distributed algorithms*.

1.1 Defining Communication Efficiency

Distributed computing comes with its own set of challenges and significant complexity. Interconnection networks are expensive and sometimes behave in unpredictable ways. Programming models, like the *Message-Passing Interface (MPI)*, are at the same time considered low-level and not very user-friendly by modern standards and require knowledge of the underlying algorithms and technology to make efficient use of the available resources.

There are multiple sources of complexity in distributed computing. Many of those stem from the fact that the network is typically the most *under-provisioned resource* in a cluster computer. This is inevitable, as the costs of providing dedicated communication links with a fixed bandwidth between any pair of nodes (a *fully-connected* network) would increase quadratically with the size of the network, which is clearly absurd. The network topologies that are used in real-world supercomputers, such as Clos networks [Clo53], Fat-Trees [Lei85], multi-dimensional torus topologies, or Dragonfly networks [Kim+08], aim to provide a good balance between the amount of simultaneous communication and cost. An overview of interconnection network topologies, performance, and pitfalls is given by Hennessy and Patterson [HP11].

Borkar [Bor13] argued in 2013 that in order to be cost-effective, a future exascale supercomputer's communication capabilities would have to be scaled highly sublinearly with regard to the amount of computation happening in the connected subsystems. In a 2015 presentation on Google's data centre networks, Vahdat [Vah15, starting at 17:12 minutes] reported that 'typically—certainly at data centre scale—the resource that is most scarce is the network. We tend to underprovision the network because we don't know how to build big networks that deliver lots of bandwidth'. Additionally, high-performance interconnects account for a significant and growing fraction of the total power consumption of distributed systems, estimated between 12 % [Abt+10] and 27 % [Ber+08, Section 7.3.8].

All this leads us to the central insight that as cluster computers become larger, the cost of local computation is diminishing, while communication becomes more expensive. In this dissertation, we strive to design algorithms that put this observation front and centre. This requires limiting both the number of messages exchanged and the amount of information transmitted by any node in the network. We obtain the following three properties that we want parallel algorithms to fulfil.

Definition 1.1 (Efficient Parallel Algorithms)

We call a parallel algorithm efficient *if the total amount of work performed by all processors is not asymptotically larger than that of the fastest sequential algorithm.*

This is the classical definition of efficiency in parallel algorithms and remains desirable regardless of the amount of communication. Next, we impose a limitation on the amount of information transmitted.

Definition 1.2 (Communication-Efficient Algorithms)

We call a parallel algorithm communication-efficient *if every processor sends and receives a sublinear amount of information, measured both in its local input size and the amount of local work. Formally, let n be the size of the entire input and p be the number of nodes. Each node shall have $\mathcal{O}(n/p)$ elements in its input. Let f be a function bounding the local work performed by the algorithm on a given input size. Then, every processor shall send and receive no more than $o(\min(n/p, f(n/p)))$ machine words of information in total.*

Unless stated otherwise, we will assume that an algorithm's input is distributed arbitrarily but roughly evenly across the processors, ie, out of a total of n input items, every processor has $\mathcal{O}(n/p)$. This is mostly done to simplify the analysis, and our algorithms are correct even if this assumption is violated. However, their running times may suffer as an imbalance in local input sizes will usually carry a corresponding running time penalty. We shall make no assumptions about the *distribution* of the items onto the processors. Assuming a random distribution would require redistributing the entire input in the general case where we do not know the items' distribution, precluding communication efficiency.

A key ingredient to achieving communication efficiency is the *owner computes* paradigm, which states that data should be processed where it is stored, and—conversely—a processor is responsible for all items in its input. Avoiding data movement by moving the computation to the data—and not the other way around—helps in our quest for communication efficiency.

Note that the term *communication efficiency* is also used in other research communities, with definitions adapted to the respective problem domains. Prominently, communication efficiency has become a focus of machine learning research, see, eg, Refs. [Sur+17; Elg+20].

Lastly, we also want the algorithms to be able to scale to extremely large machines. Limiting the amount of information transmitted alone is insufficient, as the number of messages sent and received cannot be allowed to grow arbitrarily.

Definition 1.3 (Highly Scalable Algorithms)
We call a parallel algorithm highly scalable *if every processor is involved in at most a polylogarithmic number of message exchanges, measured in the input size and the number of processors. Formally, let n be the input size and p the number of processors. Then, for some positive constant c, no processor shall send and receive more than $\mathcal{O}((\log(np))^c)$ messages.*

While the title of this dissertation emphasises communication efficiency, we design our algorithms to have all three of these properties wherever possible. The reason for emphasising communication efficiency over parallel efficiency and polylogarithmic latency is that the latter goals are much more established in the study of parallel algorithms. Indeed, they are in some ways the 'default' goals in the field (see, eg, Ref. [JáJ11] and the definition of the complexity class NC).

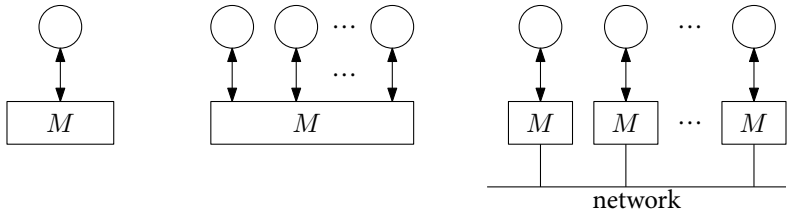
1.2 Machine Models

In this section, we describe the machine models used in the following chapters. We begin by refreshing the classical *Random Access Machine* (RAM) model, on which the other models build, and the *Parallel* Random Access Machine (PRAM) family, before introducing our models for distributed computing.

1.2.1 The Word Random Access Machine (Word RAM)

The word RAM [FW90] is an extension of von Neumann's [vNeu45] Random Access Machine (Figure 1.1 (a)) with a w-bit word size. Input items and their identifiers (pointers) are assumed to fit into a constant number of such machine words. In addition to the features of the RAM, such as indirect addressing, unit-time load and store operations, and unit-time elementary operations on the contents of registers, the word RAM also permits bitwise Boolean operations in unit time. This models the instructions available on real-world computers more closely. The key restriction is that all operations are performed on individual machine words to prevent abuses of the RAM model's unit-cost assumption through excessive operand lengths [FW90].

However, the unit-cost assumption does not model real-world computers' performance well (any more). Due to cache hierarchies, the speed of memory accesses can vary by a factor of around 100 [Bin18a]. Pipelining necessitates complex branch predictors, and mispredictions cause a significant delay, on the order of tens of wasted cycles, which can have a large impact on applications (see, eg, Refs [KS06; EW19] for the impact on sorting algorithms). A load from an unpredictable address in memory can be 100 times more expensive than a predictable load.



(a) RAM model (b) PRAM model (c) Distributed models

Figure 1.1: Illustrations of RAM, PRAM, and distributed machine models.

Moreover, the speed with which a machine can execute instructions on the data far exceeds its ability to fetch data from memory even when the access pattern is predictable. Therefore, algorithms need to be designed to account for this to be fast (see, eg, Refs. [LL99; Axt+17] for the example of sorting). Combined with the observation that $\log n \leq 50$ for all but the very largest applications, these variations in access time mean that an $\mathcal{O}(n)$ time algorithm might be no faster in practice than an efficient $\mathcal{O}(n \log n)$ time algorithm, even on large inputs (eg, walking a random cyclic permutation vs. an efficient sorting algorithm).

Nevertheless, the simplicity of the (Word) RAM is appealing and makes it a good basis for other machine models that extend it.

1.2.2 The Parallel Random Access Machine (PRAM)

The parallel random access machine (PRAM, Figure 1.1 (b)) is an early theoretical model for shared memory machines (see, eg, Refs. [JáJ11; JáJ92]). In a PRAM, multiple independent processors access a global shared memory. The processors themselves have the same properties as in the RAM model. Communication between processors is possible only via the shared memory. Access to memory is assumed to be synchronised, leading to a differentiation of the model by how simultaneous read and write operations are handled. Either operation can be allowed *concurrently* or *exclusively*, abbreviated *C* and *E*, respectively. This gives rise to three variants, abbreviated *EREW*, *CREW*, and *CRCW*.[2] Observe that the last variant, CRCW, needs additional specification of how conflicting concurrent write operations are resolved. Commonly, either an *arbitrary* processor's value is written, or the processor with the lowest ID is given *priority*. Note that while the CRCW PRAM is the most powerful, a CRCW PRAM with p processors can be emulated with an $\mathcal{O}(\log p)$ slowdown by an EREW PRAM with p processors (see, eg, Ref. [JáJ11]).

Algorithms in the PRAM model are often analysed with regard to their *work* and *span* (also called *time*), where the *work* is the total number of operations required, and the *span* or *time* is the number of parallel steps needed if an unlimited number of processors is available.

Much like the RAM model is a very simplified view on sequential processors, the PRAM model does not accurately represent real-world shared-memory machines. The perhaps largest inconsistency is caused by the assumption of synchronised operation, which is contrary to

[2] Theoretically, an *ERCW* PRAM could also be defined, but the usefulness of such a machine is doubtful.

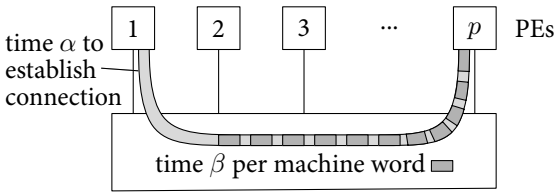


Figure 1.2: Illustration of the parameters of our message passing model.

reality. Additionally, concurrent write accesses on a single memory location cause *contention* in practice. Variants of the model address these concerns, eg, Gibbons et al.'s [GMR98] Queued Read, Queued Write (QRQW) PRAM, but are rarely used.

1.2.3 Our Distributed Machine Model

Summary: single-ported, full-duplex point-to-point communication; p PEs; message startup latency α; communication time per word β; bounds are maximum over all PEs' critical path lengths.

Consider a distributed machine (eg, a cluster computer) with p processing elements (PEs). The PEs are numbered 1 to p and connected by a network so that each PE can communicate with any other PE (Figure 1.1 (c)). Each message has exactly one sender and one receiver (*point-to-point* communication). Several messages may be exchanged simultaneously, with the exception that each PE may only send at most one message *and* receive at most one message at any time (*single-ported, full-duplex* communication). That is, a PE can send and receive simultaneously, but it cannot send to or receive from multiple receivers at the same time. Establishing the connection to another PE is assumed to take time α independent of any notion of distance between the PEs because this duration is dominated by (software) costs incurred on the endpoints rather than the propagation of signals in hardware. Once such a connection has been established, transmitting a machine word takes time β. Refer to Figure 1.2 for an illustration of the model parameters. As a result, sending or receiving a message of size m machine words takes time $\alpha + \beta m$, regardless of which two PEs are communicating. This is very similar to the model of Chan et al. [Cha+07] and practically identical to the model of Sanders et al. [San+19]. Observe that in real-world cluster computers, we have $\alpha \gg \beta \gg 1$ with all three being orders of magnitude apart and continuing to grow apart exponentially [Cou05; Dem19].

We are interested in *bottleneck* running times and communication volumes, ie, the maximum over all PEs of the length of the critical path of the computation. For example, a naïve broadcast of data of size ℓ (where PE 1 sends the data to PE 2, which passes it on to PE 3, etc.) has communication time $(\beta\ell + \alpha)(p - 1)$ even though each PE sends and receives at most a single message of size ℓ. This is because the operation is completed only once the last PE has received the data, which depends on all previous communication steps.

We treat α and β as variables in asymptotic analyses. This allows us to capture internal work x, communication volume y, and latency z separately in a single term: $\mathcal{O}(x + \beta y + \alpha z)$.

Often, all three aspects are important, and combining them into a single expression for running time allows us to specify them concisely.

Recalling Definitions 1.1 to 1.3, let n be the global input size, and let each of the p PEs have no more than $\mathcal{O}(n/p)$ items in its local input. No assumption shall be made about the items' distribution onto the PEs. Then we want algorithms' running time to be expressible as $f(\mathcal{O}(n/p)) + \beta \cdot \mathrm{o}(\min(n/p, f(n/p))) + \alpha \cdot \mathcal{O}(\mathrm{polylog}(np))$, where f describes the algorithm's local processing time and $p \cdot f(\mathcal{O}(n/p)) \subseteq \mathcal{O}(F(n))$ is asymptotically no larger than the running time $F(n)$ of the fastest sequential algorithm.

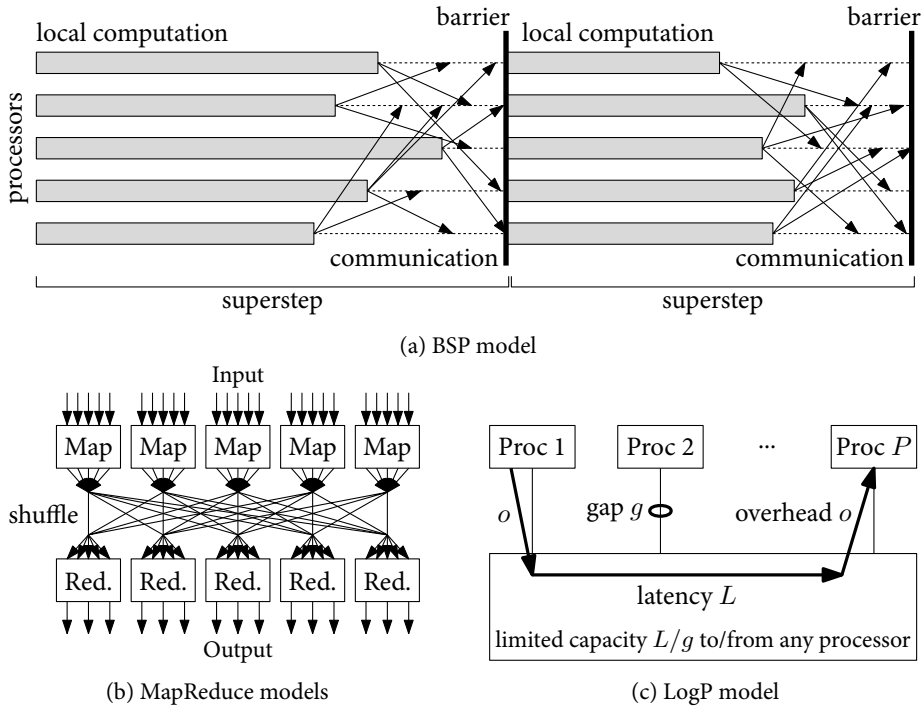
Comparison to Other Machine Models

We introduce the above model because existing models insufficiently capture our requirements. For example, many models only count the total number of messages exchanged, and do not differentiate between one PE sending p messages and p PEs sending one message each, while in our model—and in practice—running times would differ by a factor of p. We now differentiate our model from other parallel models of computation in more detail.

Bulk-Synchronous Parallel (BSP) Model. The BSP model [Val90] can be thought of as an adaptation of the PRAM model, where the shared memory is replaced by local memory and a network, and communication and synchronisation are explicit. Computation proceeds in a series of global *supersteps*, each of which consists of local computations and one round of message exchanges, followed by a barrier synchronisation. This is illustrated in Figure 1.3 (a). A message of size m takes time mg to send, where g ('gap') is a property of the machine, similar to our parameters α and β. The running time of a superstep is the maximum amount of local computation plus g times h, the maximum amount of incoming or outgoing data of any processor, plus time l ('latency') for the barrier synchronisation. The total running time is the sum of all supersteps' running times. Often, the focus is on keeping the number of supersteps low.

Summing up the values of h over all supersteps yields the bottleneck communication volume which we consider in the definition of communication efficiency (Definition 1.2). However, the BSP model allows arbitrarily fine-grained communication at no additional cost, in effect assuming $\alpha = 0$ by not differentiating between sending one message of size m and sending m messages of size 1. This does not model real-world communication networks, where α is typically much larger than β, which is again much larger than the unit cost of a local operation [Cou05; Dem19]. Furthermore, we make heavy use of collective communication operations, which are not directly represented in the BSP model. Moreover, an implementation of the global message exchange with direct delivery, as the BSP model is typically implemented, has startup cost $\mathcal{O}(\alpha p)$ per superstep in our model [Axt+15]. This would cause the actual value of the latency parameter ℓ to be $\ell = \alpha p$ in our model (see also Ref. [Axt+15]), which violates our definition of scalability, Definition 1.3. Furthermore, we are not aware of any concrete results on sublinear communication volume in the BSP model.

MapReduce. A popular early programming model for Big Data applications, MapReduce [DG08] in its basic form takes two functions from the user. The *map* function transforms each input element into zero or more key-value pairs. These pairs are then globally aggregated by



(a) BSP model

(b) MapReduce models

(c) LogP model

Figure 1.3: Illustrations of selected models of distributed computation.

their keys, called the *shuffle* phase. The *reduce* function takes as input a key and the set of all values associated with this key and outputs the result for that key. Figure 1.3 (b) illustrates the data flow in this model. An extension of the model also allows for a *combine* function that locally summarises key-value pairs output by the *map* function before shuffling.

The shuffling phase requires redistribution of large parts of the input, typically via distributed sorting, and often dominates overall running time [Ull12]. In the general case, this requires communication of the entire input. Therefore, MapReduce is not a good fit for expressing communication efficiency.

Massive Parallel Computing (MPC) Model. The MPC model [KSV10; BKS17; GSZ11] (sometimes also called $\mathcal{MRC}$, short for *MapReduce Class*) is another abstraction of MapReduce, Hadoop, and Spark. In it, the system is composed of $\mu \in o(n)$ machines, each of which has $S \in o(n)$ local memory. The computation proceeds in rounds, and in each round, each machine can send and receive messages of total size up to S. Messages are received in the following round. The input is arbitrarily distributed, and the output is provided in a distributed manner. The goal is to construct algorithms that use $\mathcal{O}(\text{polylog}(n))$ rounds.

Our model shares many of the same ideas as the MPC model—arbitrary input distribution, distributed output, fully decentralised computation—but we dislike the focus on the number of rounds, allowing arbitrarily fine-grained communication at no additional cost (*cf.* our criticism

of the BSP model above). Nor does the MPC model address communication volume directly, making it awkward to design algorithms with the goal of communication efficiency in it. Additionally, algorithms that make full use of the provisions of the MPC model are unlikely to perform well in practice, as the model does not consider algorithms' parallel efficiency, and allows nearly quadratic total space usage [San20].

LogP Model. The LogP model [Cul+93; Cul+96] aims to provide a more accurate representation of parallel machines and to close many of the 'loopholes' that allow for surprisingly fast algorithms in the PRAM model which perform poorly when implemented on real-world machines. It is an asynchronous distributed-memory model which assumes a point-to-point message-passing network. A machine is characterised by four parameters that give the model its name, namely the communication delay or *latency* (L), the *overhead* of sending or receiving a message (o), the communication bandwidth (represented by its inverse, the minimum *gap* g between successive messages), and the number of *processors* (P). See Figure 1.3 (c) for an illustration.

We are in complete agreement with the goals of the LogP model. Yet its focus on accurately modelling the behaviour of real-world machines forces algorithm designers to carefully consider all possible combinations of (relative values of) its parameters. This makes the LogP model a suitable platform for studying the most basic operations provided by, eg, a communication library, but obscures the broader picture at higher levels. Indeed, the paper introducing the model demonstrates this well: in its description of how to implement the broadcast operation in the LogP model, it states that 'the optimal broadcast tree for p processors is unbalanced with the fan-out at each node determined by the relative values of L, o, and g' [Cul+93], but does not state how. The actual running time of an optimal broadcast algorithm in the LogP model [Kar+93] is a complicated expression, much more so than what we obtain in our model (see below in Section 1.3.3). For our purposes, the LogP model is therefore much too detailed and provides insufficient abstraction.

CONGEST/LOCAL Models. The $\mathcal{CONGEST}$ and $\mathcal{LOCAL}$ models [Pel00] model the network as a graph of processors (nodes) and communication links (edges), which at the same time is also the input to the problem. Communication is possible only with a node's neighbours in the graph and is usually assumed to be synchronous (though asynchronous versions of $\mathcal{CONGEST}$ exist). In each round, a processor can send and receive a message to/from each of its neighbours and perform an unbounded amount of local computation. The $\mathcal{LOCAL}$ model assumes that nodes have unique IDs that can be represented in $\mathcal{O}(\log p)$ bits and allows messages to be of arbitrary size. In the $\mathcal{CONGEST}$ model, the message size is limited to $\mathcal{O}(\log p)$ bits.

While interesting in theory, the assumptions of synchronicity and the encoding of the problem into the network limit the applicability of these models in practice, and it is not immediately clear how algorithms formulated in these models can be implemented to run efficiently on real-world machines.

Resource-Oblivious Models. In resource-oblivious models, algorithms are specified without any mention of machine parameters such as cache and block size [Fri+12] *(cache-oblivious)* or number of PEs [Bil+16] *(network-oblivious)*. These models require loading data items before

they can be processed. Therefore, reading the entire input even once is the equivalent of communicating the entire local input in our model. As a result, these models are not suitable for adaptation to communication-efficient algorithms, and, in fact, cannot even be used to express communication efficiency.

Parallel Memory Hierarchy Models. Similarly, models such as the *parallel external memory model* [Arg+08] and the *parallel hierarchical memory model* [JW94] also cannot be used to express communication-efficiency, as attaining sublinear communication volume is impossible in models that require loading (ie, communication) of the data to allow processing.

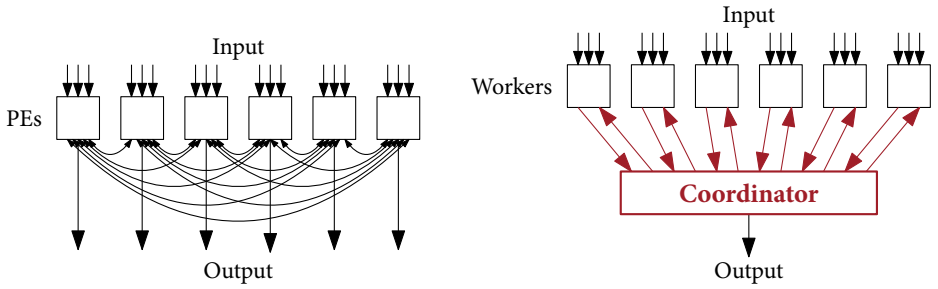
Communication-Avoiding Algorithms. Though closely related in name and also striving to minimise communication volume as well as latency, *communication-avoiding algorithms*, eg, Refs. [Chr+13; DY14], perform data-oblivious calculations, often in a linear algebra and—more recently—machine learning context. They apply to 'code that looks like nested loops accessing arrays' [Dem19], or more precisely, nested loops with array accesses following a data-independent affine pattern. Often, there are lower bounds for these problems that preclude algorithms meeting our definition of communication-efficiency. Moreover, the bounds proven in these works are most relevant for computations with work superlinear in the input size. We go into the opposite direction, looking at problems with linear or even sublinear work.

1.2.4 Mini-Batch Model

In some cases, the input may be too large to store it even in a distributed manner. However, we may not need to: *streaming algorithms* process the data as it arrives, using it to update their (small) internal state without storing the entire input. For a long time, *sequential* streaming algorithms were the main focus of research in this area; refer to Ref. [Mut05] for a survey on results in this field. But the same motivation that led us to consider parallel and distributed single-shot algorithms also applies to data streams. In the last decade, such *distributed streaming models* have gained some popularity, and the *continuous monitoring model* is the most popular choice in the algorithms community. However, as we state below, this model is full of unrealistic assumptions that make it unsuitable for the design of communication-efficient algorithms. Therefore, we use a *mini-batch model* instead, similar to models that are already widely used in practice [Zah+13] and more rarely also in theory [TT19].

Instead of processing each item individually, we process items as a series of *mini-batches* which arrive on small time intervals and in a distributed fashion: some of their items arrive at each processing element (PE). They might, for example, be delivered over the network or be read in blocks from a file system. Apache Spark Streaming, where this is called the *discretized streams model* or sometimes *micro-batch model*, defines them as the set of all items that arrived within a certain time window since the previous batch finished [Zah+13]. Because memory is limited, the algorithm is allowed to keep some local state of limited size, but items of past mini-batches cannot be accessed any longer. Only the items of the current mini-batch are available in memory at each point in time.

This is a natural generalisation of multiple other models of streaming algorithms. On a sequential machine with batches of size 1, we obtain the sequential streaming model (see, eg, [TT19]). In a distributed model with p sites (nodes) which exchange fixed-size messages



(a) Our mini-batch model

(b) Continuous monitoring model

Figure 1.4: Comparison of data flow in our mini-batch model and the continuous monitoring model. Communication bottlenecks highlighted in red.

with a coordinator, batches of a single item per site yield the continuous monitoring model described below. In this dissertation, we use the distributed model of Section 1.2.3 as the underlying machine model. An illustration of data flow in this combination of models is shown in Figure 1.4 (a). We use the terms *batch* and *mini-batch* interchangeably.

Unless explicitly specified, we shall make no assumptions about the distribution of mini-batch sizes across PEs or over time, nor about the distribution of items. In algorithm analysis, we denote by b the maximum number of items in the current batch at any PE. Thus, an algorithm expressed in the mini-batch model can handle arbitrarily imbalanced inputs without any impact on correctness; however, load balance may suffer if the number of items per PE differs widely.

Note that the machine learning community uses the term 'mini-batch' in the context of stochastic optimisation (eg, Ref. [Dek+12]), where the input is partitioned into mini-batches which are then used for training in a randomised way. However, the term is also used outside the context of streaming algorithms ('online learning') in machine learning literature. Therefore, these uses of the term are not directly comparable.

Comparison to Other Models

As before, we compare our model to others that are popular in the literature.

Continuous Monitoring Model (CMM). In the *continuous distributed streaming model* or *continuous monitoring model* [CMY11; Cor13], one of the PEs is designated the *coordinator* and the others are called *players* or *sites*. The sites each receive a stream of items and can communicate only with the coordinator, whose task is to *continuously monitor* some function over the union of all items seen so far at any site. The goal is to minimise the number of bits communicated between the sites and the coordinator. An illustration of data flow in this model is shown in Figure 1.4 (b).

The model does not consider the number of messages over which these bits are spread out, in effect assuming $\alpha = 0$ (*cf.* our criticism of the BSP model). Additionally, *all communication* is between one of the sites and the coordinator, meaning that the bottleneck communication volume is equal to the total communication volume. This severely limits scalability as the

scalability of algorithms in the model is inherently limited by the load on the coordinator. Since most algorithms in the model communicate $\Omega(p)$ bits in total, they do not typically fulfil our criteria for communication efficiency. Additionally, the assumption of per-item synchronisation of the processors as well as synchronous operation of the network is in stark contrast to the guarantees of real-world networks, where the cost of synchronisation is high. Hence, the huge number of synchronisation operations required would be prohibitively expensive.

Massive Unordered Distributed (MUD) Computation. Mud algorithms [Fel+10] are a model for MapReduce-style computations which combine MapReduce with streaming computations. The reducers operate on a stream, with no random access to other items, and are limited to polylogarithmic space. Only a single pass over the data is allowed. Feldman et al. [Fel+10] show that every symmetric (ie, order invariant) deterministic streaming algorithm that is defined on all inputs can be simulated by a mud algorithm in small space, although with exponential time overhead. As with other MapReduce models, communication volume is not addressed directly, limiting its usability for our purposes.

1.3 General Preliminaries

A brief overview of the notation that we introduced up to this point is shown in Table 1.1.

When we give time or space bounds in writing, we will sometimes omit in reference to which quantity the bound is given if it is clear from the context. Such bounds are always upper bounds. For example, *logarithmic time* usually means $\mathcal{O}(\log n)$ for input size n, and *logarithmic latency* is time $\mathcal{O}(\alpha \log p)$.

1.3.1 Definitions

We generally use 1-based array indexing: the elements of an array a of size n are $a[1]$ to $a[n]$ or, equivalently, a_1 to a_n. When we wish to refer to a range $\{a, \dots, b\}$, we often use the notation $[a\,..\,b]$. Similarly, a slice out of a sequence s is referred to as $s[a\,..\,b] := \langle s[a], \dots, s[b] \rangle$. For closed real-valued intervals, we use the standard notation $[a, b] = \{x \in \mathbb{R} \mid a \leq x \leq b\}$, the open interval $(a, b) = \{x \in \mathbb{R} \mid a < x < b\}$, and the half-open intervals $(a, b]$ and $[a, b)$ are defined analogously.

A call to rand() returns a uniformly distributed random value in $[0, 1)$ in constant time.

Throughout this dissertation, logarithms with an unspecified base are binary logarithms: $\log x := \log_2 x$. When we mean the natural logarithm, we write $\ln x$. Powers of logarithms are written as $\log^c x := (\log x)^c$.

1.3.2 Pseudocode

We present some of our algorithms using high-level pseudocode in a *single program multiple data* (SPMD) style. This means that the same program runs on each PE, and operates on local data, with all variables being local. Communication happens predominantly through collective operations. We can refer to data on remote PEs using the notation $x@i$, which refers to the value of variable x on PE i. Such an access implies communication. For example, $\sum_i x@i$ denotes the

Table 1.1: Summary of notation used throughout this dissertation.

Symbol	Meaning
n	input size
p	number of PEs
α	communication overhead per message
β	communication cost per machine word
w	machine word size
b	batch size per PE (mini-batch model)

global sum of x over all PEs, which can be computed using a sum (all-)reduction, see below. In general, our pseudocode is very similar to that of Sanders et al. [San+19].

Our pseudocodes are optimised for clarity. As a result, they do not necessarily reflect our actual implementations, which often comprise additional optimisations, details of which are given as *Implementation Details* in the associated chapter's evaluation section.

1.3.3 Collective Communication

Our algorithms communicate primarily through collective operations, which operate on sequences of one or more objects. In the following, let $k \in \mathbb{N}$ be the size of the message, measured in machine words. A *broadcast* distributes a message from one PE to all others. A *reduction* applies an associative operation (eg, sum, maximum, or minimum) to a sequence of objects, returning the result at a designated PE (without loss of generality, this is usually the first PE). An *all-reduction* makes the result of the reduction available at all PEs and is conceptually equal to a reduction followed by a broadcast. A *prefix-sum* or *scan* computes $\sum_{i=1}^{j} x@i$ on PE j, where x is a sequence of length k. The *scatter* operation distributes k items onto a set of $y \leq p$ PEs such that each PE gets a piece of size k/y. Symmetrically, a *gather* operation collects pieces of size k/y from $y \leq p$ PEs on a single PE. All of these operations can be performed in time $T_{coll}(k) \in \mathcal{O}(\beta k + \alpha \log p)$ [Bal+95; SST09].

In an *all-to-all personalised communication* (*all-to-all* for short) each PE sends one message of size k to every other PE. This can be done in time $T_{all\text{-}to\text{-}all}(k) \in \mathcal{O}(\beta k p + \alpha p)$ using direct point-to-point delivery or in time $\mathcal{O}(\beta k p \log p + \alpha \log p)$ using hypercube indirect delivery [Lei92, Theorem 3.24]. The *all-to-all broadcast* (aka *all-gather* or *gossiping* operation) collects a message of size k from each PE and delivers all messages to all PEs. This operation can be implemented with running time $\mathcal{O}(\beta k p + \alpha \log p)$ [Kum+94].

For concrete algorithms that implement these collective operations efficiently, we refer the reader to Chapter 13 of Ref. [San+19].

1.3.4 Search Trees

Search trees can be used to represent a sorted sequence of objects such that a number of operations on that sequence can be performed in logarithmic time. Among these operations

is *inserting* or *removing* objects as well as *searching* for the next-largest objects given a key. Sometimes we will also require a *split* operation that given a key x splits a search tree T into two search trees T_1 and T_2 such that T_1 contains all objects of T whose keys are at most x and T_2 contains the objects of T whose keys are larger than x. The reverse operation, joining two search trees T_1 and T_2 where all keys occurring in T_1 are smaller than the smallest key of any object in T_2 into a combined tree T, is called *join*. Search trees can also support the *select* operation, which given a rank i returns the i-th smallest object in the tree, and *rank*, which given an object x returns how many objects' keys are at most as large as that of x.

A B+ tree (see, eg, Ref. [San+19, Chapter 7]) is a search tree that can support all the above operations in logarithmic time. Although based on B-trees [BM72], the inner nodes of a B+ tree only store keys. All objects (ie, key-value pairs) are stored in the leaf nodes. Like with B-trees, the nodes have an arbitrary but fixed maximum degree d, and—except for the root—every node is at least half full at any point in time, ie, each inner node has at least $\lceil d/2 \rceil$ children and each leaf node stores at least $\lceil d/2 \rceil$ objects. By maintaining a linked list between the leaf nodes, advancing to the next-larger or -smaller item takes only constant time. Like B-trees, the join and split operations can be supported in logarithmic time (see, eg, Ref. [San+19, Chapter 7.3.2]). If we additionally keep track of subtree sizes, rank and select operations can be handled in logarithmic time as well (eg, Ref. [San+19, Chapter 7.5.2]).

1.3.5 Concentration Inequalities

An event is said to occur *with high probability (w.h.p.)* if its probability of occurring can be made arbitrarily close to 1 by increasing a certain number (which is usually clear from context). An example of this would be an algorithm that is correct with probability $1 - 1/n$ for an input size of n items.

Chernoff Bounds. We use the following multiplicative Chernoff bounds [San+19; MU05] to bound the probability that a sum $X = X_1 + \ldots + X_n$ of n independent random variables taking values in $\{0, 1\}$ deviates substantially from its expected value $\mathbf{E}[X]$. For $0 < \varphi < 1$, we have

$$\mathbf{P}[X < (1 - \varphi) \cdot \mathbf{E}[X]] \leq \exp\left(-\frac{\varphi^2}{2}\mathbf{E}[X]\right) \tag{1.1}$$

$$\mathbf{P}[X > (1 + \varphi) \cdot \mathbf{E}[X]] \leq \exp\left(-\frac{\varphi^2}{3}\mathbf{E}[X]\right) . \tag{1.2}$$

Hoeffding's Inequality. A more general bound is Hoeffding's inequality. Let $X_1, \ldots, X_n$ be n independent random variables, where X_i is strictly bounded by the interval $[a_i, b_i]$, and let $X := X_1 + \ldots + X_n$ be their sum. Then, for any $t > 0$, by Ref. [Hoe63, Theorem 2] we have

$$\mathbf{P}[|X - \mathbf{E}[X]| \geq t] \leq 2 \exp\left(-\frac{2t^2}{\sum_{i=1}^{n} (b_i - a_i)^2}\right) . \tag{1.3}$$

If the a_i and b_i are all equal, respectively, and $c := b_1 - a_1$, then this simplifies to $2\exp\left(-\frac{2t^2}{nc^2}\right)$.

1.4 Communication Efficiency in Big Data Challenges

'Big Data' is at the same time central to the continued progression of the information revolution, and a much over-used marketing term. At the height of its hype phase, you could find the phrase on airport billboards, advertising large companies' supposed trend leadership and grandiloquently asserting their software solutions' superior capabilities. A vast number of laypeople have heard the phrase 'Big Data', yet even experts struggle to precisely define it. Instead of a precise definition, it is often loosely characterised by three 'V's: (i) *Volume* (the quantity of data) (ii) *Velocity* (the speed with which the data arrives or needs to be processed), and (iii) *Variety* (the kind of data) [KM16]. Clearly, coping with high data volume and velocity is directly related to the scalability of a system—which is sometimes also considered a characteristic of big data—and fall under the purview of algorithmicists designing the computational foundation of a big data processing system. As we have argued above, handling ever-larger data while maintaining fast turn-around times—or achieving them in the first place—is only possible by leveraging distributed computing. As a result, a variety of big data *frameworks* have been developed and proved enormously popular: MapReduce [DG08] and Apache Hadoop, its most popular implementation, marked the point where big data processing became accessible to anyone without requiring specialist knowledge. Later, Apache Spark [Zah+10] was developed to alleviate the strain of limitations in the MapReduce paradigm and has since surpassed it in popularity.[3] Another contender similar to Apache Spark is Thrill [Bin+16], which we use as the platform for implementing our checkers in Chapter 4. For high-velocity (streaming) applications, Apache Flink [Ale+14], Apache Spark Streaming [Zah+13], and Apache Storm are widely used in industry, see, eg, Ref. [Tos+14] for an account of how Storm is (or was at the time) used at Twitter. Both Flink and Spark Streaming show good performance, and, despite the latter using a batched computation model (see also Section 1.2.4), low latency [Kar+18].

Historically, MPI dominated the world of distributed computing. Its origins are in High-Performance Computing (HPC), and its interface is designed to permit efficient implementations of arbitrary algorithms. It is, therefore, an abstraction of the hardware, providing only the fundamentals, such as collective operations (see Section 1.3.3), but not higher-level algorithms. This makes it extremely general and highly efficient, yet also hard to use. For distributed computing to gain popularity outside the performance-oriented world of HPC, a simpler environment was needed. This is what MapReduce provided, and accordingly, it took the world of practitioners by storm. Its map–shuffle–reduce cycle can be adapted to a wide range of tasks while abstracting away the tiring details and pitfalls of the systems that preceded it. However, it is slow and inefficient in its generality. The observation that many applications do not actually require a full reshuffling led to the development of the next generation of big data frameworks, with Apache Spark as the most popular among them. Their interfaces are still easy to use for the programmer, maybe even more so than MapReduce due to a larger standard library of operations. But in their implementations, more specialised algorithms than all-to-all shuffles are used to implement the operations, avoiding unnecessary data exchanges (a recent example of this that has not yet been integrated into any of the popular frameworks is communication-efficient

[3] As measured by Google search trends: `https://trends.google.com/trends/explore?date=all&q=/m/0ndhxqz,/m/0fdjtq` (accessed 18th August 2020).

sorting of strings and other non-scalar data types [BSS20]). Meanwhile, these frameworks still handle (static) load balancing and fault tolerance automatically, letting the user focus on solving their problems.

It is in this space that this dissertation operates, pushing operations that are used in the algorithmic layers of these frameworks towards communication efficiency. The ultimate goal of such work is to allow these systems to scale to ever-larger clusters to keep up with the exponential growth of data set sizes. It is clear that algorithms in this space need to behave robustly with respect to a range of parameters outside our control, such as input distribution. Naturally, the systems' scalability and performance is a major concern. Therefore, our focus is on designing communication-efficient algorithms that perform well not just in theory, but also in practice (*cf.* the methodology of *Algorithm Engineering*, eg, Refs. [San09; San10]). In this dissertation, we study several elementary operations offered by or used internally in such big data systems, for which we present highly scalable communication-efficient algorithms.

1.5 Contributions

This dissertation consists of three parts. The first part focuses on various kinds of selection algorithms, the second part on weighted sampling in different forms, and the third part on probabilistic checkers for common operations in big data frameworks. Table 1.2 lists our main contributions.

Chapter 2 addresses selection problems and fittingly begins with perhaps the most common of them: the classical selection problem in Section 2.3, where the goal is to identify the element with rank k in the union of all machines' inputs. First, we consider the case where the input is arbitrarily distributed across the machines, except for our general assumption that each of the p processors shall have $\mathcal{O}(n/p)$ of the n input items. There, we show that with a small adaptation to the process of choosing the pivots, the problem can be solved efficiently without previous works' assumption of randomly distributed inputs. Our algorithm requires time $\mathcal{O}(n/p + \beta \min(\sqrt{p} \log_p n, n/p) + \alpha \log p)$ with high probability. An experimental evaluation in Section 2.7.1 shows that this algorithm scales well. Next, we show that the problem of selection from locally sorted sequences, also known as the multisequence selection problem, can be solved much more efficiently if the output rank is allowed to vary by a constant factor. For this case, we obtain an expected running time of $\mathcal{O}(\log k + \alpha \log p)$. When the output rank is fixed, the expected running time is $\mathcal{O}\big(\log^2 k + \alpha \log k \log p\big)$. In Section 2.4, we use these algorithms to construct a communication-efficient bulk priority queue. Later, in Section 3.7, we apply this to weighted reservoir sampling where our algorithm shows far better scaling than a centralised approach. The third pillar of the selection chapter is frequent item selection, ie, identifying the k objects that occur most often in the input. In Section 2.5, we give several options for different error models, including a *probably approximately correct* and a *probably exactly correct* algorithm. For the former case, we obtain expected running time $\mathcal{O}(n/p + \beta \cdot 1/\varepsilon \cdot \sqrt{\log p \cdot \log(n/\delta)/p} + \alpha \log n)$ to compute an (ε, δ)-approximation, ie, one that is accurate to within relative error ε with probability at least $1 - \delta$. Experiments, presented in Section 2.7.2, show that our fully distributed algorithms clearly outperform more centralised ones. A generalised version of the problem, identifying the k objects with the highest sums of

associated values, is considered in Section 2.6. Our algorithm's running time is only a $\sqrt{\log p}$ factor in communication volume higher than our most frequent objects algorithm.

Chapter 3 considers a multitude of weighted sampling problems. First, in Section 3.3, we revisit construction of a classical data structure for sampling from a categorical distribution: *alias tables*. We begin by solving 'an easy exercise to the reader', the details of which 'are all straightforward'[4] (Vose 1991, Ref. [Vos91]), eliminating the linear memory overhead of the fastest known construction algorithm, in Section 3.3.1. The result is a linear-time alias table construction algorithm that requires only a constant amount of additional space. We then parallelise this algorithm for shared memory in Section 3.3.2 by showing how its state at any point in time can be quickly computed in advance, which allows us to cut a problem instance into parts of equal size to utilise all processors. This algorithm has linear work and logarithmic span in the PRAM model. Our experiments (Section 3.8.2) show that it scales well until the memory bandwidth of the machine is saturated. We also present a variant where each processor greedily assigns items from its input until it gets stuck, switching to our full parallel algorithm only for the remaining items. This modification successfully keeps overhead for the parallel algorithm low and shows excellent scaling behaviour on non-adversarial inputs. Lastly, we show how the algorithm's memory overhead can be reduced to a sublinear term by using coarser prefix sums in the splitting procedure. More precisely, we show how $\Theta(n/\log n)$ auxiliary space can be achieved without an increase in asymptotic work or span. Section 3.3.3 shows how the problem can be solved efficiently on distributed-memory machines using a two-level approach. Constructing this data structure takes time $\mathcal{O}(n/p + \alpha \log p)$ and the time to sample a single item is $\mathcal{O}(\alpha)$. Replicating the top-level data structure simplifies the query process at a cost of $\mathcal{O}(\beta p)$ additional preprocessing time. This variant can also be used on shared-memory machines—using any sequential alias table construction algorithm at the bottom level—to achieve extremely fast preprocessing, though this comes at the cost of 50 % slower queries due to the additional indirection (Section 3.8.3). It is also of note that this method scales far better when using our new sequential alias table construction algorithm as its base case because the memory overhead of Vose's classical construction algorithm limits scalability.

The second main contribution in Chapter 3 is our *output-sensitive* algorithm for weighted sampling with replacement. Output-sensitive here means that its running time depends only on the number of unique items in the sample, in which it is linear, and not the number of samples drawn. These may be an exponential factor apart if the input distribution is very skewed (eg, a power-law distribution). To the best of our knowledge, the algorithm we give in Section 3.4 is the first parallel as well as the first *sequential* algorithm for weighted sampling with replacement with this property. It is based on integer sorting and reduction trees. Computing the necessary data structure takes linear work and logarithmic span on top of (parallel) integer sorting of the input into u buckets, where u is the logarithm of the quotient between the largest and the smallest weight in the input. This is generally possible with linear work and logarithmic span on many machine models. Sampling then has near-optimal work $\mathcal{O}(s + \log n)$ and span $\mathcal{O}(\log n)$ on a CREW PRAM, where s is the number of unique items in the output. In a distributed setting with randomly allocated items, construction takes time $\mathcal{O}(n/p + \alpha \log p)$ and queries $\mathcal{O}(s/p + \log p)$.

[4] That no solution to this exercise has been published or implemented to date to the best of our knowledge gives us the impression that the exercise might not have been as easy to others as it was to Mr Vose.

Experiments in Section 3.8 show good scalability, although the speedups for data structure construction are limited by the available memory bandwidth. Nonetheless, queries—while significantly more complex than for alias tables—benefit from the data structure's memory locality, and consistently outperform the two-level method on all inputs. On inputs that yield many duplicate samples, the algorithm is much faster than sampling from an alias table. A slightly relaxed version of the algorithm outperforms alias tables in query speed even when the input weights are uniformly random, making it the consistently fastest algorithm for sampling with replacement.

In Section 3.5, we construct a truthful estimator for the number of unique items in a sample with replacement, which we then apply 'in reverse' to determine how large the sample should be so that it contains at least k unique items with sufficient probability. This allows us to efficiently sample *without* replacement using the above algorithm for sampling *with* replacement and some post-processing. We achieve expected work $\mathcal{O}(k + \log n)$ and span $\mathcal{O}(\log n)$ for sampling on a CREW PRAM. In a distributed setting with randomly allocated items, the expected query time is $\mathcal{O}(k/p + \alpha \log n \log p)$. The running time of construction is the same as for sampling with replacement in both cases.

Poisson sampling is a generalisation of Bernoulli sampling to weighted items, where each item's weight specifies its probability of being included in the sample. Our algorithm for Poisson sampling, described in Section 3.6, is in effect another application of integer sorting with suitably constructed keys. We achieve linear work and logarithmic span for preprocessing. The work for sampling is linear in the sample size, and its span is logarithmic as well. In a distributed setting, the algorithm is communication-free.

Last among our algorithms for weighted sampling, Section 3.7 considers the problem of sampling without replacement from distributed data streams, presented as a series of *mini-batches* of new items. Using the bulk priority queue from the selection chapter, augmented search trees, and an adaptation of an existing sequential algorithm, we describe an efficient algorithm. Processing a batch consisting of up to b items per processor takes expected time $\mathcal{O}((b+1)\log(b+k) + \alpha \log k \log p)$, although the latency can be reduced to $\alpha \log p$ if the sample size is allowed to vary. Our evaluation (Section 3.8.4) shows that this fully decentralised algorithm handily beats a centralised version in almost all cases. The main contribution here is to demonstrate that fully decentralised streaming algorithms are possible and can be much more efficient than what is achievable with the centralised models that are almost universally used in the distributed streaming literature.

In Chapter 4 we turn to the problem of checking the correctness of big data frameworks' basic operations at runtime. The goal is to detect incorrect results caused by software bugs as well as errors introduced through hardware failures, such as cosmic rays triggering bit flips in the CPU or memory, or defective memory locations. We show that a large number of operations can be checked using reductions to checkers for two central problems: permutation and (sum) aggregation. While many (folklore) results exist for the former problem, we formally prove their accuracy and determine their communication cost in Section 4.4.

The aggregation checker presented in Section 4.3 is a novel application of the data structure underpinning counting Bloom filters and count-min sketches. By reducing the key space dramatically through hashing and aggregating the result modulo a randomly chosen number, we obtain a small fingerprint of the data. These fingerprints are designed to be invariant

under aggregation for certain classes of operations such as sum reductions. Thus, by applying this procedure to both the input and the output, we can compare their fingerprints and be fairly certain that the aggregation was performed correctly if the fingerprints match. A theoretical analysis of the false positive probability of our checker promises excellent accuracy with very low communication cost: a false positive probability below 10^{-20} can be achieved with a message size of only 960 bits, ie, 15 machine words on a 64-bit machine.[5] We obtain running times of $\mathcal{O}((n/p + \beta)\log(1/\delta) + \alpha \log p)$ for aggregation checking and $\mathcal{O}((n/(pw) + \beta)\log(1/\delta) + \alpha \log p)$ for our permutation checker, where w is the machine's word size. In both cases, δ is the maximum allowed false positive rate, ie, an incorrect result is detected with probability at least $1 - \delta$.

Scaling experiments with an implementation in Thrill, a big data processing framework, in Section 4.6 show that as a result of our careful engineering, our checkers have negligible overhead in practice. An extensive experimental evaluation with deliberately introduced errors in low-accuracy configurations further shows that both of our checkers achieve or even exceed the claimed accuracy. They achieve this despite (and in some cases likely because of) the non-ideal behaviour of the real-world hash functions used in place of the random hash functions assumed in the analysis.

[5] That is roughly half the space occupied by an ASCII encoding of the sentence preceding this footnote.

Table 1.2: Our main contributions. Parameters (see also Table 1.1): input size n; output size or rank k; unique items in output s, number of PEs p, machine word size w, logarithm of maximum input weight span $u = \log(w_{\max}/w_{\min})$, here assuming $u \le n/p$. Parallel sorting of n integers from $[0\,..\,u]$ takes time $\text{isort}_u(n)$. Error bounds: relative error $\le \varepsilon$ with failure probability $\le \delta$.

Problem	**Section**	**(Expected) Running Time, $\mathcal{O}(\cdot)$**
unsorted selection	2.3.1	$\frac{n}{p} + \beta \min\left(\sqrt{p}\frac{\log n}{\log p}, \frac{n}{p}\right) + \alpha \log p$
sorted selection, bulk PQ insert* + deleteMin*	2.3.2, 2.4	$\log k + \alpha \log p$ (flexible k), $\log^2 k + \alpha \log k \log p$ (fixed k)
most frequent objects	2.5	$\frac{n}{p} + \beta \frac{1}{\varepsilon}\sqrt{\frac{\log p}{p} \log \frac{n}{\delta}} + \alpha \log n$
top-k sum aggregation	2.6	$\frac{n}{p} + \beta \frac{\log p}{\varepsilon}\sqrt{\frac{1}{p} \log \frac{n}{\delta}} + \alpha \log n$
alias table construction (PRAM)	3.3.2	work $\mathcal{O}(n)$, span $\mathcal{O}(\log n)$
2-level alias table (distributed)	3.3.3	construction $\mathcal{O}\left(\frac{n}{p} + \alpha \log p\right)$, query $\mathcal{O}(\alpha)$
sampling w&w/o replacement, precomputation (PRAM)	3.4, 3.5	work $\mathcal{O}(n + \text{isort}_u(n))$, span $\mathcal{O}(\log n + \text{isort}_u(n))$
sampling w&w/o replacement, precomputation (distributed)	3.4.1, 3.5.1	$\frac{n}{p} + \alpha \log p$
sampling w&w/o replacement, query (PRAM)	3.4, 3.5	work $\mathcal{O}(s + \log n)$ & $\mathcal{O}(k + \log n)$, resp., span $\mathcal{O}(\log n)$
sampling with replacement, query (distributed)	3.4.1	$\frac{s}{p} + \log p$ (randomly distributed items)*
sampling without replacement, query (distributed)	3.5.1	$\frac{k}{p} + \alpha \log^2 n$ (randomly distributed items)*
reservoir sampling, mini-batch of $\le b$ items per PE	3.7	$(b+1)\log(b+k) + \alpha \log k \log p$
sum/count aggregation checking (average, median)	4.3	$\left(\frac{n}{p} + \beta\right) \log \frac{1}{\delta} + \alpha \log p$
permutation checking (sort, union, merge, zip, …)	4.4	$\left(\frac{n}{pw} + \beta\right) \log \frac{1}{\delta} + \alpha \log p$

* This only serves to make bounds easily presentable here, the algorithm makes no such assumption.

Chapter 2

2 Selection Problems

We present scalable parallel algorithms with sublinear per-processor communication volume and low latency for several fundamental problems related to finding the most relevant elements in a set, for various notions of relevance. We begin with the classical selection problem with unsorted input, and subsequently present generalisations with locally sorted inputs and dynamic content (bulk-parallel priority queues). Then we move on to finding frequent objects and top-k sum aggregation.

Motivation. Selection problems are among the most fundamental algorithmic problems. Finding the median and other quantiles is a basic problem in statistics. As we will see later, finding the element of rank k from the union of p sorted sequences is ideally suited for implementing distributed priority queues, which have numerous applications themselves, such as (best-first) branch-and-bound methods (see, eg, Refs. [KZ93; San98]). In data mining, we often have to find the globally most frequent items from a large distributed data set, or those whose associated values add up to the highest sums. Common to all of these problems is that they have a large input and a relatively small output. More precisely, we consider problems that ask for the k most 'relevant' results from a large set of possibilities and devise communication-efficient algorithms for them.

References. This chapter is based on a conference paper [HS16] published jointly with Peter Sanders. The sections included in this dissertation are mainly the work of the author of this dissertation. Sections of the paper on multicriteria top-k and item redistribution that are largely due to Peter Sanders are not included in this dissertation. The analyses in Sections 2.3.1 and 2.3.2.b) are originally due to Peter Sanders and were significantly extended by the author of this dissertation for inclusion in the present work. Large parts of this chapter were copied verbatim from the paper and the corresponding technical report [HSM15], which contains additional proofs that had to be omitted in the conference version. The author of this dissertation is also the author of all implementations and conducted all evaluations.

Overview. In the simplest case, our input consists of totally ordered elements, and we ask for the k smallest of them—the classical selection problem. Several variants of this problem are studied in Section 2.3. For the classical variant with unsorted inputs, a careful analysis of a known algorithm [San98] shows that the previously assumed random allocation of the inputs is not actually necessary, and we get running time $\mathcal{O}(n/p + \log p)$. For locally sorted inputs, we obtain a latency of $\mathcal{O}(\log^2 kp)$ communication startups, ie, messages. Interestingly, we can return to $\log p$ latency if we are willing to relax the output size k by a constant factor. This uses a new technique for obtaining a pivot element with a given rank that is much simpler

Table 2.1: Summary of notation used in this chapter.

Symbol	Meaning
M, n	input multiset and its size $n := \|M\|$
k	rank of desired output item
ε	relative error bound
δ	maximum permitted failure probability
ρ	sampling rate (Bernoulli sampling)
p	number of PEs
α, β	communication characteristics of the machine model, see Section 1.2.3

than the previously proposed techniques based on sorting. When used with exact bounds, this algorithm has latency $\mathcal{O}(\log k \log p)$.

Bulk-parallel priority queues are a data structure generalisation of the selection problem. Previous parallel priority queues are not communication efficient in the sense that they move the elements around, either by sorting [DP92] or by random allocation [San98]. Section 2.4 generalises the results on selection from Section 2.3. The key to making this work is to use an appropriately augmented search tree data structure that efficiently supports insertion, splitting, and the operations required by the parallel selection algorithm.

A fundamental problem in data mining is finding the most frequently occurring objects. This is challenging in a distributed setting since the globally most frequent elements do not have to be locally frequent in any particular PE's local input. In Section 2.5, we develop very fast sampling-based algorithms that find an (ε, δ)-approximation or *probably approximately correct answer*, ie, with probability at least $1 - \delta$ the output is correct within εn. The communication volume and latency of the algorithms are logarithmic in n, p, and $1/\delta$. From a simple algorithm with running time factor $1/\varepsilon^2$ we go to more sophisticated ones with factor $1/\varepsilon$. We also show how to compute the exact result with probability at least $1 - \delta$ if the elements are non-uniformly distributed.

Subsequently, we generalise these results to sum aggregation, where object occurrences are associated with a value. In Section 2.6, we are thus looking for the objects whose values add up to the largest sums. By aggregating the objects locally before sampling, we can handle this problem efficiently with an adaptation of our most frequent objects algorithm.

All of the above algorithms have the unavoidable limitation that the output may be unevenly distributed over the PEs for general distributions of the input objects. This can lead to load imbalance affecting the efficiency of the overall application. Offsetting this imbalance will require communication so that one might argue that striving for communication efficiency in the selection process is in vain. However, our methods have several advantages over non-communication-efficient selection algorithms. First of all, priorities can be ignored during redistribution since all selected elements are relevant. Hence, we can employ any data redistribution algorithm we want. In particular, we can use an adaptive algorithm that only moves data when truly necessary. In Ref. [HSM15], we give one such algorithm that combines prefix

Table 2.2: Our main results. Parameters as in Table 2.1: input size n; output size k; number of PEs p; message startup overhead α; communication cost per machine word β; relative error ε; failure probability δ.

Problem	**Asymptotic running time in our model $\mathcal{O}(\cdot)$**	
	old	**new**
unsorted selection	$\Omega\left(\beta\frac{n}{p}\right) + \alpha \log p$ [San98]	$\frac{n}{p} + \beta \min\left(\sqrt{p}\frac{\log n}{\log p}, \frac{n}{p}\right) + \alpha \log p$
sorted selection	$\Omega(k \log n)$ (seq.) [Var+91]	$\log^2 k + \alpha \log k \log p$ (fixed k) $\log k + \alpha \log p$ (flexible k)
bulk PQ insert* + deleteMin*	$\log \frac{n}{k} + \alpha\left(\frac{k}{p} + \log p\right)$ [San98]	$\log^2 k + \alpha \log k \log p$ (fixed k) $\log k + \alpha \log p$ (flexible k)
heavy hitters	$\frac{n}{p} + \alpha\frac{\sqrt{p}}{\varepsilon} \log n \cdot \log \frac{\log n}{\delta\epsilon}$ [HYZ19] †	*cf.* top-k most frequent items
top-k most frequent items	$\Omega\left(\frac{n}{p} + \beta\frac{k}{\varepsilon} + \alpha\frac{1}{\varepsilon}\right)$ [BO03] †	$\frac{n}{p} + \beta\frac{1}{\varepsilon}\sqrt{\frac{\log p}{p} \log \frac{n}{\delta}} + \alpha \log n$
top-k sum aggregation	$\frac{n}{p} + \beta\frac{1}{\varepsilon}\sqrt{p \log \frac{n}{\delta}} + \alpha p$ (centralised)	$\frac{n}{p} + \beta\frac{\log p}{\varepsilon}\sqrt{\frac{1}{p} \log \frac{n}{\delta}} + \alpha \log n$

† Monitoring query that continuously updates the result

sums and merging to minimise data movement and incurs only logarithmic additional delay.[1] Delegating data movement to the responsibility of the application also has the advantage that we can exploit properties of the application that general selection algorithms cannot. For example, by randomising the necessary data migrations of the priority queue in best-first branch-and-bound applications [KZ93; San98], we can try to steer the system away from situations with bad data distribution. Our approach requires redistribution of the data only when load imbalance has been detected, and not proactively as in prior implementations.

Notation. Refer to Table 2.1 for a summary of the notation used in this chapter. Our input usually consists of a multiset or sequence M of $|M| = n$ objects, each represented by a single machine word. If these objects are ordered, we assume that their total ordering is unique. This is without loss of generality since we can make the value v of object x unique by replacing it by the pair (v, x) for tie-breaking.

Results. Our main results in this chapter are listed in Table 2.2. Refer to Section 2.2.1 for further discussion and proofs of the running times of existing algorithms in our model.

[1] This algorithm is primarily the work of Peter Sanders and is therefore not included in this dissertation.

2.1 Preliminaries

Fast Inefficient Sorting. Using a brute force algorithm that performs all pairwise object comparisons in parallel (see, eg, Ref. [Axt+15]), $\mathcal{O}\left(\sqrt{p}\right)$ randomly distributed objects can be sorted in time $\mathcal{O}(\alpha \log p)$.

Bernoulli Sampling. A simple way to obtain a sample of a set M of objects is to select each object with probability ρ independent of the other objects. The resulting sample has size $\rho\,|M|$ in expectation and is at most a constant factor larger or smaller with high probability. This follows directly from the Chernoff bounds of Equations (1.1) and (1.2). For small sampling probability ρ, the naïve sampling process can be significantly accelerated from time $\mathcal{O}(|M|)$ to time $\mathcal{O}(\rho\,|M|)$ with high probability by using *skip values*. A skip value of x indicates that the following $x-1$ elements are skipped, and the x-th element is sampled. These skip values are geometrically distributed with parameter ρ and can be generated in constant time as $\lfloor \ln(\text{rand}())/\ln(1-\rho) \rfloor$. This formula is obtained by the inversion method using the geometric distribution's continuous analogue, the exponential distribution (see, eg, Ref. [Dev86, Chapter III.2]).

Zipf's Law. In its simplest form, Zipf's Law states that the frequency of an object from a multiset M with $|M| = n$ is inversely proportional to its rank among the objects ordered by frequency. These values are very concentrated and model word frequencies in natural languages, city population sizes, and many other rankings well [Aue13; Zip65]. In this distribution, the most frequent element is j-times more frequent than that of rank j. Here, we consider the general case with exponent parameter s, ie, $x_i = ni^{-s}H_{n,s}^{-1}$, where $H_{n,s} = \sum_{i=1}^{n} i^{-s}$ is the n-th *generalised harmonic number*.

2.2 Related Work

Distributed Selection. Floyd and Rivest [FR75] developed a modification of quickselect [Hoa61] using two pivots to achieve a number of comparisons that is close to the lower bound. This algorithm, FR-select, can be adapted to a distributed memory parallel setting [San98] (very similar algorithms are also described in Refs. [Rei85; Raj90]). FR-select picks the pivots based on sorting a random sample S of $\mathcal{O}\left(\sqrt{p}\right)$ objects, which can easily be done in logarithmic time if the sample is randomly distributed (see Section 2.1). Pivots ℓ and r are chosen as the sample objects with ranks $k\,|S|\,/n \pm \Delta$ where $\Delta = p^{1/4+\delta}$ for some small constant $\delta > 0$. For the analysis, one can choose $\delta = 1/6$, which ensures that with high probability, the range of possible values shrinks by a factor $\Theta(p^{1/3})$ in every level of recursion.

Sanders [San98] proves that a constant number of recursion levels suffice if $n \in \mathcal{O}(p \log p)$ and if the objects are distributed randomly. Note that the latter assumption prevents the algorithm from being communication-efficient because it requires moving all objects to random PEs for general inputs.

Plaxton [Pla89] shows a superlogarithmic lower bound for selection on a large class of interconnection networks. This bound does not apply to our model since we assume a more powerful network.

(Bulk) Parallel Priority Queues. There has been intensive work on parallel priority queues. The most scalable solutions are Sanders's [San98] randomised priority queue and Deo and Prasad's [DP92] parallel heap. Refer to Refs. [Wim+15a; Wim+15b] for a recent overview of further approaches, most of which target shared memory architectures and operate on centralised data structures with limited scalability.

A natural way to parallelise priority queues is to use bulk operations. In particular, operation deleteMin* supports deletion of the k smallest objects of the queue. Such a data structure can be based on a heap where nodes store sorted sequences rather than objects [DP92]. However, an even faster and simpler randomised way is to use multiple sequential priority queues, one on each PE [San98]. This data structure adopts the idea of Karp and Zhang [KZ93] to give every PE a representative approximate view of the global situation by sending inserted objects to random queues. However, in contrast to Karp and Zhang [KZ93], Sanders [San98] implements an exact deleteMin* using parallel selection and a few further tricks. Note that the random insertions prevent communication efficiency in this case.

Most Frequent Objects, Sum Aggregation. Considerable work has been done on distributed top-k computations in wide area networks, sensor networks, and for distributed streaming models [BO03; YZ13; HYZ19]. However, these papers use a server–helper approach where all communication has to go over a coordinator node (*cf.* Section 1.2.4). This implies up to p times higher bottleneck communication volume compared to our results. Only rarely do randomised versions with better running times exist. Nevertheless, communication volume at the coordinator node still increases with $\sqrt{p}$ [HYZ19]. Furthermore, the top-k most frequent objects problem has received little attention in distributed streaming algorithms. The only work we could find requires $\mathcal{O}(Np)$ working memory at the coordinator node, where $N \in \mathcal{O}(n)$ is the number of distinct objects in the union of the streams [BO03]. The majority of papers instead considers the significantly easier problem of identifying the *heavy hitters*, ie, those objects whose occurrences account for more than a fixed proportion of the input, or *frequency tracking*, which tracks the approximate frequencies of all items but requires an additional selection step to obtain the most frequent ones [HYZ19; YZ13; Man+05].

Much work has been done on aggregation in parallel settings, eg, in the database community [CR07; Li+13; Mül+15]. However, these papers all ask for *exact* results for *all* objects, not approximations of the k most important ones. We do not believe that exact queries can be answered in a communication-efficient manner. The multicriteria top-k selection problem, where the items are pre-aggregated and sorted by local aggregate value, and we seek the k items with the highest aggregate values, is somewhat related. However, in addition to the issues listed above, results on the multicriteria top-k selection problem such as Refs. [CW04; MTW05] have severely limited scalability, as the number of PEs cannot exceed the number of criteria.

2.2.1 Running Times of Existing Algorithms

We now prove or give references for the running times given for previous works in Table 2.2.

Unsorted Selection. Previous algorithms, see Ref. [San98], rely on randomly distributed input data or explicitly redistribute the data. Thus, they have to move $\Omega(n/p)$ elements in the worst case.

Sorted Selection. We could not find any prior results on distributed selection from sorted sequences and thus list a sequential result of Varman et al. [Var+91].

Bulk Parallel Priority Queue. The result in Ref. [San98] relies on randomly distributed input data. Therefore, in operation insert*, each PE needs to send its $\mathcal{O}(k/p)$ elements to random PEs, costing $\mathcal{O}(\alpha k/p)$ time. Then, operation deleteMin* is fairly straight-forward and mostly amounts to a selection. The deterministic parallel heap [DP92] needs to sort inserted elements, and then they travel through $\log(n/p)$ levels of the data structure, which is allocated to different PEs. This alone implies communication cost $\Omega(\beta k/p \cdot \log p)$.

Heavy Hitters Monitoring. Huang et al. [HYZ19] give a randomised algorithm for monitoring the heavy hitters that requires time $\mathcal{O}\left(\frac{n}{p} + \alpha\frac{\sqrt{p}}{\varepsilon}\log n \cdot \log\frac{\log n}{\delta\varepsilon}\right)$ in our model.

Proof. All communication is between the controller node and a monitor node. Thus, the maximum amount of communication is at the controller node. Each update, which consists of a constant number of words, is transmitted separately. Hence, the communication term given by the authors transfers directly into our model. Making the algorithm correct for all time instances and increasing the success probability to arbitrary values of δ accounts for the $\log(\log n/(\delta\epsilon))$ factor [HYZ19]. □

Top-k Frequent Objects Monitoring. Monitoring Query 1 of Ref. [BO03] performs top-k most frequent object monitoring in $\Omega(n/p + \beta \cdot k/\varepsilon + \alpha \cdot 1/\varepsilon)$ time for relative error bound ε, ie, no unreported element has more than εn *more* occurrences than a (mistakenly) reported element (in the notation of Ref. [BO03], ε is the absolute error and corresponds to εn in our notation). This algorithm also has further restrictions: it does not provide approximate counts of the objects in the top-k set. It can only handle a small number N of distinct objects, all of which must be known in advance. It requires $\Theta(Np)$ memory on the coordinator node, which is prohibitive if N and p are large. It must also be initialised with a top-k set for the beginning of the streams, using an algorithm such as TA [FLN03]. We now present a family of inputs for which the algorithm uses the claimed amount of communication.

Initialise with $N = k+2$ items $O_1, \ldots, O_{k+2}$ that all have the same frequency. Thus, the initial top-k set comprises an arbitrary subset of k of these. Choose one of the two objects that are *not* in the top-k set—refer to this object as O_s—and pick a peer (PE) N_f. Now, we send O_s to N_f repeatedly, and all other items to all other peers in an evenly distributed manner (each peer receives around the same number of occurrences of each object). After at most $2\varepsilon n$ steps, the top-k set has become invalid and an expensive full resolution step is required.[2] Per the instructions of Ref. [BO03], we choose $F_0 = 0$ and $F_j = 1/(p-1)$ for $j > 0$. We can repeat this cycle to obtain an instance of the example family for any input size n. Note that the number of 'cheap' resolution steps during this cycle depends on the choice of the F_j values, for which Babcock and Olston give rough guidelines of what might constitute a good choice, but do not present a theoretical analysis of how it influences communication. Here, we ignore their cost and focus solely on the cost of 'expensive' resolutions.

[2] This actually happens much earlier, the first 'expensive' resolution is required after only $\mathcal{O}(\varepsilon n/p)$ steps. This number increases, but will never exceed $\sum_{i=0}^{\infty} \varepsilon n(k+1)^{-i} \leq 2\varepsilon n$.

By the above, at least one 'expensive' resolution round is required for every $\mathcal{O}(pn\varepsilon)$ items in the input. Since the resolution set contains $k+1$ objects (the top-k set plus the non-top-k object with a constraint violation), each 'expensive' resolution has communication cost $\Theta(\beta kp + \alpha p)$. Thus, we obtain communication cost for expensive resolutions of $\Omega((\beta kp + \alpha p)/(p\varepsilon)) = \Omega(\beta \cdot k/\varepsilon + \alpha \cdot 1/\varepsilon)$, for a given relative error bound ε. Additionally, each item requires at least constant time in local processing, accounting for the additive $\Omega(n/p)$ term. The actual worst-case communication cost is likely even higher, but this example suffices to show that the approach of Babcock and Olston [BO03] is not communication-efficient in our model.

2.3 Selection

We consider the problem of identifying the element of rank k from a set of n objects which is distributed over p processors. First, we analyse the classical problem with unsorted input, before turning to locally sorted input in Section 2.3.2.

2.3.1 Selection from Unsorted Sequences

Our first result is that using some minor adaptations and careful analysis, the parallel FR-select algorithm of Section 2.2 does not actually need randomly distributed data. We replace the sampling process used in the algorithm of Sanders [San98] with Bernoulli sampling and sample redistribution based on indirect delivery in a hypercube. Algorithm 2.1 gives high-level pseudocode for our selection algorithm.

Theorem 2.1 (Selection from Unsorted Sequences)
Consider n elements distributed over p PEs such that each PE holds $\mathcal{O}(n/p)$ elements. With high probability, the globally k-th smallest of these elements can be identified in time

$$\mathcal{O}\left(\frac{n}{p} + \beta \min\left(\sqrt{p}\log_p n,\ \frac{n}{p}\right) + \alpha \log n\right) .$$

Proof. Our base case is $k = 1$, ie, searching for the global minimum, which is handled accordingly with a minimum reduction. The time for this is scanning (what remains of) the local input plus one collective operation. This is clearly bounded by $\mathcal{O}(n/p + \alpha \log p)$. Otherwise, we pick two pivots and partition the input into three groups. Partitioning takes time linear in the size of what remains of the local input. Determining the total sizes of these groups across all PEs is a sum all-reduction. We then recurse on the group that contains the target item.

The main difficulty is in picking the right pivots (function pickPivots). We adopt the mathematical aspects of pivot selection—which ranks in the sample to choose—from Ref. [San98] (see Section 2.2). The algorithmic parts, however, are somewhat different. Sanders's [San98] algorithm computes the pivots based on a sample of size $\mathcal{O}(\sqrt{p})$. There, this is easy since the objects are randomly distributed in all levels of recursion. Here we can make no such assumption and rather assume Bernoulli sampling with probability $\sqrt{p}/\sum_i |s@i|$. The resulting sample has expected size $\sqrt{p}$, but its exact size varies (though not by more than a constant factor

Algorithm 2.1 : Communication-efficient selection from unsorted sequences.

Input : $s = \langle s_1, \ldots \rangle$ a local sequence of objects, $k \in \mathbb{N}$ the target rank
Output : The k-th smallest object in s

Function select(s, k)
  if $k = 1$ **then return** $\min_i(\min_j s[j])@i$ — *base case: return global minimum*
  $(\ell, r) :=$ pickPivots(s, k)
  $a := \langle e \in s \mid e < \ell \rangle$
  $b := \langle e \in s \mid \ell \leq e \leq r \rangle$
  $c := \langle e \in s \mid e > r \rangle$ — *partition*
  $n_a := \sum_i |a|@i, \quad n_b := \sum_i |b|@i$ — *reduction*
  if $n_a \geq k$ **then return** select(a, k)
  if $n_a + n_b < k$ **then return** select$(c, k - n_a - n_b)$
  return select$(b, k - n_a)$

Input : s, k as in select, δ a parameter in $(0, 1/4)$ chosen freely
Output : ℓ, r, two pivots to use for selecting rank k

Function pickPivots(s, k)
  $n := \sum_i |s@i|$ — *global number of objects*
  Perform Bernoulli sampling with success probability $\rho = \sqrt{p}/n$.
  Spread samples uniformly over the PEs using hypercube routing with random targets.
  Sort samples so that t_i denote the i-th smallest sample — *use fast inefficient sorting*
    (let $t_i = -\infty$ for $i \leq 0$, $t_i = \infty$ for $i >$ #samples).
  $R := k/n \cdot$ #samples — *base rank for pivot choice,* $\mathbf{E}[R] = k\sqrt{p}/n$
  $\Delta :=$ #samples$^{1/2+2\delta}$ — *distance between the samples chosen as pivots,* $\mathbf{E}[\Delta] = p^{1/4+\delta}$
  return $(t_{R-\Delta}, t_{R+\Delta})$ — *lower and upper pivot*

with high probability). Although this can be done in time proportional to the local sample size (see Section 2.1), we have to be careful because we have no guarantees on load balance in the deeper levels of recursion. Since the items are arbitrarily distributed, what remains of the input in some level of recursion might be so imbalanced that most samples come from the same processor. To alleviate this, we spread the sample evenly over the PEs. On the spread sample, we then use the fast inefficient sorting algorithm of Section 2.1, which runs in time $\mathcal{O}(\alpha \log p)$.

To spread the sample, we construct a hypercube over the p PEs (adaptations for values of p that are not powers of two exist, see, eg, Ref. [Kat88]) and determine a target PE for each sample item uniformly at random. Then we route the items to their destination in $\log p$ steps with a bit-fixing algorithm (see, eg, Ref. [MU17, Chapter 4.6.1]). This takes time $\mathcal{O}(\beta\sqrt{p} + \alpha \log p)$ w.h.p. because items always make progress towards their target, and thus no PE can ever send or receive more items than are in the sample, ie, more than $\mathcal{O}(\sqrt{p})$ w.h.p.

By Section 2.2 and Ref. [San98], choosing $\delta = 1/6$ ensures that the total problem size shrinks by a factor of $\Omega(p^{1/3})$ in each level of recursion with high probability. Hence, after iteration i, the local problem size is bounded by $\min(n/p, n/p^{i\Omega(p^{1/3})})$. Summing this over all iterations,

we get $\mathcal{O}(n/p)$ work on each PE and $\log_{\Omega(p^{1/3})}(n/p) \subseteq \mathcal{O}(\log_p n)$ levels of recursion with high probability.

Of course, the bound $\mathcal{O}(\beta n/p)$ also applies to communication volume. However, we also know that even if all samples come from a single PE, the communication volume per level of recursion is bounded by the sample size, which is $\mathcal{O}\left(\sqrt{p}\right)$ w.h.p.

Finally, the number of startups can be limited to $\mathcal{O}(\log p)$ per level of recursion. Spreading and sorting the sample both have $\mathcal{O}(\log p)$ latency per level of recursion, as do the collective operations used in the selection. Thus, we obtain a total latency of $\mathcal{O}(\log p \log_p n) = \mathcal{O}(\log n)$. □

Corollary 2.2 (Selection from Unsorted Sequences with Random Allocation)

If the elements are randomly distributed across the PEs, Algorithm 2.1 can be adapted to run in time $\mathcal{O}(n/p + \alpha \log n)$ w.h.p.

Proof. The adaptation consists of removing the spreading of samples. All other steps communicate only a number of machine words that is subsumed by their latencies. □

Corollary 2.3 (Simplified Formula for Theorem 2.1)

If α and β are viewed as constants, the bound from Theorem 2.1 reduces to $\mathcal{O}(n/p + \log p)$.

Proof. The bound from Theorem 2.1 immediately reduces to $\mathcal{O}(n/p + \log n)$ for constant α and β. We can further simplify this by observing that when the $\log n$ term dominates n/p, then $\log n \in \mathcal{O}(\log p)$. □

2.3.2 Selection from Locally Sorted Sequences

Selection on locally sorted input is easier than the unsorted problem from Section 2.3.1 since we only have to consider the locally smallest objects and can locate keys in logarithmic time. Indeed, this problem has been studied as the *multisequence selection* problem [Var+91; SSP07]. A particularly simple and intuitive method is based on an adaptation of the well-known quickselect algorithm [Hoa61; San+19]. Axtmann et al. [Axt+15, Section 4.1] also describe this algorithm and state that '[t]his algorithm may be folklore', and it has been on the slides of Sanders' lecture on parallel algorithms since at least 2008 [San08]. For self-containedness, we also give that algorithm in Section 2.3.2.a), slightly adapting it to the needs of this dissertation. Subsequently, in Section 2.3.2.b), we describe an algorithm that converges much faster if there is some flexibility in the output rank, ie, the precise value of k is allowed to vary in some range.

2.3.2.a) Multisequence Selection [Axt+15]

This algorithm needs running time $\mathcal{O}(\alpha \log^2 kp)$. It can be seen as a step backwards compared to our algorithm from Section 2.3.1 as using a single random pivot forces us to do a deeper recursion. We do that because it makes it easy to tolerate unbalanced input sizes in the deeper recursion levels. However, it is possible to reduce the recursion depth to some extent by choosing pivots more carefully and by using multiple pivots at once. We study this below for a variant of the selection problem where we are flexible about the number k of objects to be selected.

Algorithm 2.2 : Multisequence selection [Axt+15].

Input : $s = \langle s_1, \ldots \rangle$ a locally sorted sequence of objects, $k \in \mathbb{N}$ the target rank
Output : The k-th smallest object in the union of all PEs' sequences s

```
1 Function msSelect(s, k)
2     if Σ_{1≤i≤p} |s@i| = 1 then                                  — base case
3         return the only nonempty object.
4     Jointly select a pivot v uniformly at random.
5     Find j so that s[1 .. j] < v and s[j + 1 .. |s|] ≥ v.  — eg, binary search on s[1 .. min(k, |s|)]
6     if Σ_{1≤i≤p} |j@i| ≥ k then
7         return msSelect(s[1 .. j]], k)
8     else
9         return msSelect(s[j + 1 .. |s|], k − Σ_{1≤i≤p} |j@i|)
```

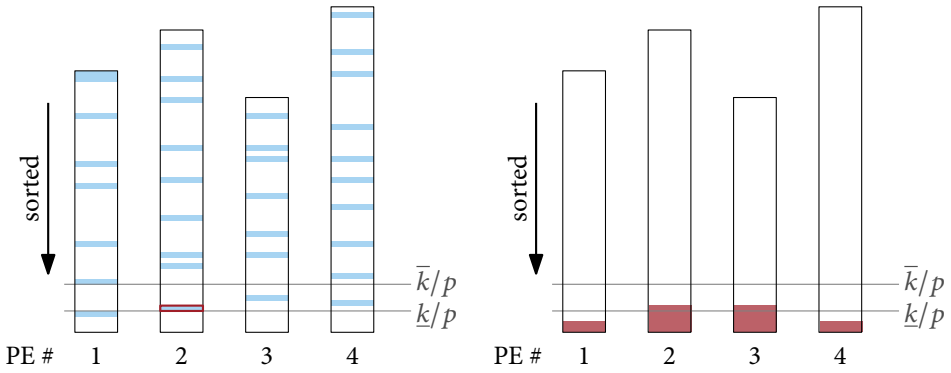
Algorithm 2.2 gives high-level pseudocode. The base case occurs if there is only a single object (and $k = 1$). We can also restrict the search to the first k objects of each local sequence. A random object is selected as a pivot. This can be done in parallel by choosing the same random number between 1 and $\sum_i |s@i|$ on all PEs (eg, by agreeing on a common seed for a pseudorandom number generator beforehand). Using a prefix sum over the sizes of the sequences, this object can be located in time $\mathcal{O}(\alpha \log p)$. Where ordinary quickselect has to partition the input doing linear work, we can exploit the sortedness of the sequences to obtain the same information in time $\mathcal{O}(\log \sigma)$ with $\sigma := \max_i |s@i|$ by doing binary search in parallel on each PE. If items are evenly distributed, we have $\sigma = \Theta(\min(k, n/p))$, and thus only time $\mathcal{O}(\log \min(k, n/p))$ for the search, which partitions all the sequences into two parts. Deciding whether we have to continue searching in the left or the right parts needs a global reduction operation taking time $\mathcal{O}(\alpha \log p)$. As in ordinary quickselect, the expected depth of the recursion is logarithmic in the number of total candidates, ie, $\mathcal{O}(\log \sum_i |s[1 .. k]@i|) \subseteq \mathcal{O}(\log \min(kp, n))$. We obtain the following result.

Theorem 2.4 (Selection from Locally Sorted Sequences (msSelect) [Axt+15])
Algorithm 2.2 can be implemented to run in expected time

$$\mathcal{O}((\alpha \log p + \log \min(n/p,\, k)) \cdot \log \min(kp, n)) \subseteq \mathcal{O}\big(\alpha \log^2 kp\big) \ .$$

2.3.2.b) Selection from Locally Sorted Sequences with Flexible k

The $\mathcal{O}(\log^2 kp)$ startups incurred in Theorem 2.4 can be reduced to $\mathcal{O}(\log p)$ if we are willing to give up some control over the number of objects that are returned. We now accept two input parameters $\underline{k}$ and $\overline{k}$, and the algorithm returns the k smallest objects with $\underline{k} \le k \le \overline{k}$. We begin with a simple algorithm that runs in logarithmic time if $\overline{k} - \underline{k} \in \Omega(\overline{k})$ and then explain how to refine that for the case $\overline{k} - \underline{k} \in o(\overline{k})$.



(a) Step 1: Bernoulli sample of the input (in blue), red box marks globally smallest sample.

(b) Step 2: Fewer than $\underline{k}$ elements—marked in red—are smaller than the pivot.

Figure 2.1: Example of amsSelect showing a single level of the recursion. The algorithm proceeds with a recursive call on the elements larger than the pivot, with $\underline{k}$ and $\overline{k}$ reduced by the number of elements excluded from consideration.

The basic idea for the simple algorithm is to take a Bernoulli sample of the input using a success probability of $1/x$ for some $x \in \left[\underline{k}\,..\,\overline{k}\right]$ (Figure 2.1 (a)). Then, the expected rank of the smallest sample object is x, ie, we have a truthful estimator for an object with the desired rank. Moreover, this object can be computed efficiently if working with locally sorted data: the local rank of the smallest local sample is geometrically distributed with parameter $1/x$. Such a number can be generated in constant time (see Section 2.1). By computing the global minimum of these locally smallest samples, we can get the globally smallest sample v in time $\mathcal{O}(\alpha \log p)$. We can also count the exact number k of input objects bounded by this estimate in time $\mathcal{O}(\log k + \alpha \log p)$—we locate v in each local data set in time $\mathcal{O}(\log k)$ and then sum the found positions (Figure 2.1 (b)). If $k \in \left[\underline{k}\,..\,\overline{k}\right]$, we are done. Otherwise, we can use the acquired information as in any variant of quickselect.

At least in the recursive calls, $\overline{k}$ can be close to the total (remaining) input size. Then it is a better strategy to use a dual algorithm based on computing a global maximum of a Bernoulli sample. In Algorithm 2.3 we give pseudocode for a combined algorithm which dynamically chooses between these two cases. It is an interesting problem which value should be chosen for x. The formula used in Algorithm 2.3 maximises the probability that $k \in \left[\underline{k}\,..\,\overline{k}\right]$. Interestingly, $x \neq 2/(\overline{k} + \underline{k})$, ie, it is *not* optimal to aim for $\mathbf{E}[k] = (\overline{k} + \underline{k})/2$. While the actual value, whose derivation we give below, is close to the arithmetic mean when $\overline{k}/\underline{k} \approx 1$, it is significantly smaller otherwise. The reason for this asymmetry is that larger sampling rates decrease the variance of the geometric distribution.

Theorem 2.5 (Approximate Selection from Locally Sorted Sequences (amsSelect))
If $\overline{k} - \underline{k} \in \Omega\left(\overline{k}\right)$, *then* amsSelect *from Algorithm 2.3 finds the k smallest elements with* $\underline{k} \leq k \leq \overline{k}$ *in expected time* $\mathcal{O}\left(\log \overline{k} + \alpha \log p\right)$.

Algorithm 2.3 : Approximate multisequence selection.

Input : $s = \langle s_1, \ldots \rangle$ a locally sorted sequence of objects; $\underline{k}, \overline{k} \in \mathbb{N}$ upper and lower bounds on the number of selected objects; $n = \sum_{i=1}^{p} |s@i| \in \mathbb{N}$ the global input size

Output : The local part of the k globally smallest objects for some $k \in \mathbb{N}$ with $\underline{k} \le k \le \overline{k}$

```
1  Function amsSelect(s, k̲, k̄, n)
2      if k̲ < n − k̄ then                                              — min-based estimator
3          x := ⌊(k̄ − k̲ + 1) ln(rand()) / ln((k̲−1)/(k̄))⌋              — x ~ Geo(1 − ((k̲−1)/k̄)^(1/(k̄−k̲+1)))
4          if x > |s| then v := ∞ else v := s[x]
5          v := min_{1≤i≤p} v@i                                       — minimum reduction
6      else                                                           — max-based estimator
7          x := ⌊(k̄ − k̲ + 1) ln(rand()) / ln((n−k̄)/(n−k̲+1))⌋          — x ~ Geo(1 − ((n−k̄)/(n−k̲+1))^(1/(k̄−k̲+1)))
8          if x > |s| then v := −∞ else v := s[|s| − x + 1]
9          v := max_{1≤i≤p} v@i                                       — maximum reduction
10     Find j so that s[1 .. j] ≤ v and s[j + 1 .. |s|] > v.   — eg, binary search on s[1 .. min(k, |s|)]
11     k := Σ_{1≤i≤p} |j@i|                                           — sum all-reduction
12     if k < k̲ then                                                  — recursion on larger elements
13         return s[1 .. j] ∪ amsSelect(s[j + 1 .. |s|], k̲ − k, k̄ − k, n − k)
14     else if k > k̄ then                                             — recursion on smaller elements
15         return amsSelect(s[1 .. j], k̲, k̄, k)
16     else
17         return s[1 .. j]
```

Proof. One level of recursion takes time $\mathcal{O}(\alpha \log p)$ for collective communication operations (min or max and sum reduction) and time $\mathcal{O}(\log \overline{k})$ with high probability for locating the pivot v using exponential search, as it is among the first $\mathcal{O}(\overline{k})$ items in each local list with high probability. It remains to show that the expected recursion depth is constant.

We actually analyse a weaker algorithm that keeps retrying with the same parameters rather than using recursion and that uses probe $x = \underline{k}$. We show that, nevertheless, there is a constant success probability (ie, $\underline{k} \le k \le \overline{k}$ with constant probability). The rank of v is geometrically distributed with parameter $1/x$. The success probability becomes

$$\mathbf{P}\left[\underline{k} \le k \le \overline{k}\right] = \left(1 - \frac{1}{\underline{k}}\right)^{\underline{k}-1} - \left(1 - \frac{1}{\underline{k}}\right)^{\overline{k}} = \frac{\underline{k}}{\underline{k}-1}\left(1 - \frac{1}{\underline{k}}\right)^{\underline{k}} - \left(1 - \frac{1}{\underline{k}}\right)^{\underline{k}\cdot\frac{\overline{k}}{\underline{k}}} \approx \frac{1}{e} - e^{-\frac{\overline{k}}{\underline{k}}} \ .$$

This is a positive constant if $\overline{k} - \underline{k} \in \Omega\left(\overline{k}\right)$, and the last simplification follows from the classic formula $e^x = \lim_{n\to\infty}\left(1 + \frac{1}{n}\right)^{nx}$. Since the actual value of x is chosen to maximise the proba-

bility that $\underline{k} \le k \le \overline{k}$ (see derivation below), our actual algorithm converges at least as fast as the version analysed above. Thus, the expected recursion depth is constant. □

We now show the derivation of the formula used in Algorithm 2.3. Observe that $k \in [\underline{k}\,..\,\overline{k}]$ if and only if the rank of the first item in a Bernoulli sample with some success probability ρ is between $\underline{k}$ and $\overline{k}$. This rank follows a geometric distribution with parameter ρ, and therefore, $\mathbf{P}[k \le a] = 1 - (1-\rho)^a$ for any $a \in \mathbb{N}$. Thus, we obtain:

$$\begin{aligned}\mathbf{P}\left[\underline{k} \le k \le \overline{k}\right] &= \mathbf{P}\left[k \le \overline{k}\right] - \mathbf{P}[k < \underline{k}] \\ &= (1-\rho)^{\underline{k}-1} - (1-\rho)^{\overline{k}},\end{aligned}$$

which we wish to maximise. Hence, we must set its derivative equal to zero and solve for ρ:

$$\begin{aligned}\frac{\partial\, \mathbf{P}\left[\underline{k} \le k \le \overline{k}\right]}{\partial \rho} &= \overline{k}(1-\rho)^{\overline{k}-1} - (\underline{k}-1)(1-\rho)^{\underline{k}-2} \stackrel{!}{=} 0 \\ \Leftrightarrow\ (1-\rho)^{\underline{k}-2}\left(\overline{k}(1-\rho)^{\overline{k}-\underline{k}+1} - \underline{k} + 1\right) &= 0 \\ \Leftrightarrow\ (1-\rho)^{\overline{k}-\underline{k}+1} &= \frac{\underline{k}-1}{\overline{k}} \\ \Leftrightarrow\ \rho &= 1 - \left(\frac{\underline{k}-1}{\overline{k}}\right)^{\frac{1}{\overline{k}-\underline{k}+1}}.\end{aligned}$$

The second case follows analogously.

Additional Improvement: Multiple Concurrent Trials

The running time of Algorithm 2.3 is dominated by the logarithmic number of startup overheads for the two reduction operations it uses. We can exploit that reductions can process multiple items using little additional time. The idea is to execute d copies of the pivot selection process simultaneously, ie, taking d Bernoulli samples of the input and computing d estimates for an object of rank x. If the rank of any of these estimates turns out to be between $\underline{k}$ and $\overline{k}$, the recursion can be stopped. Otherwise, we solve a recursive instance consisting of those objects enclosed by the largest underestimate and the smallest overestimate found.

Theorem 2.6 (amsSelect with Multiple Concurrent Trials)
If $\overline{k} - \underline{k} \in \Omega\left(\overline{k}/d\right)$, an algorithm processing batches of d Bernoulli samples can be implemented to run in expected time $\mathcal{O}\left(d \log \overline{k} + \beta d + \alpha \log p\right)$.

Proof. A single level of recursion takes expected time $\mathcal{O}(d \log \overline{k} + \beta d + \alpha \log p)$ for taking d samples and two reductions on vectors of length d. The success probability for any single sample is $\Omega(1/d)$ analogous to the proof of Theorem 2.5: $\mathbf{P}[\underline{k} \le k \le \overline{k}] \approx 1/e - \exp(-\overline{k}/\underline{k})$ is in $\Omega(1/d)$ for each sample if $\overline{k} - \underline{k} \in \Omega(\overline{k}/d)$. Therefore, the probability that at least one of the d independent samples is successful is a positive constant. The remainder of the analysis proceeds as in Theorem 2.5, again retrying with the same parameters rather than using recursion. □

For example, setting $d = \Theta(\log p)$, we obtain a total running time of $\mathcal{O}(\log k \log p + \alpha \log p)$ in expectation for an allowed variation of d in the output rank.

2.3.2.c) Adaptations for Exact Output Rank

In Section 2.3.2.a), we noted that the pivot choice in algorithm msSelect could be improved to reduce the expected recursion depth. One possibility to construct such an algorithm is setting $\underline{k} = \overline{k} = k$ in algorithm amsSelect, which we analyse in the following. We directly observe that setting $\underline{k} = \overline{k}$ simplifies the pivot choice formulae in Lines 3 and 7 of Algorithm 2.3 to $x := \lfloor \ln(\text{rand}())/\ln(1-1/k) \rfloor$ and $x := \lfloor \ln(\text{rand}())/\ln(1-1/(n-k+1)) \rfloor$, respectively. These are geometrically distributed with parameters $1/k$ and $1/(n-k+1)$. The expected rank of the smallest (or largest, respectively) sample is thus the target rank k. However, our previous analysis, which computes the probability of hitting a rank between $\underline{k}$ and $\overline{k}$ without taking recursion into account, is of no help if $\underline{k} = \overline{k}$. Therefore, we now provide an analysis of amsSelect for exact output rank.

Theorem 2.7 (amsSelect with Exact Output Rank)
When using amsSelect *with* $\underline{k} = \overline{k} = k$*, its running time is* $\mathcal{O}(\log^2 k + \alpha \log k \log p)$ *in expectation.*

Proof. As observed above, the success probability of the sampling process becomes $1/k$ (without loss of generality, we assume the min-based case for the remainder of the proof, and the max-based case, where the success probability is $1/(n-k+1)$, follows symmetrically). Then the expected rank of the smallest sample is k. Moreover, there is a constant probability that its rank is larger than k by a constant factor. Let X be the random variable describing the pivot's rank. Using the geometric distribution function's cumulative distribution function, we obtain that this probability is

$$\begin{aligned}\mathbf{P}[k < X \leq c \cdot k] &= \mathbf{P}[X \leq c \cdot k] - \mathbf{P}[X \leq k] \\ &= (1-1/k)^k - (1-1/k)^{c \cdot k} \\ &\geq e^{-1} - e^{-c} \quad .\end{aligned}$$

This is a positive constant for all $c > 1$. Therefore, within a constant number of recursion levels (or using a constant number of concurrent trials), we expect to have found a pivot that reduces the problem size from n to $\Theta(k)$.

We also have a constant probability of the pivot's rank being between $q \cdot k$ and k for some positive constant q. We can again use the geometric distribution's cumulative distribution function to compute the probability that the pivot's rank is only a constant factor smaller than k. Let $q \in (0,1)$. Then $\mathbf{P}[q \cdot k < X \leq k] \geq e^{-q} - e^{-1}$ by the same derivation as above. This probability is also a positive constant for all $q \in (0,1)$. Therefore, we also expect the lower bound to shrink by a constant factor within a constant number of recursive calls.

Thus, within an expected $\mathcal{O}(\log k)$ further levels of recursion, we will have solved the problem completely. In total, the expected recursion depth is $\mathcal{O}(1 + \log k)$. Since the local work per recursive call is bounded by $\mathcal{O}(\log k + \alpha \log p)$ (Theorem 2.5), the claimed bound follows. □

Of course, we can again use multiple concurrent trials to reduce the expected recursion depth further.

2.4 Application: Bulk Parallel Priority Queues

We build a global bulk-parallel priority queue from local sequential priority queues as in Refs. [KZ93; San98], but never actually move elements. This immediately implies that insertions simply go to the local queue and thus require only $\mathcal{O}(\log n)$ time without any communication. Of course, this complicates operation deleteMin*. The number of elements to be retrieved from the individual local queues can vary arbitrarily, and the set of elements stored locally is not at all representative for the global content of the queue. We can not even afford to individually remove the objects returned by deleteMin* from their local queues.

To fulfil these requirements, we replace the ordinary priority queues used by Sanders [San98] with search tree data structures that support insertion, deletion, selection and ranking of objects as well as splitting and joining trees in logarithmic time (see also Section 1.3.4). To become independent of the actual tree size of up to n, we furthermore augment the trees with two arrays storing the path to the smallest and largest object respectively. To access an element with rank $i \le n/2$, we thus jump directly to position $\lceil \log n \rceil - \lceil \log i \rceil$ of the former path (for $i > n/2$, jump to position $\lceil \log n \rceil - \lceil \log(n-i) \rceil$ of the latter path). This allows us to skip the search for the leftmost (rightmost) subtree in which the item is located. Thus, we can restrict the search to a subtree of height $\mathcal{O}(\log \min(i, n-i))$ in constant time. Because we only require access to items with rank up to k here, this way, all required operations can be implemented to run in time $\mathcal{O}(\log \min(k, n-k))$ rather than $\mathcal{O}(\log n)$ (without loss of generality, we shall assume $k < n-k$ for the remainder of this section).

Operation deleteMin* now becomes very similar to the multi-sequence selection algorithms from Section 2.3.2. The only difference is that instead of sorted arrays, we are now working on search trees. This implies that selecting a local object with specified local rank (during sampling) now takes time $\mathcal{O}(\log k)$ rather than constant time. However, asymptotically, this makes no difference since, for any such selection step, we also perform a ranking step, which takes time $\mathcal{O}(\log k)$ anyway in both representations.

One way to implement the recursion in the selection algorithms is via splitting. Since the split operation is destructive, after returning from the recursion, we have to reassemble the previous state using a join operation. This allows us to keep the description simple. Another way that is faster and simpler to implement in practice is to represent a subsequence of s by s itself plus cursor information specifying rank and key of the first and last object of the subsequence.

Now, we obtain the following by applying our results on selection from Section 2.3.2.

Corollary 2.8 (Bulk Deletion from Parallel Priority Queues)
Operation deleteMin* *can be implemented to run in the following expected times. With fixed batch size k, expected time $\mathcal{O}(\log^2 k + \alpha \log k \log p)$ suffices. For flexible batch sizes in $[\underline{k} .. \overline{k}]$ with $\overline{k} - \underline{k} \in \Omega(\overline{k})$, we need expected time $\mathcal{O}(\log \overline{k} + \alpha \log p)$. For $\overline{k} - \underline{k} \in \Omega(\overline{k}/d)$, expected time $\mathcal{O}(d \log \overline{k} + \beta d + \alpha \log p)$ is sufficient.*

Note that flexible batch sizes might be adequate for many applications. For example, the parallel branch-and-bound algorithm from Ref. [San98] can easily be adapted: in iteration i of its main loop, it deletes the smallest $k_i \in \mathcal{O}(p)$ elements (tree nodes) from the queue, expands these nodes in parallel, and inserts newly generated elements (child nodes of the processed

nodes). Let $K = \sum_i k_i$ denote the total number of nodes expanded by the parallel algorithm, define m as the number of nodes expanded by a sequential best first algorithm, and let h be the length of the path from the root to the optimal solution. The original algorithm, deleting exactly p elements in each iteration, needs at most $m/p + h$ iterations. This bound also holds for our algorithm: because we never expand fewer nodes than the original algorithm, we require at most as many iterations. Therefore, we obtain a bound of $K = m + \mathcal{O}(hp)$ on the number of nodes expanded by our adaptation. Note also that a typical branch-and-bound computation will insert significantly more nodes than it removes—the remaining queue is discarded after the optimal solutions are found. Hence, the local insertions of our communication-efficient queue are a big advantage over previous algorithms, which move all nodes [KZ93; San98].

In Section 3.7, we use this bulk-parallel priority queue to construct an efficient distributed reservoir sampling for uniform and weighted inputs.

2.5 Top-k Most Frequent Objects

We describe two *probably approximately correct (PAC)* algorithms to compute the top-k most frequent objects of a multiset M with $|M| = n$, followed by a *probably exactly correct (PEC)* algorithm for suitable inputs. Sublinear communication is achieved by transmitting only a small random sample of the input. For bound readability, we assume that M is distributed over the p PEs such that no PE has more than $\mathcal{O}(n/p)$ objects. This is not restricting, as the algorithms' running times scale linearly with the maximum fraction of the input concentrated at one PE.

We express the algorithms' error relative to the total input size n. This is reasonable—consider a large input where k objects occur twice, and all others once. If we expressed the error relative to the objects' frequencies, the problem could not be solved in a communication-efficient way for such inputs. Thus, we refer to the algorithms' *absolute* error as E, which we define as the count of the most frequent object that was missed by the algorithm minus that of the least frequent object that was erroneously output. If the result is exact, then $E = 0$. With this definition, the relative error e becomes E/n. Let δ limit the probability that the output exceeds bound ε, ie, $\mathbf{P}[e > \varepsilon] \leq \delta$. We then refer to the result as an (ε, δ)-approximation.

2.5.1 Basic Approximation Algorithm

First, we take a Bernoulli sample of the input. Sampling is done locally. The frequencies of the sampled objects are counted using distributed hashing—a local object count with key k is sent to PE $h(k)$ for a hash function h that we here expect to behave like an idealised random function. Sending uses a specialised all-to-all collective communication operation. We then select the k most frequently sampled objects using the unsorted selection algorithm from Section 2.3.1. An example is illustrated in Figure 2.2.

Theorem 2.9 (Most Frequent Objects, Probably Approximately Correct Algorithm)
Algorithm PAC *can be implemented to compute an* (ε, δ)*-approximation of the top-k most frequent objects in expected time* $\mathcal{O}\left(\beta \frac{\log p}{p\varepsilon^2} \log \frac{k}{\delta} + \alpha \log n\right)$.

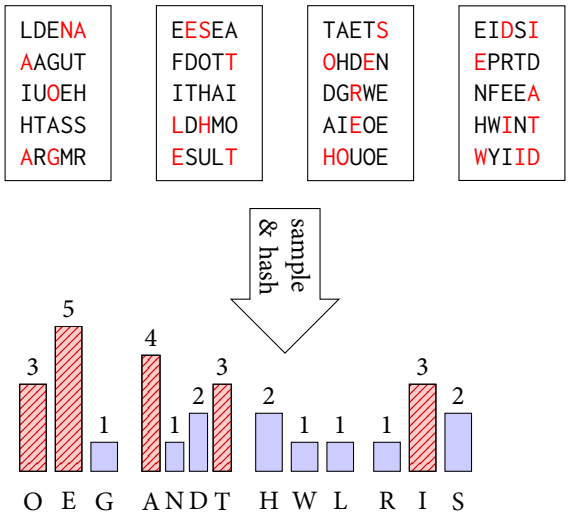


Figure 2.2: Example for our PAC top-k most frequent objects algorithm. Top: input distributed over four PEs, sampled elements ($\rho = 0.3$) highlighted in red. Bottom: distributed hash table counting the samples. The $k = 5$ most frequent objects, highlighted in red, are to be returned after scaling with $1/\rho$. Estimated result: $(E, 17)$, $(A, 13)$, $(T, 10)$, $(I, 10)$, $(O, 10)$. Exact result: $(E, 16)$, $(A, 10)$, $(T, 10)$, $(I, 9)$, $(D, 8)$. Object D was missed, instead O (count 7) was returned. Thus, the algorithm's error is $8 - 7 = 1$ here.

To prove this theorem, we first consider a fixed sampling probability, and show the resulting running time in Lemma 2.10 and error in Lemma 2.11. By solving the error formula for the required sample size, we obtain the final piece of the puzzle.

Lemma 2.10 (Most Frequent Objects, Algorithm PAC Running Time)
For sampling probability ρ, Algorithm PAC *runs in $\mathcal{O}\left(\beta\frac{n\rho}{p}\log p + \alpha\log n\right)$ time in expectation.*

Proof. Bernoulli sampling is done in time $\mathcal{O}(n\rho/p)$ with high probability by generating skip values with a geometric distribution using success probability ρ (see Section 2.1). Since the number of sample elements in a Bernoulli sample is a random variable, so is the running time. Each PE's local sample has size $\mathcal{O}(n\rho/p)$ with high probability by a simple application of the Chernoff bound of Equation (1.2).[3]

To count the sampled objects, each PE aggregates its local samples in a hash table, counting the occurrence of each sample object during the sampling process. It inserts its share of the sample into a distributed hash table [SM13] whose hash function we assume to behave like

[3] We can also bound the maximum size of *any* PE to at most $\mathcal{O}(n\rho/p + \log p)$ with high probability. To obtain this bound, apply a Normal approximation to the per-PE sample sizes, which are binomially distributed. Then we can apply a bound on the maximum of p i.i.d. Normal random variables, similar to the process used in the proof of Theorem 3.18.

a random function, thus distributing the objects randomly among the PEs. The elements are communicated using indirect delivery to maintain logarithmic latency. To reduce communication volume, the incoming sample counts are merged with a hash table in each step of the reduction. Because each item's destination is random, this ensures that every item's occurrences are aggregated directly. This keeps the expected size of these messages to $\mathcal{O}(n\rho/p)$. Therefore, the entire distributed hash table insertion process takes time $\mathcal{O}(\beta n\rho/p \cdot \log p + \alpha \log p)$ in expectation.

From this hash table, we select the object with rank k using Algorithm 2.1 in expected time $\mathcal{O}(\beta n\rho/p + \alpha \log n\rho)$. This pivot is broadcast to all PEs, which then determine their subset of at least as frequent sample objects in expected time $\mathcal{O}(n\rho/p + \alpha \log p)$. These elements are returned. Overall, the claimed time complexity follows using the estimates $p \leq n$ and $n\rho \leq n$ to simplify the α-term. □

Lemma 2.11 (Most Frequent Objects, Algorithm PAC Error Bounds)
Let $\rho \in (0, 1)$ be the sampling probability. Then, Algorithm PAC*'s absolute error E exceeds a value of Δ with probability*

$$\mathbf{P}[E \geq \Delta] \leq n \cdot \exp\left(-\frac{\Delta^2 \rho k}{12n}\right) + k \cdot \exp\left(-\frac{\Delta^2 \rho}{8n}\right).$$

Proof. We bound the error probability as follows: the probability that the absolute error E exceeds some value Δ is at most n times the probability that a single object's value estimate deviates from its true value by more than $\Delta/2$ in either direction. For non-top-k objects, our concern is the possibility of *overestimation*, while for top-k objects, *underestimation* is interesting. These probabilities can be bounded using Chernoff bounds. We denote the count of element j in the input by x_j and in the sample by s_j. Further, let F_j be the probability that the error for element j exceeds $\Delta/2$ in either direction. Let $j \geq k$, and observe that $x_j \leq n/k$. Using Equation (1.2) with $X = s_j$, $\mathbf{E}[X] = \rho x_j$, and $\varphi = \frac{\Delta}{2x_j}$, we obtain:

$$F_j \leq \mathbf{P}\left[s_j \geq \rho\left(x_j + \frac{\Delta}{2}\right)\right] \leq \exp\left(-\frac{\Delta^2 \rho k}{12n}\right)$$

This leaves us with the most frequent $k-1$ elements, whose counts can be bounded as $x_j \leq n$. As overestimating them is not a concern, we apply the Chernoff bound in Equation (1.1) and obtain $\mathbf{P}\left[s_j \leq \rho\left(x_j - \frac{\Delta}{2}\right)\right] \leq \exp\left(-\frac{\Delta^2 \rho}{8n}\right)$. In sum, all error probabilities add up to the claimed value. □

We bound this result by an error probability δ. This allows us to calculate the minimum required sampling probability given δ and $\Delta = \varepsilon n$. Solving the above equation for ρ yields

$$\rho n \geq \frac{4}{\varepsilon^2} \cdot \max\left(\frac{3}{k} \ln \frac{2n}{\delta},\ 2 \ln \frac{2k}{\delta}\right), \tag{2.1}$$

which is dominated by the latter term in most cases and yields $\rho n \geq \frac{8}{\varepsilon^2} \ln \frac{2k}{\delta}$ for the expected sample size.

Proof (Theorem 2.9). Equation (2.1) yields $\rho n = \mathcal{O}\left(\frac{1}{\varepsilon^2} \log \frac{k}{\delta}\right)$. The claimed running time bound then follows from Lemma 2.10. □

Note that if we can afford to aggregate the local input, we can also use the Sum Aggregation algorithm from Section 2.6 and associate a value of 1 with each object.

2.5.2 Increasing Communication Efficiency

Sample sizes proportional to $1/\varepsilon^2$ quickly become unacceptably large as ε decreases. To remedy this, we iterate over the local input a second time and count the most frequently sampled objects' occurrences exactly. This allows us to reduce the sample size and improve communication efficiency at the cost of increased local computation. We call this *Algorithm EC* for *exact counting*. Again, we begin by taking a Bernoulli sample. Then we find the $k^* \geq k$ globally most frequent objects in the sample using the unsorted selection algorithm from Section 2.3.1, and count their frequency in the overall input exactly. The identity of these objects is broadcast to all PEs using an all-gather (gossiping, all-to-all broadcast) collective communication operation. After local counting, a global reduction sums up the local counts to exact global values. The k most frequent of these are then returned.

Lemma 2.12 (Most Frequent Objects, Exact Counting Size)
When counting the k' most frequently sampled objects' occurrences exactly, choosing ρ for an expected sample size of $n\rho = \frac{2}{\varepsilon^2 k'} \ln \frac{n}{\delta}$ suffices to ensure that the result of Algorithm EC *is an (ε, δ)-approximation of the top-k most frequent objects.*

Proof. With the given error bounds, we can derive the required sampling probability ρ similarly as in Lemma 2.11. However, we need not consider overestimation of the k' most frequent objects, as their counts are known exactly. We can also allow the full margin of error towards underestimating their frequency (Δ instead of $\Delta/2$) and can ignore overestimation. This way, we obtain a total expected sample size of $\rho n \geq 2/(\varepsilon^2 k') \cdot \ln(n/\delta)$. □

We can now calculate the value of k' that minimises the total communication volume and obtain $k^* = \max(k, \frac{1}{\varepsilon}\sqrt{\frac{2 \log p}{p} \ln \frac{n}{\delta}})$. Substituting this into the sample size equation of Lemma 2.12 and adapting the running time bound of Lemma 2.10 then yields the following theorem.

Theorem 2.13 (Most Frequent Objects, Exact Counting Algorithm)
Algorithm EC *can be implemented to compute an (ε, δ)-approximation of the top-k most frequent objects in $\mathcal{O}\left(\frac{n}{p} + \beta \frac{1}{\varepsilon}\sqrt{\frac{\log p}{p} \cdot \log \frac{n}{\delta}} + \alpha \log n\right)$ time in expectation.*

Proof. Sampling and hashing are done as in Algorithm *PAC* (Section 2.5.1), with the sampling probability of Lemma 2.12 and $k' = k^*$. From the resulting distributed hash table, we select the object with rank $k^* = \max\left(k, \frac{1}{\varepsilon}\sqrt{\frac{2 \log p}{p} \ln \frac{n}{\delta}}\right)$ as a pivot. This requires expected time $\mathcal{O}\left(\beta \frac{n\rho}{p} + \alpha \log n\rho\right)$ using Algorithm 2.1. The pivot is broadcast to all PEs, which then determine

their subset of at least as frequent sample objects using $\mathcal{O}(n\rho/p + \alpha \log p)$ time in expectation. Next, these k^* most frequently sampled objects are distributed to all PEs using an all-gather operation in time $\mathcal{O}(\beta k^* + \alpha \log p)$. Now, the PEs count the received objects' occurrences in their local input, which takes $\mathcal{O}(n/p)$ time. These counts are summed up using a vector-valued reduction, again requiring $\mathcal{O}(\beta k^* + \alpha \log p)$ time. We then apply Algorithm 2.1 a second time to determine the k most frequent of these objects. Overall, the claimed time complexity follows by substituting k^* for k' in the sampling probability from Lemma 2.12. □

Substituting the value of k^* from before then yields $\rho n = 1/\varepsilon \cdot \sqrt{p/\log p \cdot \ln(n/\delta)}$ for the total required sample size of Algorithm *EC*. Note that this term grows with $1/\varepsilon$ instead of $1/\varepsilon^2$, reducing per-PE communication volume to $\mathcal{O}\left(\frac{1}{\varepsilon}\sqrt{\frac{\log p}{p}} \log \frac{n}{\delta}\right)$ machine words in expectation.

To continue the example from Figure 2.2, we may set $k^* = 8$. Then, the k^* most frequently sampled objects (E, A, T, I, O, D, H, S) with $(16, 10, 10, 9, 7, 8, 7, 6)$ occurrences, respectively, will be counted exactly. The result would now be correct.

Note that Algorithm *EC* can be *less* communication-efficient than Algorithm *PAC* if ε is large, ie, the result is very approximate. Then, k^* can be prohibitively large, and the necessity to communicate the identity of the objects to be counted exactly, requiring time $\mathcal{O}(\beta k^* + \alpha \log p)$, can cause a loss in communication efficiency.

2.5.3 Probably Exactly Correct Algorithm

Up until now, our algorithms are independent of the characteristics of the input. However, this makes it hard to design communication-efficient algorithms that are *probably exactly correct*, ie, algorithms that are exactly correct with probability at least $1 - \delta$ for some $\delta > 0$. This is because there might be numerous items with very similar counts, and distinguishing them with a sample-based approach would require excessive sample sizes to give good guarantees. We can work around this if we can assume that the frequency distribution of the objects has some kind of *gap* (see Figure 2.3 for an illustration). Then, there is a limited number of relevant objects, which we can identify in the sample, and on which we can then perform exact counting to obtain a probably exactly correct result. Thus, choose k^* to ensure that the top-k most frequent objects in the input are among the top-k^* most frequent objects in the sample with probability at least $1 - \delta$.

A *probably exactly correct (PEC)* algorithm to compute the top-k most frequent objects of a multiset whose frequency distribution has a sufficient gap can therefore be formulated as follows. Take a small sample with sampling probability ρ_0, the value of which we will consider later. From this small sample, we deduce the required value of k^* to fulfil the above requirements. Now, we apply Algorithm *EC* using this value of k^*. Let x_i be the object of rank i in the input, and $\hat{s}_j$ that of rank j in the small sample.

Lemma 2.14 (Most Frequent Objects, Algorithm PEC Parameter Choice)
It suffices to choose k^* in such a way that $\hat{s}_{k^*} \leq \rho_0 x_k - \sqrt{2\rho_0 x_k \ln \frac{k}{\delta}} = \mathbf{E}[\hat{s}_k] - \sqrt{2\mathbf{E}[\hat{s}_k] \ln \frac{k}{\delta}}$ to ensure that the result of Algorithm PEC *is correct with probability at least $1 - \delta$.*

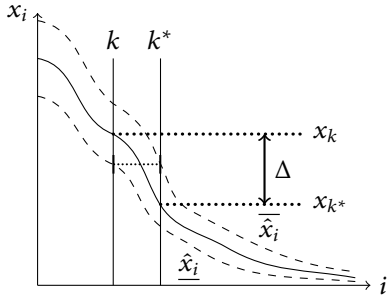


Figure 2.3: Example of a distribution with a gap. The dashed lines indicate the upper and lower high-probability bounds on x_i estimated from the sample.

Proof. For the result to be incorrect, a top-k object must not be part of the top-k^* objects of the sample, causing it to be eliminated before exact counting. We use the Chernoff bound of Equation (1.1) to bound the probability that this happens. Define $\hat{s}(x_i)$ as the number of samples of the object with input rank i in the first sample, and $x(\hat{s}_j)$ to be the exact number of occurrences in the input of the object with rank j in the first sample. Using $X = \hat{s}(x_k)$, $\mathbf{E}[X] = \rho_0 x_k$, and $\varphi = 1 - \hat{s}_{k^*}/(\rho_0 x_k)$, we obtain

$$\begin{aligned}\sum_{j=1}^{k} \mathbf{P}\left[\hat{s}(x_j) \leq \hat{s}_{k^*}\right] &\leq k \cdot \mathbf{P}\left[\hat{s}(x_k) \leq \hat{s}_{k^*}\right] \\ &\leq k \cdot \exp\left(-\left(1 - \frac{\hat{s}_{k^*}}{\rho_0 x_k}\right)^2 \frac{\rho_0 x_k}{2}\right) .\end{aligned}$$

We bound this value by the algorithm's error probability δ and solve for $\hat{s}_{k^*}$, which yields the claimed value. □

This only works for distributions with a sufficient gap, as otherwise, $k^* \gg k$ would be necessary. Furthermore, the choice of ρ_0 presents a trade-off between the time and communication spent on the first sample and the exactness of k^*, which we have to estimate more conservatively if the first sample is small. This is due to less precise estimations of $\mathbf{E}[\hat{s}_k] = \rho_0 x_k$ if ρ_0 is small. To keep things simple, we can choose a relative error bound ε_0 and use the sample size from our PAC algorithm of Theorem 2.9. The value of ε_0—and thus, ρ_0—that minimises the communication volume depends on the distribution of the input data.

Theorem 2.15 (Most Frequent Objects, Probably Exactly Correct Algorithm)
If the value of k^* computed from Lemma 2.14 satisfies $k^* \in \mathcal{O}(k)$, then Algorithm PEC *requires time asymptotically equal to the sum of the running times of algorithms* PAC *and* EC *from Theorems 2.9 and 2.13.*

Proof. In the first sampling step, we are free to choose an arbitrary relative error tolerance ε_0. The running time of this stage is $\mathcal{O}\big(\beta \log(p)/(p\varepsilon_0^2) \cdot \log(n/\delta) + \alpha \log n\big)$ by Theorem 2.9. We

then estimate k^* by substituting the high-probability bound $\mathbf{E}[\hat{s}_k] \geq \hat{s}_k - \sqrt{2\hat{s}_k \ln(1/\delta)}$ (w.h.p.) for its expected value in Lemma 2.14 (note that as $\hat{s}_k$ increases with growing sample size and thus grows with falling ε_0, the precision of this bound increases). In the second stage, we can calculate the required value of ε from k^* by solving the expression for k^* in the proof of Theorem 2.13 for ε, and obtain $\varepsilon = 1/k^* \cdot \sqrt{2\log(p)/p \cdot \log(n/\delta)}$. Since $k^* \in \mathcal{O}(k)$ by assumption, the second stage's running time is as in Theorem 2.13. In sum, the algorithm requires the claimed running time. □

Zipfian Inputs. We now consider a concrete distribution, namely inputs distributed according to Zipf's law (see Section 2.1). The objects in such an input have a power-law frequency distribution and fulfil the gap requirements above. We obtain the following result:

Theorem 2.16 (Most Frequent Objects, Algorithm PEC for Power-Law Inputs)
For inputs distributed according to Zipf's law with exponent s, a sample size of $\rho n = 4k^s H_{n,s} \ln \frac{k}{\delta}$ *is sufficient to compute a probably exactly correct result. Algorithm* PEC *then requires expected time* $\mathcal{O}\left(\frac{n}{p} + \beta \frac{\log p}{p} k^s H_{n,s} \log \frac{k}{\delta} + \alpha \log n\right)$ *to compute the k most frequent objects with probability at least* $1 - \delta$.

Proof. Knowing the value distribution, we do not need to take the first sample. Instead, we can calculate the expected value of k^* directly from the proof of Lemma 2.14 and obtain

$$\mathbf{E}[k^*] = k\left(1 - \sqrt{\frac{2\ln\frac{k}{\delta}}{\rho x_k}}\right)^{-1/s}.$$

This immediately yields $\rho > \frac{2}{x_k} \ln \frac{k}{\delta}$, and we choose $\rho = \frac{4}{x_k} \ln \frac{k}{\delta} = 4\frac{k^s H_{n,s}}{n} \ln \frac{k}{\delta}$, ie, twice the required minimum value. This gives us the claimed sample size. Now, we obtain $\mathbf{E}[k^*] = k\left(2 + \sqrt{2}\right)^{1/s}$. In particular, k^* is only a constant factor away from k. Plugging the above sampling probability into the running time formula for our algorithm using exact counting, we obtain the exact top-k most frequent objects with probability at least $1 - \delta$ in the claimed running time. □

Note that the number of frequent objects decreases sharply with $s > 1$, as the k-th most frequent one has a relative frequency of only $\mathcal{O}(k^{-s})$. The $k^s H_{n,s}$ term in the communication volume is thus small in practice, and, in fact, unavoidable in a sampling-based algorithm. We can easily convince ourselves of this by observing that this factor is the reciprocal of the k-th most frequent object's relative frequency. This object needs at least one occurrence in the sample for the algorithm to be able to find it. Thus, the sample size must be in $\Omega(k^s H_{n,s})$.

2.5.4 Refinements

We now discuss potential refinements that might prove worthwhile when implementing such an algorithm in practice.

Choice of k^*. In practice, the choice of k^* in Section 2.5.2 depends on the communication channel's characteristics β, and, to a lesser extent, α, in addition to the problem parameters. Thus, an optimised implementation should take them into account when determining the number of objects to be counted exactly. However, this will likely have to be done empirically.

Using Distributed Bloom Filters. Communication efficiency of the algorithm using exact counting could be improved further by counting sample elements with a distributed single-shot bloom filter (dSBF) [SM13] instead of a distributed hash table. We transmit their hash values and locally aggregated counts. As multiple keys might be assigned the same hash value, we need to determine the element of rank $k^* + \kappa$ instead of k^*, for some safety margin $\kappa > 0$. We request the keys of all elements with higher rank, and replace the (hash, value) pairs with (key, value) pairs, splitting them where hash collisions occurred. Now, we need to determine the element of rank k^* among the $k^* + \kappa + \#\text{collisions}$ elements so obtained. If an element whose rank is at most k^* was part of the original $k^* + \kappa$ elements, we are finished. Otherwise, we have to increase κ to determine the missing elements. Observe that if the frequent objects are dominated by hash collisions, this implies that the input distribution is flat and that numerous nearly equally frequent elements exist. Thus, we may not need to count additional elements in this case.

2.6 Top-k Sum Aggregation

Generalising from Section 2.5, we now consider an input multiset of keys with associated non-negative counts and ask for the k keys whose counts have the largest sums. Again, the input M is assumed to be spread over the p PEs such that no PE has more than $\mathcal{O}(n/p)$ objects. Define $m := \sum_{(k,v)\in M} v$ as the sum of all counts, and let no PE's local sum exceed $\mathcal{O}(m/p)$.[4] Errors are now expressed relative to m. We additionally assume that the local input and a key-aggregation thereof—eg, a hash table mapping keys to their local sums—fit into RAM at every PE.

It is easy to see that except for the sampling process, the algorithms of Section 2.5 carry over directly, but a different approach is required in the analysis.

Pseudo-Sampling. Since the items now have weights, the Bernoulli sampling approach used in Section 2.5 is no longer viable. However, we do *not* use its weighted equivalent, Poisson sampling (see Section 3.6, because it would make bounding the algorithm's error much harder. Instead, we use a 'pseudo-sampling' technique that ensures that the difference between an item's expected and actual sample count is bounded in a certain way.

Let s be the desired size of the sample and define $v_{\text{avg}} := m/s$ as the expected count required to yield a sample. When sampling an object (k, v), its expected sample count is thus v/v_{avg}. To

[4] This assumption is not strictly necessary, and the algorithm would still be communication-efficient without it. However, making this assumption allows us to limit the expected number of samples transmitted by each PE as $\mathcal{O}(s/p)$ for global sample size s. To violate this criterion, a small number of PEs would have to hold $\Omega(s/p)$ elements that are likely to yield at least one sample, making up for a large part of the global sample size without contributing to the result. In such a rare setting, we would incur up to a factor of p in communication volume and obtain running time $\mathcal{O}\left(\frac{n}{p} + \beta\frac{\log p}{\varepsilon}\sqrt{p\log\frac{n}{\delta}} + \alpha\log n\right)$.

retain constant time per object, we add $\lfloor v/v_{\text{avg}} \rfloor$ *pseudo-samples* directly, and one additional sample with probability $\rho = v/v_{\text{avg}} - \lfloor v/v_{\text{avg}} \rfloor$ using a Bernoulli trial. Previously, we used a geometric distribution to reduce the number of calls to the random number generator. Now, we need to use its continuous analogue, the exponential distribution, which describes the distance between events occurring independently at a certain fixed rate (in this case, v_{avg}). Thus, because the ρ are already normalised, the total amount of residual sample probability, ie, how much of the value of the ρ, to be skipped before an item is sampled can be computed as $-\ln(\text{rand}())$.

To improve accuracy and speed up exact counting, we aggregate the local input in a hash table and sample the aggregate counts. This allows us to analyse the algorithms' error independent of the input objects' distribution. A direct consequence is that for each key and PE, the number of samples deviates from its expected value by at most 1, and the overall deviation per key $|s_i - \mathbf{E}[s_i]|$ is at most p.

Probably Approximately Correct Algorithms. Sampling is done using local aggregation as described in the paragraph above. From then on, we proceed exactly as in Algorithm *PAC* from Section 2.5.1. We obtain the following theorem:

Theorem 2.17 (Sum Aggregation, Probably Approximately Correct Algorithm)
We can compute an (ε, δ)-approximation of the top-k highest summing items in expected time $\mathcal{O}\left(\frac{n}{p} + \beta\frac{\log p}{\varepsilon}\sqrt{\frac{1}{p}\log\frac{n}{\delta}} + \alpha\log n\right)$.

Proof. The part of an element's sample count that is determined by sampling is the sum of up to p Bernoulli trials $X_1, \ldots, X_p$ with differing success probabilities. Therefore, its expected value μ is the sum of the probabilities, and we can use Hoeffding's inequality (Equation (1.3)) to bound the probability of significant deviations from this value. Let $X := \sum_{i=1}^{p} X_i$. Then,

$$\mathbf{P}[|X - \mu| \geq t] \leq 2e^{-\frac{2t^2}{p}} \quad . \tag{2.2}$$

We now use this to bound the likelihood that an object has been very mis-sampled. Consider an element j with exact sum $x(j)$ and sample sum s_j. For some threshold Δ, consider the element mis-sampled if $\left|x(j) - s_j v_{\text{avg}}\right| \geq \Delta/2$, ie, its estimated sum deviates from the true value by more than $\Delta/2$ in either direction. Thus, we substitute $t = \Delta/(2v_{\text{avg}})$ into Equation (2.2) and bound the result by δ/n to account for all elements. Solving for the expected sample size $s = m/v_{\text{avg}}$, we obtain $s \geq \varepsilon^{-1}\sqrt{2p\ln(2n/\delta)}$.

In total, we require time $\mathcal{O}(n/p)$ for sampling, $\mathcal{O}(\beta s/p \cdot \log p + \alpha \log p)$ for insertion (as no PE's local sum exceeds $\mathcal{O}(m/p)$, none will yield more than $\mathcal{O}(s/p)$ samples in expectation), and $\mathcal{O}(\beta s/p + \alpha \log n)$ for selection. We then obtain the claimed bound as the sum of these components. □

As in Section 2.5, we can use exact summation to obtain a more precise answer. We do not go into details here, as the procedure is nearly identical. The main difference is that a lookup in the local aggregation result now suffices to obtain exact local sums without requiring consultation of the input.

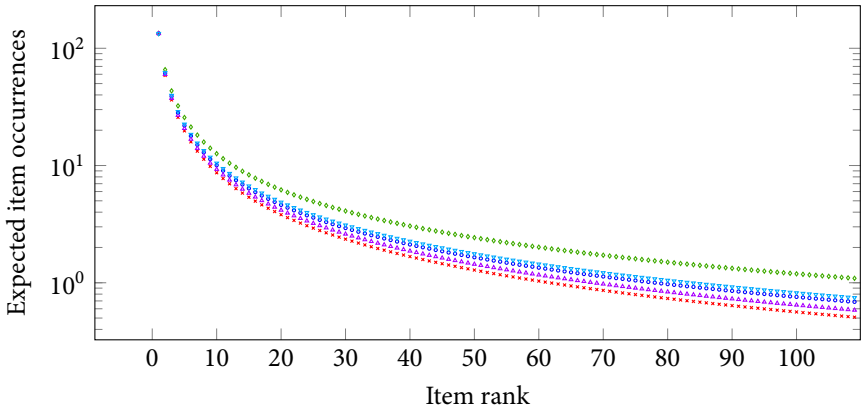


Figure 2.4: Example of the Zipf's law input distribution used for selection for five PEs using 100 object keys and $n = 1000$.

2.7 Experimental Evaluation

We now present an experimental evaluation of the unsorted selection from Section 2.3.1 and the top-k most frequent objects algorithms from Section 2.5.

Experimental Setup. All algorithms were implemented in C++11 using OpenMPI 1.8 and Boost.MPI 1.59 for inter-process communication. Additionally, Intel's Math Kernel Library in version 11.2 was used for random number generation. All code was compiled with the `clang++` compiler in version 3.7 using optimisation level `-Ofast -march=sandybridge`. The experiments were conducted on InstitutsCluster II at Karlsruhe Institute of Technology, a distributed system consisting of 480 computation nodes, of which 128 were available to us. Each node is equipped with two Intel Xeon E5-2670 processors for a total of 16 cores with a nominal frequency of 2.6 GHz, and 64 GiB of main memory. In total, 2048 cores were available to us. An InfiniBand 4X QDR interconnect provides networking between the nodes.

Methodology. We run *weak scaling* benchmarks, which show how wall-time develops for fixed per-PE problem size n/p as p increases. We consider $p = 1$ to 2048 PEs, doubling p in each step. Each PE is mapped to one physical core in the cluster.

2.7.1 Unsorted Selection

Input Generation. Per PE, we consider 2^{24}, 2^{26}, and 2^{28} integer elements. To construct hard instances, we select the k largest—ie, least likely—elements from Zipf distributions (see Section 2.1). To simulate non-randomly distributed data, we use slightly different parameters for the distributions at each PE. At the same time, we limit these differences to ensure that the selection does not become a local operation at any one PE. The number of object keys lies between $2^{20} - 2^{16}$ and 2^{20} elements, with each PE's value chosen uniformly at random. Similarly, the exponent s is uniformly distributed between 1 and 1.2. This ensures that several

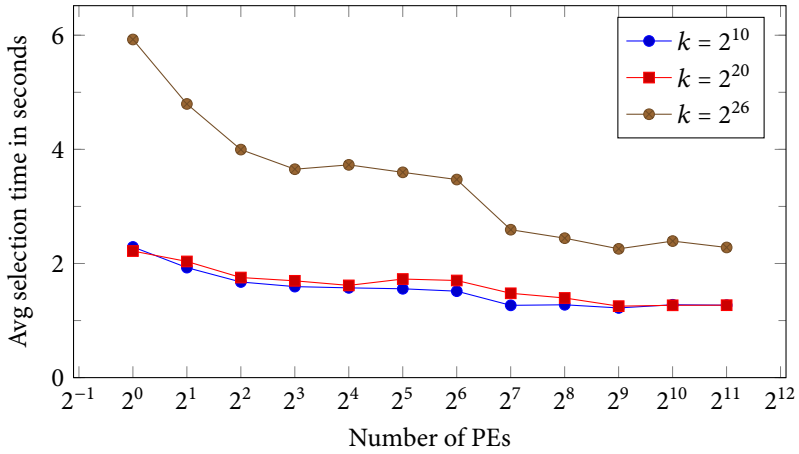


Figure 2.5: Weak scaling plot for selecting the k-th largest object for different values of k. Input size per PE $n/p = 2^{28}$, Zipf distribution.

PEs contribute to the result, so that the distribution is asymmetric, without the computation becoming a local operation at one PE, which has all the largest elements. Figure 2.4 shows an example for $p = 5$ and the first 100 object keys. We used several values of k, namely 1024, 2^{20}, and 2^{26}.

Results. Figure 2.5 shows the results for selecting the k-th largest values from the input, for the above values of k. We can see that in most cases, the algorithm scales even better than the bounds lead us to expect—running time decreases as more PEs are added. This might be caused by decreasing local sample sizes. The effect is especially prominent when selecting an element of high rank ($k = 2^{26}$ in Figure 2.5). The majority of the time is spent in partitioning, ie, local work, dominating the time spent on communication. This underlines the effect of communication efficiency.

2.7.2 Top-k Most Frequent Objects

As we could not find any competitors to compare our methods against, we use two naïve centralised algorithms as a baseline. The first algorithm, *Naïve*, samples the input with the same probability as algorithm PAC, but instead of using a distributed hash table and distributed selection, each PE sends its aggregated local sample to a coordinator. The coordinator then uses quickselect to determine the elements of rank $[1\,..\,k]$ in the global sample, which it returns. Algorithm *Naïve Tree* proceeds similarly but uses a merging tree reduction to send the elements to the coordinator to reduce latency. Similar to Algorithm *PAC*'s hash table insertion operation, this reduction aggregates the counts by key in each step to keep communication volume low.

Input Generation. We consider 2^{24}, 2^{26} and 2^{28} elements per PE, which are generated according to different random distributions. First, we consider elements distributed according to Zipf's

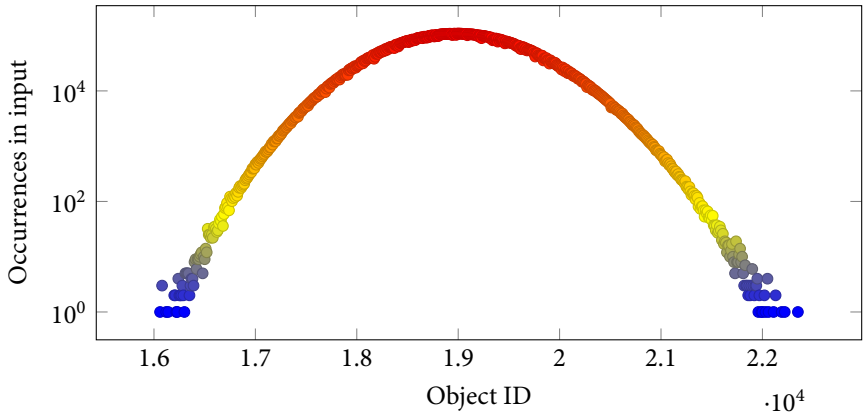


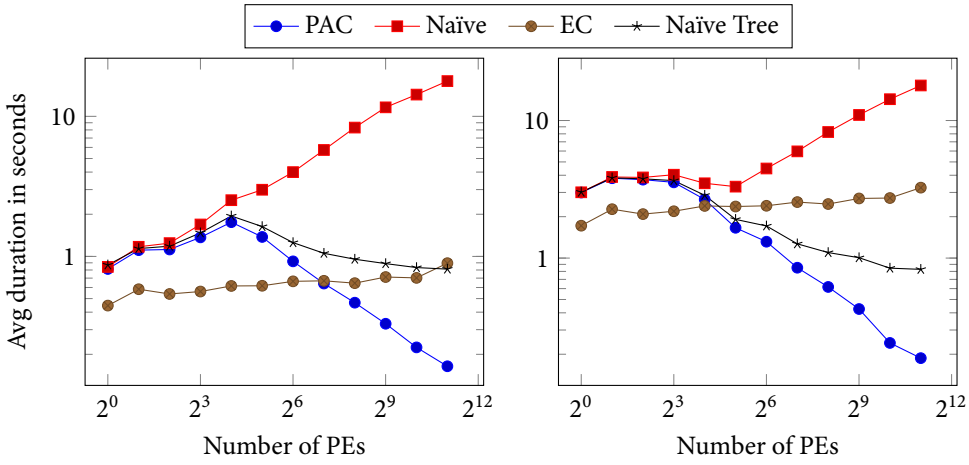
Figure 2.6: Example of the negative binomial distribution used for top-k most frequent items experiments. Each point represents 10 consecutive objects. Distribution parameters: $r = 1000$, $p = 0.05$. Sample of $n = 2^{24}$ items.

Law with 2^{20} possible values. Next, we study a negative binomial distribution with $r = 1000$ and success probability $p = 0.05$. A scatter plot of the true frequency of items in an input of 2^{24} items is shown in Figure 2.6. This distribution has a rather wide plateau, resulting in the most frequent objects and their surrounding elements all being of very similar frequency. For simplicity, each PE generates objects according to the same distribution, as the distributed hash table into which the sample is inserted distributes elements randomly. Thus, tests with non-uniformly distributed data would not add substantially to the evaluation.

Approximation Quality. To evaluate different accuracy levels, we consider the (ε, δ) pairs $(3 \cdot 10^{-4}, 10^{-4})$ and $(10^{-6}, 10^{-8})$. Recall that this requires the result to be within a relative error of ε with probability at least $1 - \delta$. This allows us to evaluate running time under different accuracy requirements.

We then select the $k = 32$ most frequent elements from the input according to the above requirements. We do not vary the parameter k here, as it has very little impact on overall performance. Instead, we refer to Section 2.7.1 for experiments on unsorted selection, which is the only subroutine affected by increasing k and shows no increase up to $k = 2^{20}$.

Results. Figure 2.7 shows the results for 2^{28} elements per PE using $\varepsilon = 3 \cdot 10^{-4}$ and $\delta = 10^{-4}$. We can clearly see that Algorithm *Naïve* does not scale beyond two nodes at all ($p > 32$). Its running time is directly proportional to p, which is consistent with the coordinator receiving $p - 1$ messages—every other PE sends its key-aggregated sample to the coordinator. Algorithm *Naïve Tree* fares better and actually improves as more PEs are added. This is easily explained by the reduced sample size per PE as p increases, decreasing sampling time. However, communication time begins to dominate, as the decrease in overall running time is nowhere near as strong as the decrease in local sample size. This becomes clear when comparing it to Algorithm *PAC*, which outperforms *Naïve Tree* for any number of PEs. We can see that it scales



(a) $n/p = 2^{26}$ (b) $n/p = 2^{28}$

Figure 2.7: Weak scaling plot for computing the 32 most frequent objects with error parameters $\varepsilon = 3 \cdot 10^{-4}$, $\delta = 10^{-4}$ and per-PE input size $n/p = 2^{26}$ (top) and $n/p = 2^{28}$ (bottom). EC suffers from constant overhead for exact counting.

nearly perfectly—doubling the number of PEs (and thereby total input size) roughly halves running time. Since these three algorithms all use the same sampling rate, any differences in running time are entirely due to time spent on communication.

Lastly, let us consider Algorithm *EC* (exact counting, see Section 2.5.2). In the beginning, it benefits from its much smaller sample size, but as p grows, the local work for exact counting clearly dominates overall running time, and Algorithm EC is no longer competitive. Since local work remains constant with increasing p, we see nearly no change in overall running time. To see the benefits of Algorithm *EC*, we need to consider stricter accuracy requirements.

In Figure 2.8, we consider approximation bounds $\varepsilon = 10^{-6}$ and $\delta = 10^{-8}$. For Algorithms *PAC*, *Naïve*, and *Naïve Tree*, this requires considering the entire input for any number of PEs, as the sample size is proportional to $1/\varepsilon^2$, which the other terms cannot offset here. Conversely, Algorithm *EC*'s sample size depends only linearly on ε, resulting in sample sizes orders of magnitude below those of the other algorithms.

Again, we can see that Algorithm *Naïve* does not scale at all. Algorithm *Naïve Tree* performs much better, with running times remaining roughly constant at around 6.5 seconds as soon as multiple nodes are used. This is the case as objects are aggregated by key in each level of the reduction tree, keeping the amount of data per level approximately constant, as each PE observes most objects with high probability. Algorithm *PAC* suffers a similar fate but is slightly faster at 6.2 seconds. This difference stems from reduced communication volume by not concentrating the entire sample at one PE. Instead, it is spread equally over all PEs by using a distributed hash table. However, both are dominated by the time spent on locally aggregating the input. Lastly, Algorithm *EC* is consistently fastest, requiring 4.1 seconds, of which 3.7

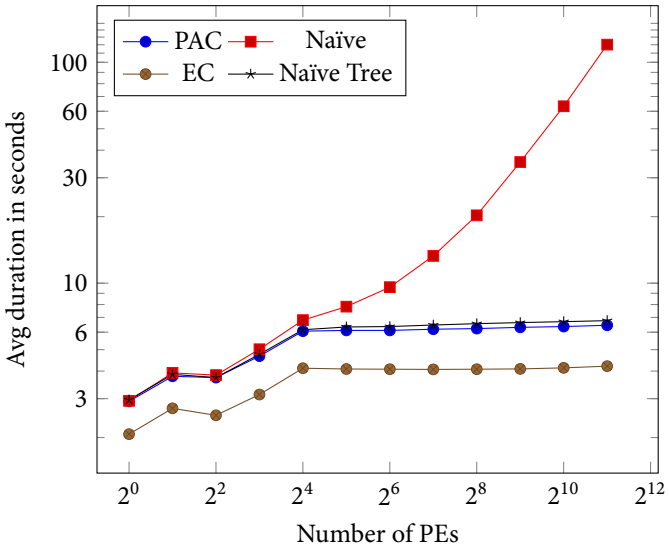


Figure 2.8: Weak scaling plot for computing the 32 most frequent objects, $n/p = 2^{28}$, $\varepsilon = 10^{-6}$, $\delta = 10^{-8}$. Only EC can use sampling, other methods must consider all objects.

seconds are spent on exact counting.[5] This clearly demonstrates that Algorithm *EC* is superior for small ε.

Smaller local input sizes do not yield significant differences, and preliminary experiments with elements distributed according to a negative binomial distribution proved unspectacular and of little informational value, as the aggregated samples have much fewer elements than in a Zipfian distribution—an easy case for selection. Surprisingly, Zipfian distributions represent hard inputs do to their long tail.

2.8 Conclusions

We have demonstrated that a variety of top-k selection problems can be solved communication-efficiently and in a highly scalable manner. The basic methods are simple and versatile: the owner-computes rule, collective communication, and sampling. Considering the significant previous work on some of these problems, it is a bit surprising that such simple algorithms give improved results for such fundamental problems. However, it seems that the combination of communication efficiency and parallel scalability has been neglected for many problems. Our methods might have a particular impact on applications where previous work has concentrated on methods with a pure server–helper scheme.

[5] When not all cores are used, memory bandwidth per core is higher. This allows faster exact counting for $p = 1$ to 8 cores on a single node. Local aggregation in the other algorithms similarly benefits from this.

There are also some open questions. For example, for the sorted selection problem of Section 2.3.2, it would be interesting to see whether there is a scalable parallel algorithm that makes an information-theoretically optimal number of comparisons as in the sequential algorithm of Varman et al. [Var+91]. It could also be interesting to analyse our algorithm for the case where $\overline{k} - \underline{k} = o(\overline{k})$, which we have ignored in this dissertation.

Chapter 3

3 Weighted Sampling

Data structures for efficient sampling from a set of weighted items are an important building block of many applications. However, few parallel solutions are known. We close many of these gaps both for shared-memory and distributed-memory machines. We give efficient, fast, and practicable algorithms for sampling single items, k items with/without replacement, subsets (Poisson sampling), and data streams (reservoir sampling). We also give improved sequential algorithms for alias table construction and sampling with replacement. Experiments on shared-memory parallel machines with up to 158 threads show near-linear speedups both for construction and queries. A distributed-memory evaluation of weighted reservoir sampling on up to 256 nodes (5120 processors) also shows good speedups.

Motivation. Weighted random sampling (WRS) asks for sampling items (elements) from a set such that the probability of sampling item i is proportional to a given weight w_i. Several variants of this fundamental computational task appear in a wide range of applications in statistics and computer science, eg, for computer simulations, data analysis, database systems, and online ad auctions (see, eg, Refs. [MR95; OR95]). Continually growing data volumes ('Big Data') imply that the input sets and even the sample itself can become large. Since the application processing the sample is often fast, sampling algorithms can easily become a performance bottleneck. Given the developments described in the introduction, we require parallel algorithms in shared and distributed memory. In *streaming algorithms*, the input may even become so large that each input element can be inspected only once and cannot be stored for later processing. However, there has been surprisingly little work on parallel weighted sampling. Here, we close many of these gaps.

References. This chapter is based on two conference papers [HS19d; HS20a] and a journal paper [HS20b] published jointly with Peter Sanders. The papers were mainly written by the author of this dissertation, save for the section on weighted random permutations, which is the work of Peter Sanders and not included in this dissertation. The proof of Lemma 3.7 is also due to Peter Sanders and included here for self-containedness. Large parts of this chapter were copied verbatim from the conference papers and the corresponding technical reports [HS19c; HS19a], which contain additional detail not present in the conference versions. The graph generation application in Section 3.9 is partially copied verbatim from the journal paper on this topic, Ref. [HS20b]. The author of this dissertation is also the author of all implementations and conducted all evaluations.

Problems Considered. Without loss of generality, let the input A consist of the items $[1 \,..\, n]$. Item i has weight $w_i \in \mathbb{R}_+$, and define $W := \sum_{i=1}^{n} w_i$ as the sum of all items' weights. An

Table 3.1: Summary of notation used in this chapter.

Symbol	Meaning
A, n	input $A = [1..n]$ of size n
k	sample size
s	number of unique items in a sample with replacement
w_i	weight of item i
W	total weight $W := \sum_{i=1}^{n} w_i$
$w_{\min}, w_{\max}$	smallest and largest weight in the input
U, u	range of weights $U := w_{\max}/w_{\min}$ and $u := \log U$
b	batch size per PE (reservoir sampling)
p	number of PEs
α, β	communication characteristics of the machine model, see Section 1.2.3

item's *relative weight* is its share of the total weight, w_i/W. The notation used in this chapter is summarised in Table 3.1. We consider the following weighted sampling problems:

WRS–1: Weighted sampling of *one item* X from a categorical (or multinoulli) distribution (equivalent to WRS–R and WRS–N for $k = 1$), ie, $\mathbf{P}[X = i] = w_i/W$.

WRS–R: Sample k items from A *with replacement*, ie, the samples are independent and, for each sample X, $\mathbf{P}[X = i] = w_i/W$. Let $s \le k$ denote the number of *different* items in the sample S. Note that we may have $s \ll k$ for skewed input distributions.

WRS–N: Sample k pairwise unequal items $s_1 \neq \cdots \neq s_k$ *without replacement* such that for any sample index $j \in [1..k]$ and item $i \notin \{s_\ell \mid \ell < j\}$, $\mathbf{P}[s_j = i] = w_i/(W - \sum_{\ell<j} w_{s_\ell})$.[1]

WRP: Permute the elements with the same process as for WRS–N using $k = n$ (in Ref. [HS19d]).

WRS–P: Sample a *subset* $S \subseteq A$ where $\mathbf{P}[i \in S] = w_i \le 1$. Also known as Poisson sampling.

WRS–B: *Batched reservoir sampling.* Repeatedly solve WRS–N when batches of new items arrive. Only the current sample and batch may be stored. Let b denote the batch size.

Results. Table 3.2 summaries our results when we process the input set on p processing elements (PEs). When applicable, our algorithms build a data structure once which is later used to support fast sampling queries. In a nutshell, the results say that we can solve the problems with (optimal) linear work and logarithmic (or even constant) latency. The exact bounds depend on the specific machine model used; see Section 3.1.1 for details. More complicated

[1] Observe that some publications use a different definition of WRS–N where the items' relative weights determine their probabilities of being *included in* the sample, rather than those of being chosen in some step. Using that definition, items with relative weight $w_i/W > 1/k$ are *infeasible* because their probability of being included in the sample exceeds one, and such items require special consideration (see, eg, Ref. [Efr15, Example 2]). We henceforth refer to this definition as *weighted sampling with defined probabilities*. The definition we use here does not suffer from this problem.

Table 3.2: Result overview (expected and asymptotic). Distributed results assume random distribution of inputs for concise presentation. Parameters as in Table 3.1: input size n, output size s, sample size k, number of PEs p, startup latency of point-to-point communication α, time for communicating one machine word β, log-weight ratio $u = \log U = \log w_{\max}/w_{\min}$, mini-batches of b items per PE. The complexity of sorting n integers from $[0\,..\,x]$ is $\text{isort}_x(n)$ (isort^* = parallel, isort^1 = sequential).

		Shared Memory (PRAM)				Distributed Memory		
		Preprocessing		Query		Preprocessing	Query	
Problem	**§**	**Work**	**Span**	**Work**	**Span**	**§**	**Time**	**Time**
WRS–1	3.3.2	n	$\log n$	1	1	3.3.3	$\frac{n}{p} + \alpha \log p$	α
WRS–R	3.4	$\text{isort}^*_u(n)$		$s + \log n$	$\log n$	3.4.1	$\text{isort}^1_u\left(\frac{n}{p}\right) + \alpha \log p$	$\frac{s}{p} + \log p$
WRS–N	3.5	$\text{isort}^*_u(n)$		$k + \log n$	$\log n$	3.5.1	$\text{isort}^1_u\left(\frac{n}{p}\right) + \alpha \log p$	$\frac{k}{p} + \alpha \log n \log p$
WRS–N						3.5.1	$\frac{n}{p} + \beta u + \alpha \log p$	$\frac{k}{p} + \log u + \alpha \log n$
WRP	[HS19d]	—	—	$\text{isort}^*_{n(u+\log n)}(n)$		[HS19d]	—	$\text{isort}^*_{n(u+\log n)}(n)$
WRS–P	3.6	n	$\log n$	$s + \log n$	$\log n$	3.6.1	$\frac{n}{p} + \log p$	$\frac{s}{p} + \log p$
WRS–B	—	—	—	—	—	3.7	—	$(b+1)\log(b+k) + \alpha \log k \log p$

formulae stem from the fact that some of the results reduce subproblems to sorting sets of integers. Integer sorting can be solved with near-linear work and near logarithmic latency. However, the bounds now also depend on the range of the involved keys. This range, in turn, depends on the range of occurring item weights. To quantify that, we define $u := \log U$ where $U := w_{\max}/w_{\min} := \max_i w_i / \min_i w_i$. We abstract from the machine model by using black-box formulae for the complexity of integer sorting. Neither competitive parallel algorithms nor more efficient sequential algorithms were previously known. The distributed algorithms are refinements of the shared-memory algorithms with the goal of reducing communication costs compared to direct distributed implementations. As a consequence, each PE mostly works on its local data (the owner-computes approach). Communication—if at all—is only performed to coordinate the PEs and is sublinear in the local work except for extreme corner cases. The owner-computes approach introduces the complication that differences in local work introduce additional parameters into the analysis that characterise the local work in different situations. Therefore, the distributed part of Table 3.2 covers the case when items are randomly assigned to PEs. This simplifies the exposition and is an approach that one can sometimes also take in practice.

We strive to describe our parallel algorithms in a model-agnostic way, ie, we largely describe them in terms of standard operations such as prefix sums for which efficient parallel algorithms are known on various models of computation. We analyse the algorithms for two simple models of computation, the CREW PRAM and our distributed message-passing model (see Section 1.2).

Overview. First, in Section 3.1, we review some known techniques that we are building on. We discuss additional related work in Section 3.2. In Section 3.3, we consider problem WRS–1. We first give an improved sequential algorithm for constructing *alias tables*, the most widely used data structure for problem WRS–1 that allows sampling in constant time. Then we parallelise that algorithm for shared memory and describe adaptations for distributed memory.

Sampling k items with replacement (problem WRS–R) seems to be trivially parallelisable with an alias table. However, using a distributed alias table does not lead to a communication-efficient distributed algorithm. For skewed input distributions where the number of *distinct* output elements s can be much smaller than k, we can generally do better than repeatedly sampling a single item. Section 3.4 develops such an algorithm which is interesting both as a parallel and as a sequential algorithm. The algorithm combines three previous techniques and requires a nontrivial analysis. In Section 3.4.1, we show how communication-efficient construction and communication-free queries can be achieved in distributed memory.

Section 3.5 employs the algorithm for problem WRS–R to solve problem WRS–N. The main difficulty here is to estimate the right number of samples with replacement to obtain a sufficient number of distinct samples. Then an algorithm for WRS–N without preprocessing is used to reduce the 'weighted oversample' to the desired exact output size.

For subset sampling (problem WRS–P), we parallelise the approach of Bringmann and Panagiotou [BP17] in Section 3.6. Once more, the preprocessing requires integer sorting. However, only $\mathcal{O}(\log n)$ different keys are needed so that linear work sorting works with logarithmic latency even deterministically on an EREW PRAM.

In Section 3.7, we adapt the sequential streaming algorithm of Efraimidis and Spirakis [ES06] to a distributed setting where items are processed in small batches. This can be done in a communication-efficient way using our previous work on bulk-parallel priority queues in Section 2.4.

Section 3.8 gives a detailed experimental evaluation of our algorithms for WRS–1 and WRS–R. Section 3.10 summarises the results and discusses possible future directions.

3.1 Preliminaries

We quickly (re-)state the most important pieces of notation used in this chapter (see also Table 3.1). The input is a finite set A of n items with associated positive weights $w_1, \dots, w_n$. Without loss of generality we will assume $A = [1 \,..\, n]$. Define $W := \sum_{i=1}^{n} w_i$ as the sum of all weights. If the weights define a categorical probability distribution, we will have $W = 1$. Let $w_{\max} := \max_i w_i$ and $w_{\min} := \min_i w_i$. Then $U := w_{\max}/w_{\min}$ is the maximum ratio between any two weights, and define $u := \log U$.

3.1.1 Parallel Integer Sorting

We need one basic toolbox operation where the concrete machine model has some impact on the complexity. Sorting n items with integer keys from $[1 \,..\, K]$ can be done with linear work in many relevant cases. Sequentially, this is possible if K is polynomial in n (radix sort). Radix sort can be parallelised even on a distributed-memory machine with linear work and span n^ε for any constant $\varepsilon > 0$ (eg, Ref. [BD18]). Logarithmic span is possible for $K \in \mathcal{O}(\log^c n)$ for any constant c, even on an EREW PRAM [RR89, Lemma 3.1]. For a CRCW PRAM, linear work and logarithmic span can be achieved with high probability when $K \in \mathcal{O}(n \log^c n)$ [RR89] (the paper gives the constraint $K = n$ in its Theorem 3.1 but the generalisation is already implicitly proven in the paper: simply increase the number of most significant bits sorted by *Fine_Sort* from $3 \log\log n$ to $(3 + c + \mathrm{o}(1)) \log\log n$, which does not change the algorithm's asymptotic running time [RR89, Lemma 3.3]). Resorting to comparison-based algorithms, we get work $\mathcal{O}(n \log n)$ and $\mathcal{O}(\log n)$ span on an EREW PRAM [Col88].

3.1.2 Bucket-Based Sampling

The basic idea behind several solutions to problem WRS–1 is to build a table of $m \in \Theta(n)$ buckets where each bucket represents a total weight of W/m. Sampling then selects a random bucket uniformly at random and uses the information stored in the bucket to determine the actual item. If item weights differ only by a constant factor, we can simply store one item per bucket and use rejection sampling to obtain constant expected query time (see, eg, Refs. [Dev86; OR95]).

Deterministic sampling with only a single memory probe is possible using Walker's alias table method [Wal77], and its improved construction due to Vose [Vos91]. An alias table consists of $m := n$ buckets of size W/n where bucket $b[i]$ represents some part w_i' of the weight of item i. The remaining weight of the heavier items is distributed to the remaining capacity of

Algorithm 3.1 : Classical construction of alias tables similar to Vose's approach [Vos91].

Input : $\langle w_1, \ldots, w_n \rangle \in \mathbb{R}^n$ the weights of the n input items
Output : b, an alias table consisting of n pairs (w, a) of (partial) weight w and alias a

```
1  Function voseAliasTable(⟨w_1, ..., w_n⟩)
2      W := Σ_{i=1}^{n} w_i                                   — total weight
3      h := {i ∈ [1..n] | w_i > W/n} as stack                  — heavy items
4      ℓ := {i ∈ [1..n] | w_i ≤ W/n} as stack                  — light items
5      b := ⟨(w_1, ⊥), (w_2, ⊥), ..., (w_n, ⊥)⟩   — init table with item weights, dummy aliases
6      while h ≠ ∅ do                                         — consume heavy items
7          j := h.pop()                                       — get a light item
8          while b[j].w > W/n do                              — still heavy
9              i := ℓ.pop()                                   — get a light item
10             b[i].a := j                                    — Fill bucket b[i] with a...
11             b[j].w := (b[j].w + b[i].w) − W/n               — ...piece of item j.
12         ℓ.push(j)                                          — Item j is light now
13     return b
```

the buckets such that each bucket only represents one other item (the *alias* a_i). Algorithm 3.1 gives high-level pseudocode for the approach proposed by Vose, storing w_i' and a_i as $b[i].w$ and $b[i].a$, respectively. The items are first classified into light and heavy items. Then the heavy items are distributed over light items until their residual weight drops below W/n. They are then treated in the same way as light items.

To sample an item from an alias table, pick a bucket index r uniformly at random, toss a biased coin that comes up heads with probability $w_r' \cdot n/W$, and return item r for heads or its alias a_r for tails.

3.1.3 Weighted Sampling using Exponential Variates

It is well-known that a uniform sample without replacement of size k out of n items $[1\,..\,n]$ can be obtained by associating with each item a uniform variate from the interval $(0,\ 1]$, $v_i := \text{rand}()$, and selecting the k items with the smallest associated variates v_i (eg, Refs. [Sun77; FMR62]). This method can be generalised to generate a *weighted* sample without replacement by raising uniform variates to the power of the inverse of the items' weights, that is, $v_i := \text{rand}()^{1/w_i}$, and selecting the k items with the *largest* associated values v_i [ES99; ES06; Efr15]. Equivalently, one can generate *exponentially distributed random variates* with the items' weights as the distribution's rate parameter, $v_i := -\ln(\text{rand}())/w_i$, and select the k items with the *smallest* associated v_i [Arr02] (sometimes called the '*exponential clocks method*'). The difference between the two is a simple $x \mapsto -\ln(x)$ mapping, with the sign inversion necessitating the switch from selecting the largest values to the smallest ones. Since generating exponential random variates is numerically more stable than computing arbitrary powers, as well as being easier to generate in a vectorised fashion, the latter method should be preferred over the former.

3.1.4 Divide-and-Conquer Sampling

Uniform sampling with and without replacement can be done using a divide-and-conquer algorithm [San+18]. To sample k out of n items uniformly and with replacement, split the set into two subsets with n' (left) and $n - n'$ (right) items, respectively. Then the number of items k' to be sampled from the left has a binomial distribution (k Bernoulli trials, each with success probability n'/n) for sampling with replacement, and a hypergeometric one (number of successes in k draws without replacement from a population of size n with n' success states) for sampling without replacement. We can generate k' accordingly and then recursively sample k' items from the left and $k - k'$ items from the right. This can be used to construct a communication-free parallel sampling algorithm. We have a tree with p leaves. Each leaf represents a subproblem of size about n/p, ie, a PE's local items. Each PE descends this tree to the leaf assigned to it (time $\mathcal{O}(\log p)$) and then generates the resulting number of samples (time $\mathcal{O}(k/p + \log p)$ with high probability). Different PEs have to draw the same random variates for the same interior node of the tree. This can be achieved by seeding a pseudo-random number generator with an ID of this node.

3.1.5 Selection from Sorted Sequences, Again

Our reservoir sampling algorithm relies on selection from the union of sorted sequences stored locally at the PEs. We describe several algorithms for this problem in Section 2.3.2. However, in the specific setting used in this chapter, other approaches may be useful as well. Hence, we provide a quick recapitulation of the most relevant options. In the remainder of this chapter, we will use T_{sel} as a placeholder for the running time of an appropriate selection operation as defined below for various cases. Let each PE hold a sorted sequence of items, and let g be an upper bound on the number of local items at any of the p PEs. We wish to select the item with global rank r (ie, the item with the globally r-th smallest key).

If the output rank r is allowed to vary in some range $[\underline{r}\,..\,\overline{r}]$, we can use algorithm amsSelect of Section 2.3.2.b). This takes time $T_{sel} \in \mathcal{O}(d \log \min(g, r) + \beta d + \alpha \log p)$ in expectation for a tuning parameter $d \in \mathbb{N}$ if $\underline{r}$ and $\overline{r}$ are far enough apart. Specifically, it requires $\overline{r} - \underline{r} \in \Omega(\overline{r}/d)$ (see Theorems 2.5 and 2.6).

If the output rank is fixed, we can still use amsSelect by supplying it with exact bounds, ie, $\overline{r} = \underline{r} = r$. This case is analysed in Section 2.3.2.c), and Theorem 2.7 shows that in expectation, we have $T_{sel} \in \mathcal{O}\big(\log^2 r + \alpha \log r \log p\big)$. Of course, it is also possible to adapt a selection algorithm for unsorted inputs. With high probability, the algorithm of Section 2.3.1 runs in time $T_{sel} \in \mathcal{O}(\log^2 g/\log p + \beta \min(g, \sqrt{p}(1 + \log_p g)) + \alpha \log(gp))$ when executed on sorted sequences (the local work improvement over Theorem 2.1 comes from the fact that partitioning is now selection on the local sequences, which takes $\mathcal{O}(\log g)$ time in each of the $\mathcal{O}(\log_p(g))$ levels of recursion). If the input is randomly distributed (eg, if all input items are independently drawn from a common distribution), then its communication volume improves because we no longer need to spread the sample (Corollary 2.2). Thus, we obtain running time $T_{sel} \in \mathcal{O}(\log^2 g/\log p + \alpha \log(gp))$ w.h.p. for this case.

3.2 Related Work

Weighted sampling is related to the more general field of sampling with unequal probabilities, much of which focuses on statistical estimation rather than sampling from categorical distributions. For a comprehensive overview of the mathematical side and sequential algorithms for numerous problems related to sampling with unequal probabilities, we refer to the monographs of Brewer and Hanif [BH83] and Tillé [Til06]. For the most part, the methods surveyed therein focus on estimating the total of a measure that is assumed to correlate with the items' weights. Among the most prominent results in this area are Refs. [HH43; HT52], and early algorithms for computing weighted samples *with defined probabilities* (see footnote on page 52), eg, Refs. [HR62; RHC62] are based on this line of research. Here, we focus on the literature on (efficiently) computing various kinds of samples from categorical distributions.

Sampling one Item (Problem WRS–1). Extensive work has been done on generating discrete random variates from a fixed categorical distribution [Wal77; Vos91; MTW04; Dev86; BP17]. All these approaches use preprocessing to construct a data structure that subsequently supports very fast (constant time) sampling of a single item. The best-known approach is the alias method [Wal77; Vos91], which computes in linear time a data structure that allows solving problem WRS–1 in constant time, *cf.* Section 3.1.2. Another well-known approach, which does not require precomputation beyond finding the weight of the heaviest item, is to use an acceptance/rejection method [OR95; Rub81]. This works well as long as the weights are not too different, ie, U is small, or the distribution has to be updated very frequently. Otherwise, the trade-off of constant update time at the cost of expected time $\mathcal{O}(U)$ for sampling an item is unfavourable. Bringmann and Larsen [BL13] explain how to achieve expected time r using only $\mathcal{O}(n/r)$ bits of space beyond the input distribution itself. The theoretical literature on generating discrete random variables culminates in two papers on maintaining data structures that allow both sampling single item and updating items' weights in constant time [HMM93; MVN03], which subsumes both WRS–R and WRS–N. In the case of polynomially bounded integer weights, Hagerup et al. [HMM93] achieve constant worst-case time for both operations. Recently, Berenbrink et al. [Ber+20] presented dynamic alias tables, an engineered data structure that allows efficient weight updates for integer weights.

Sampling Without Replacement (Problem WRS–N). The exponential clocks method, described in Section 3.1.3 (see also [ES99; ES06; Efr15; Arr02; CK07]), is a simple $\mathcal{O}(n)$ algorithm for problem WRS–N. This approach also lends itself towards use in streaming settings *(reservoir sampling)* and can be combined with a skip value distribution to reduce the number of required random variates from $\mathcal{O}(n)$ to $\mathcal{O}\left(k \log \frac{n}{k}\right)$ in expectation [ES06]. Cohen and Kaplan [CK07] mention that this approach can also be used in the model of *bottom-k sketches*. A related algorithm for WRS–N with given inclusion probabilities instead of relative weights is described by Chao [Cha82]. Braverman et al. [BOV15] present another sequential algorithm using a reduction to sampling *with* replacement *(cascade sampling)*.

More efficient algorithms for WRS–N repeatedly sample an item and remove it from the distribution using a dynamic data structure [WE80; OR95; HMM93; MVN03]. With the most efficient such algorithms [HMM93; MVN03] we achieve time $\mathcal{O}(k)$, albeit at the price of an

inherently sequential and rather complicated algorithm that might have considerable hidden constant factors.

It is also possible to combine techniques for sampling *with* replacement with a rejection method. However, the performance of these methods depends heavily on U, the ratio between the largest and smallest weight in the input, as the rejection probability rises steeply once the heaviest items are removed. Lang [Lan14] gives an analysis and experimental evaluation of such methods for the case of $k = n$. A recent practical evaluation of approaches that lend themselves towards efficient implementation is due to Müller [Mül16].

Poisson Sampling (Problem WRS–P). Poisson sampling [Háj64] is often used in statistical estimation, eg, the U.S. Census Bureau's Annual Survey of Manufactures [US 20]. It also has applications in graph generation, such as generating Chung-Lu random graphs [CL03]. Refer to Ref. [Til06] for an overview of methods used for statistical purposes. Tsai et al. [Tsa+10] present a sequential algorithm with suboptimal query time, which Bringmann and Panagiotou [BP17] improve upon to give an optimal sequential algorithm by a reduction to integer sorting. In computer science literature, Poisson sampling is sometimes also called *subset sampling*, with equivalent definitions.

Parallel Sampling. There is surprisingly little work on parallel sampling. Even uniform (unweighted) sampling had many loose ends until recently [San+18]. Efraimidis and Spirakis [ES99] note that WRS–N can be solved in parallel with span $\mathcal{O}(\log n)$ and work $\mathcal{O}(n \log n)$. They also note that solving the selection problem suffices if the output need not be sorted. The optimal dynamic data structure for WRS–1 [MVN03] admits a parallel bulk update in the (somewhat exotic) combining-CRCW-PRAM model, where concurrent write operations are *combined* in a Fetch&Add step [Got+83]. This step adds all concurrently written values to the existing contents of the memory location and also records all prefix sums that would occur if this process was to be executed sequentially. Even so, this does not help with problem WRS–N since batch sizes are one.

Some results, however, may be viewed as folklore. For example, once we have a data structure like the alias table (see Section 3.1.2), we can solve problem WRS–R by sampling from this data structure in parallel until the desired sample size has been reached.

Uniform Reservoir Sampling. Sampling from a *stream of data* has been studied since at least the early 1960s [FMR62] and several asymptotically optimal algorithms are known, see, eg, Refs. [Vit85; Li94]. The key insight of these algorithms is that it is possible to determine in constant time how many items to skip before a new item enters the reservoir by computing a random variate from a suitably parametrised geometric distribution [Dev86; Li94].

Sampling from the union of multiple data streams was only studied much more recently [CTW16; Cor+12; TW11; Cor+10]. However, these publications use the continuous monitoring model (see Introduction), and are thus inherently limited in their scalability. Recently, Tangwongsan and Tirthapura [TT19] presented a shared-memory parallel uniform reservoir sampling algorithm in a mini-batch model. Birler et al. [BRN20] propose an engineered version of using synchronised access to a shared source of skip values, based on Li's [Li94] Algorithm L. Their approach is geared towards the specific use case of maintaining a sample of a relation in a database management system to aid query optimisation.

Weighted Reservoir Sampling. Sampling from a stream of *weighted* items has received significantly less attention in the literature. Chao [Cha82] presents a simple and elegant algorithm for weighted reservoir sampling *with defined probabilities* (see footnote on page 52). Efraimidis and Spirakis give an algorithm based on associating a suitably computed key with each item, so that the k items with the largest keys form the desired sample, and also show how to compute skip distances (which they call *exponential jumps*) in constant time [ES06; Efr15]. Braverman et al. [BOV15] present an approach they call *Cascade Sampling* that is not affected by the numerical inaccuracies of floating-point representation in computers by giving an exact reduction to sampling *with* replacement.

The first—and, to the best of our knowledge, only—distributed streaming algorithm for weighted reservoir sampling was published only recently by Jayaram et al. [Jay+19]. Their algorithm is given in the continuous monitoring model, resulting in complex algorithmic challenges and requiring a *level set* construction grouping the items by their weights, withholding level sets with too few items to work around issues with extreme heavy hitters, and a process that maintains correctness in the face of withheld items.

3.3 Alias Table Construction (Problem WRS–1)

In this section, we describe algorithms for sequential (Section 3.3.1), shared-memory parallel (Section 3.3.2), and distributed-memory (Section 3.3.3) construction of alias tables.

3.3.1 Improved Sequential Alias Tables

Before discussing parallel alias table construction, we discuss a simpler, faster and more space-efficient sequential algorithm that is a better basis for parallelisation. Previous algorithms need auxiliary arrays/queues of size $\Theta(n)$ to decide in which order the buckets are filled. Vose [Vos91] mentions that this can be avoided but does not give a concrete algorithm. We now describe an algorithm with this property.

The idea of the algorithm is that two indices i and j sweep the input array for light and heavy items, respectively. The loop invariant is that the weight of items corresponding to light (heavy) items preceding i (j) has already been distributed over some buckets and that their corresponding buckets have already been constructed. Variable w stores the weight of the part of item j that has not yet been assigned to buckets. Each iteration of the main loop advances one of the indices and initialises one bucket. When the residual weight w exceeds W/n, item j is used to fill bucket i, the residual weight w is reduced by $W/n - w_i$, and index i is advanced to the next light item. Otherwise, the remaining weight of heavy item j fits into bucket j and the remaining capacity of bucket j is filled with the next heavy item. Algorithm 3.2 gives pseudocode that emphasises the high degree of symmetry between these two cases. Figure 3.1 gives an example, where Figure 3.1 (a) shows a snapshot of the state of the algorithm after processing 7 of 13 items, and Figure 3.1 (b) shows the resulting alias table.

Algorithm 3.2 : A sweeping algorithm for constructing alias tables.

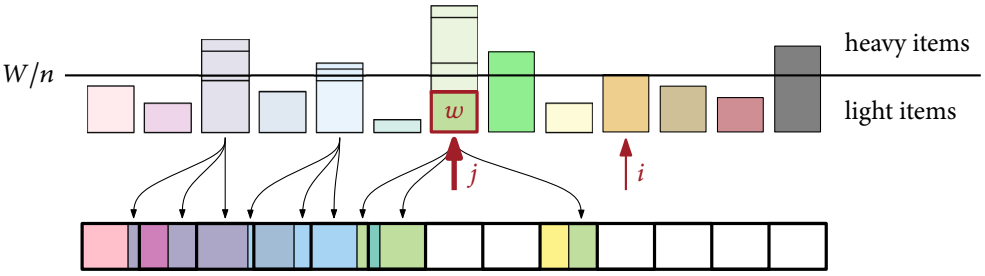
Input : $\langle w_1, \ldots, w_n \rangle \in \mathbb{R}^n$ the weights of the n input items
Output : b, an alias table consisting of n pairs (w, a) of (partial) weight w and alias a

```
1  Function sweepingAliasTable(⟨w_1, ..., w_n⟩)
2      W := Σ_{i=1}^{n} w_i                                  — total weight
3      i := min {k > 0 | w_k ≤ W/n}                          — first light item
4      j := min {k > 0 | w_k > W/n}                          — first heavy item
5      while j ≤ n do
6          if w > W/n then                                   — Pack a light bucket.
7              b[i].w := w_i                                 — Item i completely fits here.
8              b[i].a := j                                   — Item j fills the remainder of bucket i.
9              w := (w + w_i) − W/n                          — Update residual weight of item j.
10             i := min {k > i | w_k ≤ W/n}                   — next light item, assume w_{n+2} = 0
11         else                                              — Pack a heavy bucket.
12             b[j].w := w                                   — Now item j completely fits here.
13             j' := min {k > j | w_k > W/n}                  — next heavy item, assume w_{n+1} = ∞
14             j := b[j].a := j'                             — Proceed with item j'
15             w := (w + w_{j'}) − W/n                       — Compute residual weight
```

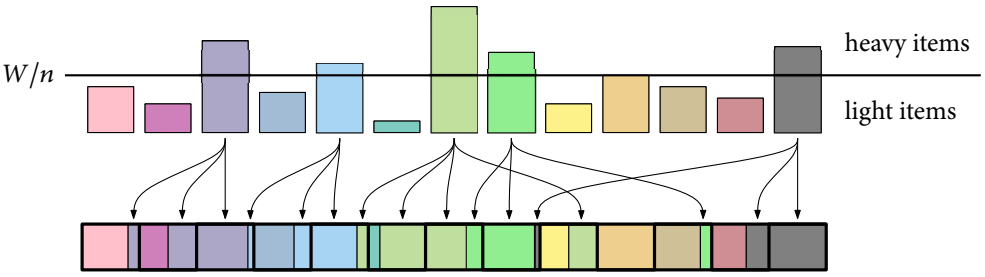
3.3.2 Parallel Alias Table Construction

The basic idea behind our *splitting-based algorithm* is to parallelise the sequential algorithm of the previous subsection by determining its state at carefully selected points during its execution.[2] In the example of Figure 3.1, splitting the computation roughly in half means reconstructing the state of Figure 3.1 (a) without actually executing the sequential algorithm. This state-reconstruction can be done in parallel for $p - 1$ positions in logarithmic time, splitting the work into p packets of equal size, one for each PE. The splitting positions are chosen in such a way that the work performed by the sequential algorithm between two subsequent positions is a $1/p$ share of the total work. This work represents a subproblem that is assigned to one PE. A state is defined by the indices of the next heavy and light element and by how much of the current heavy element remains to be assigned. The resulting subproblems consist of consecutive subsets of light and heavy elements, potentially along with a piece of a heavy item from a preceding subproblem, and/or with a piece of the last heavy item earmarked for one or more subsequent subproblems. The items of these subproblems can be allocated precisely within their respective buckets. We can thus handle the subproblems completely independently in parallel, requiring no further synchronisation or communication. The splitting of a heavy item over two or more subproblems does not complicate this beyond the sequential algorithm:

[2] Since this requires information about all items, this algorithm is primarily targeted at shared-memory machines, and distributed memory is discussed later in Section 3.3.3.



(a) State of the algorithm after packing 7 of 13 buckets.



(b) The finished alias table.

Figure 3.1: Example of alias table construction. Items' weights are given by their boxes' heights, colours are for illustration purposes only. Each bucket contains a piece of its item, aligned to the left in the bucket, and optionally an alias, aligned right and marked with an arrow from its item for illustration.

pieces of it are assigned as aliases of light items in all but the last subproblem where it occurs. Thus, the resulting data structure is still an alias table.

We first explain how to split n items into two subsets of size n' and $n - n'$. Similar to Vose's algorithm, we first compute arrays ℓ and h containing the indices of the light and heavy items, respectively. We then determine indices i and j such that $i + j = n'$ and

$$\sigma := \sum_{x \le i} w_{\ell[x]} + \sum_{x \le j} w_{h[x]} \le n'W/n \quad \text{and} \quad \sigma + w_{h[j+1]} > n'W/n \ .$$

These values can be determined by binary search over the value of j. By precomputing prefix sums of the weights of the items in ℓ and h, each iteration of the binary search takes constant time. The resulting subproblem then consists of the light items $\ell[1\,..\,i]$, the heavy items $h[1\,..\,j]$, and a piece of size $n'W/n - \sigma$ of item $h[j+1]$. See Figure 3.2 for a continuation of the example of Figure 3.1.

To split the input into p independent subproblems of near-equal size, we perform the above two-way-split for the values $n'_k = \lceil nk/p \rceil$ for $k \in [1\,..\,p-1]$. PE k is then responsible for filling a set of buckets corresponding to sets of light and heavy items, each represented by a range of indices into ℓ and h. A piece of a further heavy item may be used to make the calculation work out. Note that a subproblem might contain an empty set of light or heavy items and that

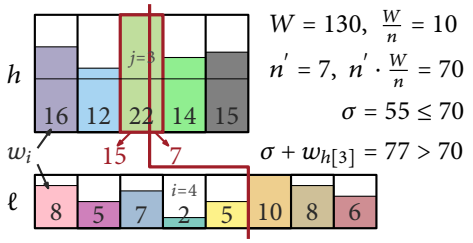


Figure 3.2: Parallel alias table construction: splitting $n = 13$ items into two parts of size $n' = 7$ and $n - n' = 6$. Continuation of the example of Figure 3.1.

a single heavy item j may be assigned partially to multiple subproblems, but only the last PE using a heavy item will fill its bucket.

Algorithm 3.3 gives detailed pseudocode. It uses function *split* to compute $p - 1$ different splits in parallel. The result triple (i, j, s) of *split* specifies that buckets $\ell[1]\cdots\ell[i]$ and $h[1]\cdots h[j]$ shall be filled using the left subproblem. Moreover, total weight s of item $h[j+1]$ is *not used* on the left side, ie, spilt over to the right side.

This splitting information is then used to make p parallel calls to procedure *pack*, giving each PE the task to fill up to $\lceil n/p \rceil$ buckets. *Pack* has input parameters specifying ranges of heavy and light items it should use. The parameter *spill* determines how much weight of item $h[j_{\min} - 1]$ can be used for that. *Pack* works similar to the sweeping algorithm from Algorithm 3.2. If the residual weight of item $h[j_{\min} - 1]$ drops below W/n, this item is also packed in this call. The body of the main loop is dominated by one if-then-else case distinction. When the residual weight of the current heavy item falls below W/n, its bucket is filled using the next heavy item. Otherwise, its weight is used to fill the current light item.

Theorem 3.1 (Parallel Alias Table Construction)

We can construct an alias table with work $\Theta(n)$ and span $\Theta(\log n)$ on a CREW PRAM.

Proof. The algorithm requires linear work and logarithmic span for identifying light and heavy items and for computing prefix sums [Ble89] over them. Splitting works in logarithmic time. Then each PE needs time $\mathcal{O}(n/p)$ to fill the buckets assigned to its subproblem. □

3.3.2.a) Performance Optimisation in Practice

Observe that we can improve the practical performance of this algorithm by greedily assigning ranges of items to the PEs before we begin the splitting, with each PE stopping when it runs out of either light or heavy items in its allotment and applying the above construction only to those items that could not be assigned. By directly assigning some items, the number of items for which the auxiliary data structures of Algorithm 3.3 have to be computed is reduced. However, the number of items that can be handled this way is dependent on the input distribution and may be arbitrarily small. As a result, we must assume that $\mathcal{O}(n)$ items remain. Nonetheless, this optimisation can yield significant improvements for typical inputs (see experiments in Section 3.8.2, where this method is labelled PSA+).

Algorithm 3.3 : SM-parallel splitting-based alias table construction (PSA).

Input : $\langle w_1, \ldots, w_n \rangle \in \mathbb{R}^n$ the weights of the n input items
Output : b, an alias table consisting of n pairs (w, a) of (partial) weight w and alias a

Function psaAliasTable($\langle w_1, \ldots, w_n \rangle$)
  $W := \sum_{i=1}^{n} w_i$ — *total weight*
  $h := \langle i \in [1\,..\,n] \mid w_i > W/n \rangle$ — *parallel traversal finds heavy items…*
  $\ell := \langle i \in [1\,..\,n] \mid w_i \leq W/n \rangle$ — *…and light items*
  for $k := 1$ **to** $p - 1$ **dopar**
    $(i_k, j_k, spill_k) := \text{split}(\lceil nk/p \rceil)$ — *split into p pieces*
  $(i_0, j_0, spill_0) := (0, 0, 0.0)$; $(i_p, j_p) := (n, n)$ — *cover boundaries*
  for $k := 1$ **to** p **dopar** pack$(i_{k-1} + 1, i_k, j_{k-1} + 1, j_k, spill_{k-1})$

Input : $n' \in \mathbb{N}$, the number of buckets to be filled by the left subproblem
Output : $i, j \in \mathbb{N}$, the last light and heavy items to be filled in the left subproblem; $s \in \mathbb{R}$ the partial weight of heavy item $j + 1$ *not* to be used by the left subproblem

Function split(n')
  $a := 1$; $b := \min(n, |h|)$ — *$[a\,..\,b]$ is the search range for j*
  loop — *binary search*
    $j := \lfloor (a + b)/2 \rfloor$ — *bisect search range*
    $i := n' - j$ — *Establish the invariant $i + j = n'$*
    $\sigma := \sum_{x \leq i} w_{\ell[x]} + \sum_{x \leq j} w_{h[x]}$ — *work to the left; use precomputed prefix sums*
    if $\sigma \leq n'W/n$ **and** $\sigma + w_{h[j+1]} > n'W/n$ **then**
      return $(i, j, w_{h[j+1]} + \sigma - n'W/n)$
    if $\sigma \leq n'W/n$ **then** $a := j + 1$ **else** $b := j - 1$ — *narrow search range*

Input : $[i_{\min}\,..\,i_{\max}]$ the range of light items to assign; $[j_{\min}\,..\,j_{\max}]$ the range of heavy items to assign; *spill* the maximum amount of heavy item $j_{\min} - 1$ to use

Procedure pack($i_{\min}, i_{\max}, j_{\min}, j_{\max}, spill$)
  $i := i_{\min}$; $j := j_{\min} - 1$ — *$\ell[i]$ and $h[j]$ are the current light/heavy items,*
  $w := spill$ — *spill is the part of the current heavy item still to be assigned*
  if $spill = 0$ **then** j++; $w := w_{h[j]}$
  loop
    if $w \leq W/n$ **then** — *pack a heavy bucket*
      if $j > j_{\max}$ **then return**
      $b[h[j]] := (w, h[j + 1])$
      $w := (w + w_{h[j+1]}) - W/n$; j++
    else — *pack a light bucket*
      if $i > i_{\max}$ **then return**
      $b[\ell[i]] := (w_{\ell[i]}, h[j])$
      $w := (w + w_{\ell[i]}) - W/n$; i++

More concretely, in a first step, PE i handles items $(i-1)/p+1$ to i/p. In this step, the PEs run the sequential assignment algorithms on their subset of the input, stopping once the next light or heavy item would be beyond their part of the input (ie, the item's index is larger than i/p). Algorithm 3.3 is then only executed on these remaining items. To do so, each PE records the index of the first item that could not be assigned greedily and the number of items in its allotment that was not assigned. The PEs then compute a prefix sum over the latter value. This allows them to compute the global rank of their unfinished items among all unfinished items. This information is required in the index calculations (arrays h and ℓ). Algorithm 3.3 then proceeds normally on these items.

3.3.2.b) Using Block-Wise Prefix Sums

We can further optimise memory usage by employing *block-wise* prefix sums in our parallel alias table construction algorithm. The idea is for each block to represent logarithmically many—ie, $\log(n)$—items for prefix sum computation, resulting in a prefix sum with $n/\log(n)$ entries. We augment this with a prefix sum over the number of heavy items in the blocks. This allows us to replace both the ℓ and h arrays and the full prefix sums over the weight of the light and heavy items. Algorithm 3.4 gives pseudocode for the computation of the required data structures and overall control flow, and Algorithm 3.5 gives pseudocode for splitting the computation. Both use a generalised block size B for clarity.

Consider first the aforementioned prefix sums, which are used to split the input into subproblems. We can now use this coarse prefix sum until we found the block containing the heavy item that is the true splitter. To do this without having to extract blocks, we use a double

Algorithm 3.4 : Parallel alias table construction with block-wise prefix sums.

Input : $\langle w_1, \dots, w_n \rangle \in \mathbb{R}^n$ the weights of the n input items
Output : b, an alias table consisting of n pairs (w, a) of (partial) weight w and alias a

1 **Function** blockAliasTable($\langle w_1, \dots, w_n \rangle$)
2 $W := \sum_{i=1}^{n} w_i$ — *total weight*
3 **for** $i := 1$ **to** $\lceil n/B \rceil$ **dopar**
4 $m := (i-1)B + 1; \quad M := \min(iB, n)$ — *boundaries of block i*
5 $h_i := \left|\left\{ j \in [m\,..\,M] \mid w_j > W/n \right\}\right|$ — *number of heavy items in block i*
6 $W_{\ell,i} = \sum \left\{ w_j \mid j \in [m\,..\,M] \wedge w_j \leq W/n \right\}$ — *total weight of light items of block i*
7 $W_{h,i} = \sum \left\{ w_j \mid j \in [m\,..\,M] \wedge w_j > W/n \right\}$ — *total weight of heavy items of block i*
8 Compute prefix sums of h_i, $W_{\ell,i}$, and $W_{h,i}$ in parallel.
9 **for** $k := 1$ **to** $p-1$ **dopar**
10 $(i_k, j_k, spill_k) :=$ splitBlock($\lceil nk/p \rceil$) — *split into p pieces, see Algorithm 3.5*
11 $(i_0, j_0, spill_0) := (0, 0, 0) \in \mathbb{N} \times \mathbb{N} \times \mathbb{R}$ — *cover boundaries*
12 $(i_p, j_p) := (n, n) \in \mathbb{N} \times \mathbb{N}$
13 **for** $k := 1$ **to** p **dopar** pack($i_{k-1}+1, i_k, \quad j_{k-1}+1, j_k, \quad spill_{k-1}$)

Algorithm 3.5 : Block-wise parallel alias table construction: splitting.

Input : $n' \in \mathbb{N}$, the number of buckets to be filled by the left subproblem, as well as W and the arrays h, W_ℓ, and W_h as computed in Algorithm 3.4; block size B

Output : $i, j \in \mathbb{N}$, the last light and heavy items to be filled in the left subproblem; $s \in \mathbb{R}$ the partial weight of heavy item $j+1$ *not* to be used by the left subproblem

1 **Function** splitBlock(n')

2 $i_{\min} := j_{\min} := 1$; $i_{\max} := j_{\max} := \lceil n/B \rceil$ — *block ranges to search*

3 **while** $i_{\min} + 1 < i_{\max}$ **and** $j_{\min} + 1 < j_{\max}$ **do** — *double binary search on blocks*

4 **invariant:** desired light item is in one of blocks $[i_{\min} .. i_{\max}]$, heavy in $[j_{\min} .. j_{\max}]$.

5 $i := \lfloor (i_{\min} + i_{\max})/2 \rfloor$; $j := \lfloor (j_{\min} + j_{\max})/2 \rfloor$ — *bisect search range*

6 *Now considering light items from blocks* $[1..i]$ *and heavy items from blocks* $[1..j]$.

7 $c_h := h[j]$ — *number of heavy items up to and including block* j

8 $c_\ell := iB - h[i]$ — *number of light items up to and including block* i

9 $R_c := (c_\ell + c_h)/n'$ — *fraction of item count satisfied*

10 $R_w := (W_{\ell,i} + W_{h,j})/(n'W/n)$ — *fraction of target weight satisfied*

11 **if** $R_c = 1$ **and** $R_w = 1$ **then return** $(iB, jB, 0)$

12 **if** $R_w \geq 1$ **and** $R_c \leq 1$ **then** $j_{\max} := j - 1$ — *too much weight with too few items*

13 **if** $R_w \leq 1$ **and** $R_c \geq 1$ **then** $j_{\min} := j + 1$ — *too many items with too little weight*

14 **if** $1 \geq R_w \geq R_c$ **then** — *count deficit that cannot be fixed with more heavy items alone*

15 $i_{\min} := i$ ⇒ *need at least as many light items*

16 **if** $1 \geq R_c \geq R_w$ **then** — *weight deficit that cannot be fixed by more light items alone*

17 $j_{\min} := j$ ⇒ *need at least as many heavy items*

18 **if** $1 \leq R_w \leq R_c$ **then** — *count surplus that cannot be fixed by fewer heavy items alone*

19 $i_{\max} := i$ ⇒ *need at most as many light items*

20 **if** $1 \leq R_c \leq R_w$ **then** — *weight surplus that cannot be fixed by fewer light items alone*

21 $j_{\max} := j$ ⇒ *need at most as many heavy items*

22 Compute prefix sum of weights of light items in blocks $[i_{\min} .. i_{\max}]$ with virtual 0^{th} item of weight $W_{\ell,i_{\min}-1}$; same for heavy items in blocks $[j_{\min} .. j_{\max}]$ and $W_{h,j_{\min}-1}$.

23 Continue search as in split of Algorithm 3.3 with $a := (j_{\min} - 1)B + 1$ and $b := jB$, where i is guaranteed to be in blocks $[i_{\min} .. i_{\max}]$ and j is constrained to blocks $[j_{\min} .. j_{\max}]$ by choice of a, b. Use prefix sums computed above to calculate σ. Use binary search on h array to find the block containing the next heavy item in line 15.

binary search that simultaneously limits the range of possible buckets for light and heavy items. This ensures that in each iteration of the binary search, we can reduce at least one search range, thereby ensuring logarithmic search depth.

The search terminates once we have limited both the number of heavy and light buckets under consideration to at most two. We can therefore 'extract' these blocks, computing their items' prefix sums, and continue the search there using the regular splitting function of Algorithm 3.3.

Theorem 3.2 (Parallel Alias Table Construction with Block-Wise Prefix Sums)
We can construct an alias table with work $\Theta(n)$ and span $\Theta(\log n)$ on a CREW PRAM with $\Theta(n/\log n)$ words of auxiliary memory.

Proof. Algorithm 3.4 is expressed with a general block size B, and we shall see that choosing $B := \log n$ results in the claimed bounds. Computing the auxiliary data structures (lines 3–8 of Algorithm 3.4) can be done with linear work and logarithmic span in $\Theta(n/B)$ words of memory. The remainder of the algorithm uses only constant additional space. Thus, the amount of auxiliary memory used by the algorithm is $\Theta(n/B)$ words.

Consider now the running time of function splitBlock of Algorithm 3.5. In each iteration of the loop, it considers the midpoint of both search ranges and reduces the search range for one of the intervals from (without loss of generality) $i_{\max} - i_{\min} + 1$ to at most $i_{\max} - \lfloor (i_{\max} - i_{\min})/2 \rfloor$. Therefore, after at most a logarithmic number of iterations, both intervals have been reduced to size at most 2. The search for the correct item has therefore been limited to a constant number of blocks, which can be 'unpacked' in time $\mathcal{O}(B)$. Continuing the search on these items using function split of Algorithm 3.3 takes time $\mathcal{O}(\log B)$. Thus, function splitBlock runs in time $\mathcal{O}(\log n + B)$.

To retain logarithmic span in the overall alias table construction algorithm, we choose $B := \log n$ and obtain the claimed bounds. □

These changes should yield a speed increase overall because alias table computation is not limited by computation but by the available memory bandwidth (see Section 3.8). Using this optimisation, we reduce memory use for the auxiliary arrays from $2n$ to $3n/\log n$ words. Including the alias table, this reduces total memory usage by almost half.

Note that we could also use a rank/select data structure, eg, that of Ref. [Shu17], instead of the array h. This data structure requires $n + o(n)$ bits, ie, $\Theta(n/\log n)$ words, and can be constructed in parallel. This change could also be made to Algorithm 3.3 independently.

3.3.3 Distributed Alias Table Construction

The shared-memory parallel algorithm described in Section 3.3.2 can also be adapted to a distributed-memory machine (eg, using PRAM emulation [Ran91]). However, this requires information about all items to be communicated. Hence, more communication-efficient algorithms are important for large n. To remedy this problem, we will now view sampling as a 2-level process implementing the owner-computes approach underlying many distributed algorithms. An illustration of this is shown in Figure 3.3.

Let E_i denote the set of items allocated to PE i. For each PE i, we create a *meta-item* of weight $W_i := \sum_{j \in E_i} w_j$. Sampling now amounts to sampling a meta-item and then delegating the task to sample an actual item from E_i to PE i. The local data structures can be built independently on each PE, possibly using a shared-memory parallel algorithm locally. Additionally, we need to build a data structure for sampling a meta-item. We present two variants of this approach in the following theorem.

Theorem 3.3 (Distributed 2-level Alias Tables)
Assuming that $\mathcal{O}(n/p)$ elements are allocated to each PE, we can sample a single item in time $\mathcal{O}(\alpha)$ after preprocessing a 2-level alias table, which can be done in time $\mathcal{O}(n/p)$ plus the following communication overhead:

i) $\beta p + \alpha \log p$ with replicated preprocessing, or

ii) $\alpha \log p$ with a distributed top-level table permitting two aliases.

Proof. Building the local alias tables takes time $\mathcal{O}(\max_i |E_i|) \subseteq \mathcal{O}(n/p)$ sequentially.

For Theorem 3.3 (i), we can perform an all-gather operation on the meta-items and compute the data structure for the meta-items in a replicated way. All-gathering p items can be implemented in time $\mathcal{O}(\beta p + \alpha \log p)$ [San+19, Chapter 13.5.1], resulting in the claimed running time bound.

For Theorem 3.3 (ii), we use the data redistribution algorithm from the extended version of the paper on which the selection chapter is based, Ref. [HSM15, Section 9], to compute a relaxed alias table that allows two aliases for each bucket on the top level. This redistribution algorithm is due to Peter Sanders and therefore not contained in this dissertation.

First, we compute the total weight W with an all-reduction. This allows each PE to determine whether its meta-item is light or heavy. PEs with a light meta-item have a *deficit* of size $W/p - W_i$, whereas PEs with a heavy meta-item have a *surplus* of size W_i *and* a deficit of size W/p. We then use the redistribution algorithm of Ref. [HSM15, Section 9] to align deficits and surpluses, and transmit the required parts of items. This algorithm is based on aligning prefix sums of *deficits* and *surpluses* with an efficient parallel merging algorithm. The resulting sequence yields the set of receivers for each PE with a surplus.

The slightly counterintuitive handling of heavy items above, simultaneously giving them a surplus and a deficit, ensures that each meta-bucket has at most two aliases. Consider a light meta-item that received the final part of a heavy meta-item's surplus, but still has some amount of deficit remaining. Then, if the next meta-item that has just above-average weight, only a small piece of it would be redistributed, which would not necessarily suffice to fill the remaining deficit of a light meta-item. Considering the entirety of the heavy meta-items as surplus fixes this, as the next heavy meta-item always has size at least W/p, and is thus guaranteed to be large enough to fill the remaining deficit.

We now consider the running time of this approach. Computing W takes time $\mathcal{O}(\alpha \log p)$ and the local computations that follow require only constant time. Matching up communication partners for the redistribution takes time $\mathcal{O}(\alpha \log p)$ using Batcher's [Bat68] parallel merging algorithm, see Ref. [HSM15] for the details. Each PE with a deficit is involved in at most two redistributions as a receiver, while PEs with a surplus (ie, a heavy meta-item) are additionally involved in one other redistribution as a sender. Each redistribution consists of one sender transmitting its meta-item to a set of receivers. For the last receiver to know how large its piece of the meta-item is, it additionally has to consult the prefix-sum of the deficits and surpluses computed by the redistribution algorithm. By subtracting the preceding PEs' deficits from the prefix sum of the surpluses up to and including its heavy meta-item, it can recover the size of its share. All collective operations involved have latency logarithmic in the number of PEs involved, ie, at most $\log p$. Thus, we obtain the claimed construction time bound.

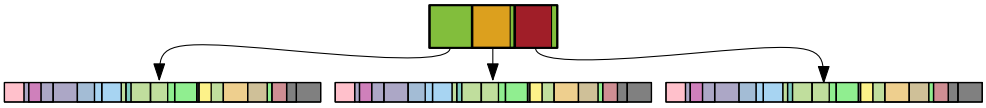


Figure 3.3: Illustration of a two-level alias table with 3 PEs.

Sampling from the resulting two-level alias table needs an additional indirection. First, a meta-bucket j is computed. Then a request is sent to PE j which identifies the subset E_i from which the item should be selected and delegates that task of sampling from E_i to PE i.[3] Computing the meta-bucket from the top-level 'two-alias table' is similar to sampling from a regular alias table, except that the buckets are now partitioned into three pieces for items i, $A_{i,1}$, and $A_{i,2}$, and we need to determine whether $x := W/p \cdot \text{rand}()$ is (a) less than or equal to W_i, in which case item i is returned, (b) at most $W_i + A_{i,1}.w$, in which the first alias $A_{i,1}$ is returned, or (c) between $W_i + A_{i,1}.w$ and W/p, in which case the second alias $A_{i,2}$ is returned. This requires only a constant amount of time. □

Redistributing Items. As discussed so far, *constructing* distributed-memory 2-level alias tables is communication-efficient. However, when large items are predominantly allocated on few PEs, *sampling* many items can lead to an overload on PEs with large W_i. We can remedy this problem by moving large items to different PEs or even by splitting them between multiple PEs. This redistribution can be done in the same way we construct alias tables. This implies a trade-off between redistribution cost (part of preprocessing) and load balance during sampling.

We now look at the case where an adversary can choose an arbitrarily skewed distribution of item sizes but where the items are randomly allocated to PEs (or that we actively randomise the allocation implying $\Theta(n/p)$ additional communication volume).

Theorem 3.4 (Item Redistribution for Distributed 2-level Alias Tables)
If items are randomly distributed over the PEs initially, it suffices to redistribute $\mathcal{O}(\log p)$ items from each PE such that afterwards, each PE has total weight $\mathcal{O}(W/p)$ in expectation and $\mathcal{O}(n/p + \log p)$ (pieces of) items. This redistribution takes expected time $\mathcal{O}(\alpha \log p)$.

Proof. Let us distinguish between *super-heavy* items whose weight exceeds $cW/(p \log p)$ for an appropriate constant c and the remaining *ordinary* items. The expected maximum weight allocated to a PE based on ordinary items is $\mathcal{O}(W/p)$ [San96]. Thus, we need only redistribute super-heavy items. Yet, by definition of super-heavy items, at most $p \log(p)/c$ of them can exist in the entire input. By standard balls-into-bins arguments [RS98, Theorem 1, Case 2], only $\mathcal{O}(\log p)$ super-heavy items are initially allocated to any PE with high probability. We adapt the redistribution used in Theorem 3.3 (ii) to cope with a logarithmic number of items to redistribute. The sending PEs simply send all of their super-heavy items to all of their associated receivers, and the receivers unpack the bulk to extract the parts they have to represent. The asymptotic complexity does not change since even messages of size $\mathcal{O}(\log p)$ can be broadcast in time $\mathcal{O}(\alpha \log p)$, eg, using pipelining [San+19, Chapter 13.1.2]. □

[3] If we ensure that meta-items have similar size (see Section 3.3.3), then we can arrange the meta-items in such a way that $i = j$ most of the time.

3.4 Output-Sensitive Sampling With Replacement (Problem WRS–R)

The algorithm of Section 3.3.2 easily generalises to sampling k items with replacement by simply executing k queries. Since the precomputed data structures are static, these queries can be run in parallel. We obtain optimal span $\mathcal{O}(1)$ and work $\mathcal{O}(k)$.

Corollary 3.5 (Weighted Sampling with Replacement using Alias Tables)
After a suitable alias table data structure has been computed, we can sample k items with replacement with work $\mathcal{O}(k)$ and span $\mathcal{O}(1)$.

Yet if the weights are skewed this may not be optimal since large items will be sampled multiple times. Here, we describe an *output-sensitive* algorithm that outputs only different items in the sample together with how often they were sampled, ie, a set S of pairs (i, k_i) indicating that item i was sampled k_i times. The work will be proportional to the output size s up to a small additive term. For highly skewed distributions, even $k \gg n$ may make sense. Note that outputting multiplicities may be important for appropriately processing the samples. For example, let X denote a random variable where item i is sampled with probability w_i/W and suppose we want a truthful estimator for the expectation of $f(X)$ for some function f that is assumed to correlate with the weights. Then the Hansen-Hurwitz estimator, $\hat{f}(X) = W/k \cdot \sum_{(i,k_i)\in S} k_i f(i)/w_i$, provides the desired result [HH43].

We will combine and adapt three previously used techniques for related problems: the bucket tables from Section 3.1.2, the divide-and-conquer technique from Section 3.1.4 [San+18], and the Poisson sampling algorithm of Bringmann and Panagiotou [BP17].

We approximately sort the items into $u = \lceil \log U \rceil$ groups of items whose weights differ by at most a factor of two using integer sorting. To do this, weight w_i is mapped to group $\lfloor \log(w_i/w_{\min}) \rfloor$. In contrast to subset sampling, this has to extend to items with even the smallest weights.

To help determine the number of samples to be drawn from each group, we build a complete binary tree with one *nonempty* group at each leaf. Interior nodes store the total weight of items in groups assigned to their subtrees. This *divide-and-conquer tree (DC-tree)* allows us to generalise the divide-and-conquer technique described in Section 3.1.4 to weighted items. Suppose we want to sample k elements from a subtree rooted at an interior node whose left subtree has total weight L and whose right subtree has total weight R. Then the number of items k' to be sampled from the left has a binomial distribution (k trials with success probability $L/(L+R)$). We can generate k' accordingly and then recursively sample k' items from the left subtree and $k - k'$ items from the right subtree. A recursive algorithm can thus split the number of items to be sampled at the root into numbers of items sampled at each group. When a subtree receives no samples, the recursion can be stopped. Since the distribution of weights to groups can be highly skewed, this stopping rule will be important in the analysis.

For each group G, we integrate bucket tables and DC-trees as follows. Let n_G be the number of items in the group, and let $[a, 2a)$ be the range of weights mapped to this group. For the bucket table we can use a very simple variant that stores these n_G items with weights from the interval $[a, 2a)$ in n_G buckets of capacity $2a$. Sampling one element then uses a rejection

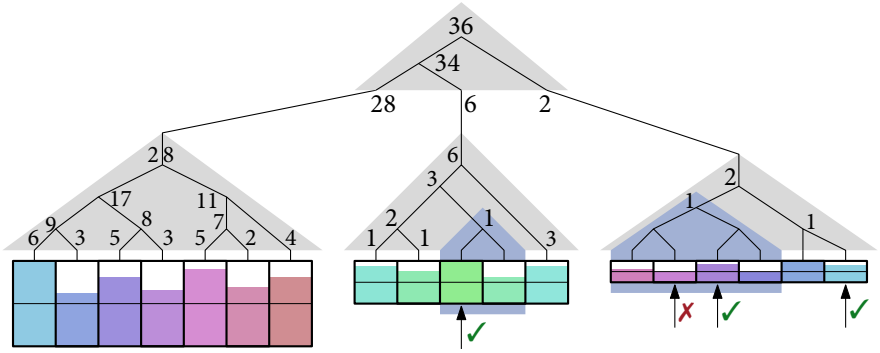


Figure 3.4: Output-sensitive sampling: assignment of multiplicities with $k = 36$.

method that repeats the sampling attempt when the random variate leads to an empty part of a bucket.[4]

We also build a DC-tree for each group. A simple linear mapping of items into the bucket table allows us to associate a range of relevant buckets b_T with each subtree T of the DC-tree. For sampling m items from a group G, we use the DC-tree to decide how many samples each subtree has to contribute. When this number decreases to 1 for a subtree T, we sample this element directly and in constant expected time from the buckets in the range b_T.[5]

Figure 3.4 gives an example of the query process. We obtain the following complexities:

Theorem 3.6 (Output-Sensitive Weighted Sampling with Replacement)

Preprocessing for problem WRS–R can be done in the time and span needed for integer sorting n elements with $u = \lceil \log U \rceil = \lceil \log(w_{\max}/w_{\min}) \rceil$ different keys[6] plus linear work and logarithmic span (on a CREW PRAM) for building the actual data structure. Using this data structure, sampling a multiset S with k items and s different items can be done with span $\mathcal{O}(\log n)$ and expected work $\mathcal{O}(s + \log n)$ on a CREW PRAM.

Proof. Besides sorting the items into groups, we have to build binary trees of total size $\mathcal{O}(n)$. This can be done with logarithmic span and linear work using a simple bottom-up reduction. The bucket-tables which have total size n can be constructed as in Section 3.3.2.

The span of a query is essentially the depth of the trees, $\log u + \log n \le 2 \log n$.

Bounding the work for a query is more complicated since, in the worst case, the algorithm can traverse paths of logarithmic length in the DC-trees for just a constant number of samples taken at its leaves. However, this is unlikely to happen, and we show that in expectation

[4] If desired, we can also avoid rejection sampling by mapping the items into up to $2n_G$ buckets of size a without gaps. This way there are at most two items in each bucket. Note that this is still different from alias tables because we need to map ranges of consecutive items to ranges of buckets. This is not possible for alias tables.

[5] For practical implementations, it may make sense to stop descending the tree when m drops below some constant. We can then join duplicates using a small hash table that fits in cache. This optimisation is described in more detail on page 89.

[6] Section 3.1.1 discusses the cost of this operation on different models of computation.

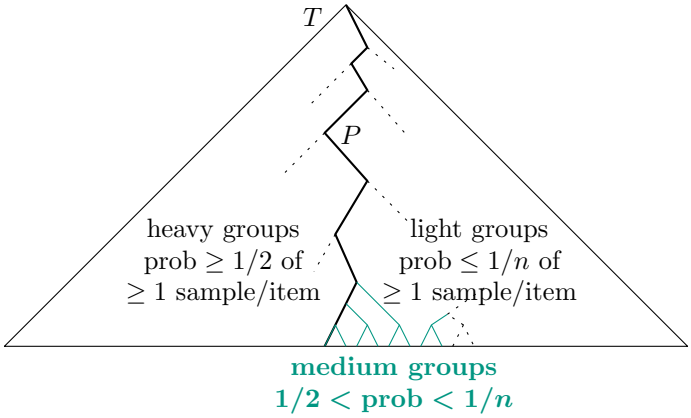


Figure 3.5: Illustration of the groups in Theorem 3.6.

the overhead is a constant factor plus an additive logarithmic term. We are thus allowed to charge a constant amount of work to each different item in the output and can afford a leftover logarithmic term.

We first consider the top-most DC-tree T that divides samples between groups. Tree T is naturally split into a *heavy* range of groups that contain some items which are sampled with probability at least $1/2$, a range of *medium* groups whose items are sampled with probability less than $1/2$ but above $1/n$, and a remaining range of *light* groups in which all items are sampled with probability at most $1/n$. Refer to Figure 3.5 for an illustration. Assuming the heavy groups are to the left, consider the path P in T leading to the first light group. Subtrees branching from P to the left are complete subtrees that lead to heavy groups only. Since all leaves represent non-empty groups, we can charge the cost for traversing the left trees to the elements in the groups at the leaves: in expectation, at least half of these groups contain elements that yield a sample.

Consider next the medium groups, ie, those whose items yield at least one sample with probability at least $1/n$. There can be at most $\log n$ such 'middle' groups because groups' weight bounds fall exponentially. Because these groups are consecutive, they fit into subtrees of T of logarithmic total size. Therefore, traversing the paths to them causes $\mathcal{O}(\log n)$ work in total.

This leaves the light groups whose items individually yield at least one sample with probability at most $1/n$. There are at most $u \leq n$ such groups. Fix an arbitrary one of them, G, and let n_G be the number of items in this group. Then the expected cost of traversing the path to G is the path's length ($\leq \log u \leq \log n$) times the likelihood that the path is taken ($\leq n_G/n$). Summing this over all light groups yields total expected work $\log n \sum_G n_G/n \leq \log n$.

Finally, Lemma 3.7 shows that the work for traversing DC-trees within a group is linear in the output size from each group. Summing this over all groups yields the desired bound. □

Lemma 3.7 (Work Bound for Sampling from a DC-tree [HS19d])
Consider a DC-tree plus bucket array for sampling with replacement of k out of n items where weights are in the range $[a, 2a)$. Then the expected work for sampling is $\mathcal{O}(s)$, where s is the number of different sampled items.

Proof (due to Peter Sanders and included here for self-containedness). If $k \geq n$, $\Omega(n)$ items are sampled in expectation at a total cost of $\mathcal{O}(n)$. So assume $k < n$ from now on. The first $\log k + \mathcal{O}(1)$ levels of T may be traversed completely, contributing a total cost of $\mathcal{O}(k)$.

For the lower levels, we count the number Y of visited nodes from which at least 2 items are sampled. This is proportional to the total number of visited nodes since nodes from which only one item is sampled contribute only constant expected cost (for directly sampling from the array) and since there are at most $2Y$ such nodes.

Let X denote the number of items sampled at a node at level ℓ of tree T. An interior node at level ℓ represents $2^{L-\ell}$ leaves with total weight $W_\ell \leq 2a2^{L-\ell}$ where $L = \lceil \log n \rceil$. X has a binomial distribution with k trials and success probability

$$\rho = \frac{W_\ell}{W} \leq \frac{2a2^{L-\ell}}{a2^{L-1}} = 4 \cdot 2^{-\ell} \quad .$$

Hence, the probability of sampling at least two items at a node at level ℓ can be estimated as

$$\mathbf{P}[X \geq 2] = 1 - \mathbf{P}[X = 0] - \mathbf{P}[X = 1] = 1 - (1-\rho)^k - k\rho(1-\rho)^{k-1} \approx (k\rho)^2/2$$

where the '$\approx$' holds for $k\rho \ll 1$ and was obtained by series development in the variable $k\rho$. The expected cost at level $\ell > \log k + \mathcal{O}(1)$ is thus estimated as the product of the above probability and the number of nodes at this level, ie,

$$2^\ell \mathbf{P}[X \geq 2] \approx 2^\ell (k\rho)^2/2 \leq 2^\ell (k \cdot 4 \cdot 2^{-\ell})^2/2 = 8k^2 2^{-\ell} \quad .$$

At level $\ell = \lceil \log k \rceil + 3 + i$ we thus get expected cost $\leq 8k^2 2^{-\lceil \log k \rceil} \cdot 2^{-3} \cdot 2^{-i} \leq k \cdot 2^{-i}$. Summing this over i to cover all lower levels of the tree yields expected total cost $Y \in \mathcal{O}(k)$. □

3.4.1 Distributed Output-Sensitive Sampling with Replacement

The batched character of sampling with replacement makes this setting attractive for a distributed implementation using the owner-computes approach. Each PE builds the data structure described above for its local items. Furthermore, we build a top-level DC-tree that distributes the samples between the PEs, ie, with one leaf for each PE. We will see below that this can be done using a bottom-up reduction over the total item weights on each PE, ie, no PRAM emulation or replication is needed. Each PE only needs to store the partial sums appearing on the path in the reduction tree leading to its leaf. Sampling itself can then proceed without communication. Each PE simply descends its path in the top-level DC-tree analogous to the uniform case [San+18, Section 3.2]. Afterwards, each PE knows how many samples to draw from its local items. Note that we assume k to be known on all PEs and that communication for computing results from the sample is not considered here.

Theorem 3.8 (Distributed Output-Sensitive Weighted Sampling with Replacement)
Sampling k out of n items with replacement (problem WRS–R) can be done in a communication-free way with processing overhead $\mathcal{O}(\log p)$ in addition to the time needed for taking the local sample. Building and distributing the DC-tree for distributing the samples is possible in time $\mathcal{O}(\alpha \log p)$.

Proof. It remains to explain how the reduction can be done in such a way that it can be used as a DC-tree during a query and such that each PE knows the path in the reduction tree leading to its leaf. First, assume that $p = 2^d$ is a power of two. Then we can use the well-known hypercube algorithm for all-reduce (eg, Ref. [Kum+94]). In iteration $i \in [1\,..\,d]$ of this algorithm, a PE knows the sum for its local $i-1$-dimensional subcube and receives the sum for the neighbouring subcube along dimension i to compute the sum for its local i dimensional subcube. For building the DC-tree, each PE simply records all these values.

For general (ie, non-power-of-two) values of p, we first build the DC tree for $d = \lfloor \log p \rfloor$. Then, each PE i with $i < 2^d$ and $j := i + 2^d < p$ receives the aggregate local item weight from PE j and then sends its path to PE j. □

Observe that the way Theorem 3.8 is stated, this sample distribution algorithm can also be applied to the naïve local sampling method of repeatedly sampling one item from an alias table.

Similar to Section 3.3.3, it depends on the assignment of the items to the PEs whether this approach is load-balanced for the local computations. Before, we needed a balanced distribution of both the number of items and the items' weights. Now the situation is better because items may be sampled multiple times but require work only once. On the other hand, we do not want to split heavy items between multiple PEs since this would increase the amount of work needed to process the sample. It would also undermine the idea of communication-free sampling if we had to collect samples of the same item assigned to different PEs. Below, we once more analyse the situation for items with arbitrary weights that are allocated to the PEs randomly.

Theorem 3.9 (Weighted Sampling with Replacement with Random Allocation)
Consider an arbitrary set of item sizes and let $u = \log(\max_i w_i / \min_i w_i)$. If items are randomly assigned to the PEs initially, then preprocessing takes expected time $\mathcal{O}\big(\text{isort}_u^1(n/p) + \alpha \log p\big)$ where $\text{isort}_u^1(x)$ denotes the time for sequential integer sorting of x elements using keys from the range $[0\,..\,u]$.[7] Using this data structure, sampling a multiset S with k items and s different items can be done in expected time $\mathcal{O}(s/p + \log p)$.

Proof. For preprocessing, standard balls-into-bins arguments tell us that a random assignment of $n \geq p \log p$ items to p PEs results in no more than $n/p + \Theta(\sqrt{n/p \cdot \log p}) = \Theta(n/p)$ items at any PE with high probability (see, eg, [RS98]), and $\mathcal{O}(\log p)$ items if $n < p \log p$. Thus, we have at most $\mathcal{O}(n/p + \log p)$ items at any PE with high probability.

Since sorting is now a local operation, we only need an efficient sequential algorithm for approximately sorting integers. The term $\alpha \log p$ is for the global DC-tree as in Theorem 3.8.

A sampling operation will sample s items. Since their allocation is independent of the choice of the sampled items, we can apply the same balls-into-bins bounds as above to conclude that no more than $\mathcal{O}(s/p + \log p)$ sample items are allocated to any PE with high probability. □

[7] Note that this will be linear in all practically relevant situations.

3.5 Sampling Without Replacement (Problem WRS–N)

A common approach to sampling without replacement is to sample *with* replacement and reject items that were already sampled. Using a hash table to quickly identify duplicate samples works well for small sample sizes in the uniform (ie, unweighted) sampling setting [San+18]. It would, however, not work well for weighted inputs with skewed distribution, where few items would dominate the sample with replacement, leading to an extremely high number of rejected items. But presume that we knew an $\ell > k$ so that a sample of size ℓ *with replacement* contains at least k and no more than $\mathcal{O}(k)$ unique items with sufficiently high probability. Then we could use the output-sensitive algorithm for WRS–R of Section 3.4 and discard the samples' multiplicities to obtain a sample without replacement of size $s \geq k$ in time $\mathcal{O}(s)$. This sample can then be downsampled to size k with linear work using the exponential clocks method (Section 3.1.3).

The crucial step, of course, is to find ℓ. This depends heavily on the distribution of the input: if all items have similar weight, ℓ will be little larger than k, but if the weights are heavily skewed, it could have to be exponentially large. Our main observation is that by having distributed the items into groups of similar weight (see Section 3.4), we already have enough information to compute a good value for ℓ in logarithmic time. The basis of this estimation is to assume that sufficiently heavy items are sampled once and lighter items are sampled with probability proportional to their weight. We precompute the data needed for the estimation for each group and then, at query time, perform a binary search over the groups. Suppose the currently considered group stores elements with weights in the range $[a, 2a)$. Then we try the value $\ell = \lceil W/a \rceil$. We (over-)estimate the resulting global number of unique samples as

$$\left|\{i \mid w_i \geq a\}\right| + \ell \cdot \sum_{i:w_i<a} \frac{w_i}{W} \ .$$

This is the precomputed *number* of items with weight $\geq a$ (those in heavier groups, including the current group) plus ℓ times the relative *weight* of the items with weight $< a$. Below we show that aiming for an overestimate of $2k$ different samples will yield at least k with constant probability. First, we state the above estimation as a lemma—which we shall prove later—providing both an over- and an underestimate.

Lemma 3.10 (Expected Number of Unique Items in a Sample with Replacement)
Let X be the random variable describing the number of unique items in a weighted sample with replacement of size ℓ. Then,

$$\left(1 - \frac{1}{e}\right) t_\ell \leq \mathbf{E}[X] \leq t_\ell \quad \text{where} \quad t_\ell := \ell \cdot \sum_{i:\ \frac{w_i}{W} < \frac{1}{\ell}} \frac{w_i}{W} + \left|\left\{i \;\middle|\; \frac{w_i}{W} \geq \frac{1}{\ell}\right\}\right| \ .$$

Before we dive into the technical parts, we now describe the algorithm in more detail.

Construction. To support fast queries, we need to do some additional precomputation during construction. First, compute W and apply the preprocessing of Section 3.4. Then, compute each group's relative total weight, $g_i := \sum_{j \in G_i} w_j/W$, where G_i denotes the group with index i, and their exclusive prefix sum, $h_i := \sum_{j<i} g_i$. Further, let $c_i := \sum_{j<i} |G_j|$ be the number of items

in all groups before group i. These prefix sums are used at query time to quickly estimate the number of unique samples for a given sample size with replacement.

Queries. At query time, we can find a suitable value of ℓ using binary search over the non-empty groups. These are exactly the leaves of the top-level DC-tree constructed by the preprocessing for WRS–R. Let i be the index of the group currently under consideration and $[a_i, 2a_i)$ be the interval of probabilities assigned to the group. Lemma 3.10 gives upper and lower bounds for the expected number of unique items in a sample of size ℓ, which are only a constant factor apart. With the data structures computed during construction, we can evaluate t_ℓ for $\ell := \lceil W/a_i \rceil$ as $t_\ell := \ell \cdot h_i + (n - c_i)$. Lemma 3.12 below states that we can find a group i whose minimum weight a_i gives us a value of ℓ such that sampling ℓ elements with replacement yields $2k$ unique elements in expectation, but at the same time not too many more (ie, $\mathcal{O}(k)$). This ensures that the output contains at least k unique elements with sufficient probability. However, this group may be empty and thus not considered in the binary search. Let G_i and G_j be the non-empty groups with items of next-smaller and next-larger weights, respectively. Then we are free to choose ℓ in the range $\left[2a_i \,..\, a_j\right]$. We do this by solving the inequality $\mathbf{E}[X] \geq (1 - 1/e)t_\ell \stackrel{!}{=} 2k$ of Lemma 3.10 for ℓ. As there are no items with weights in the range considered, the sum of weights and the set cardinality in the equation of the lemma remain constant, and we can solve for ℓ in constant time to find the most suitable value of t_ℓ, obtaining $\ell = h_i^{-1}\left(2k\frac{e}{e-1} - (n - c_i)\right)$. We then draw a sample *with* replacement of size ℓ as in the algorithm of Theorem 3.6, and discard the sampled items' multiplicities. If the resulting sample has fewer than k unique elements, it is rejected and the sampling is repeated. In a postprocessing step, we downsample the resulting sample with replacement to the required size k using the exponential clocks technique of Section 3.1.3.

Overall, we get:

Theorem 3.11 (Weighted Sampling without Replacement)

Preprocessing for problem WRS–N is possible with the same work and time bounds as for problem WRS–R (Theorem 3.6). Sampling k items without replacement is then possible with span $\mathcal{O}(\log n)$ and expected work $\mathcal{O}(k + \log n)$ on a CREW PRAM.

Proof. The computation of W as well as the g_i, h_i, and c_i is possible with linear work and logarithmic span. Thus, the running time of the construction phase is dominated by the preprocessing for WRS–R of Theorem 3.6.

The binary search for a query takes time $\mathcal{O}(\log n)$, as each step of the search requires constant time using the values computed during the construction phase. Because ℓ was chosen to yield at least $2k$ samples in expectation, the probability of the query resulting in fewer than k unique items is at most $1/2$ by Markov's inequality. Downsampling the resulting sample needs $\mathcal{O}(k)$ work and $\mathcal{O}(\log n)$ span for a selection of k out of $\Theta(k)$ items using the algorithm of Section 2.3.1. The $\log n$ term in the work bound comes from sampling with replacement (Theorem 3.6). In summary, we obtain the claimed bound. □

Proof (Lemma 3.10). We can bound the number of unique items in a sample with replacement of size ℓ by considering the probability of any item to *not* be sampled. Item i is *not* part of the

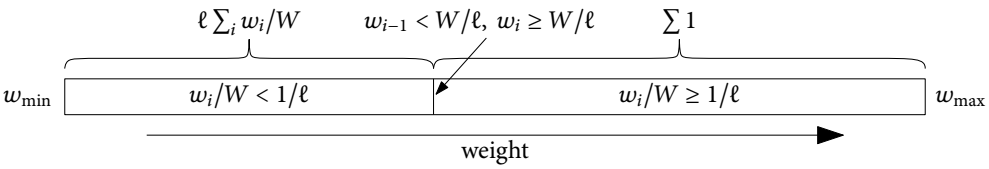


Figure 3.6: How the expected number of samples $\mathbf{E}[X]$ is influenced by the choice of ℓ depends on the distribution of the inputs (Lemma 3.10). Items with small probability of being sampled contribute linearly in their weight, whereas large items are expected to be sampled.

sample with probability $(1 - w_i/W)^\ell$, and we obtain

$$\mathbf{E}[X] = \sum_{i=1}^{n} 1 - (1 - w_i/W)^\ell$$

where X is the random variable describing the number of unique items in the sample.

We split the formula for the expectation of X into two parts: the items with relative weight w_i/W below $1/\ell$, ie, those which yield less than a single sample in expectation, and the remaining ones with $w_i/W \geq 1/\ell$ which are expected to occur in the sample.

Let us briefly consider the case of $\ell = 1$ so that we can safely assume $\ell > 1$ in the remainder of the proof. In the trivial case where the input consists of a single item and $w_1 = W$, we obtain $t_\ell = 1 \cdot 0 + 1 = 1$. Otherwise, all items are in the first group, and we have $t_\ell = 1 \cdot \sum_i w_i/W + 0 = 1$. In either case, we obtain $t_\ell = 1$ as expected and are done.

Bernoulli's inequality states $(1 + x)^r \geq 1 + rx$ for $x \in \mathbb{R}, x > -1$ and $r \in \mathbb{N}$. Because we can now assume $\ell > 1$, we have $w_i/W < 1/\ell < 1$ for the items of the first group and thus obtain the upper bound

$$1 - (1 - w_i/W)^\ell \leq \ell w_i/W \ .$$

We can also find a lower bound for the inclusion probability of items in the first group:

$$\begin{aligned} 1 - (1 - w_i/W)^\ell &= 1 - (1 - b_i/n)^{c \cdot n} && \text{where } b_i := n w_i/W \text{ and } c := \ell/n \\ &\geq 1 - e^{-cb_i} = 1 - e^{-\ell w_i/W} && \text{by } e^x \geq (1 + x/n)^n \\ &\geq (1 - 1/e)\, \ell w_i/W && \text{as } e^{-x} \leq 1 - (1 - 1/e)x \text{ for } x \in (0, 1). \end{aligned}$$

Conveniently, this lower bound is $(1 - 1/e)$ times our upper bound, and thus the first term of t is explained.

Consider now the items with $w_i/W \geq 1/\ell$, ie, the items which yield at least one sample in expectation. Because it is a probability, $1 - (1 - w_i/W)^\ell$ is clearly bounded above by 1. In the other direction, because $w_i/W \geq 1/\ell$, we obtain

$$1 - (1 - w_i/W)^\ell \geq 1 - (1 - 1/\ell)^\ell \geq 1 - 1/e \qquad \text{by } (1 + x/n)^n \leq e^x \ .$$

This again assumes $\ell > 1$. Adding the results of both parts yields the claimed inequalities. □

An example of this is illustrated in Figure 3.6. By applying the above estimation to the groups used by the algorithm of Section 3.4, we can quickly obtain an estimate for the output size that is at most a factor of two worse.

Lemma 3.12 (Group-Based Estimation of Sample Cardinality)
Applying the estimation of Lemma 3.10 to groups of items of similar weight as in Section 3.4 loosens the bound on $\mathbf{E}[X]$ *by at most a factor of two.*

Proof. Item i is in group $\lfloor \log(w_i/w_{\min}) \rfloor$. Label the groups G_1 to G_u where $u := \lceil \log U \rceil = \lceil \log(w_{\max}/w_{\min}) \rceil$ and let the value range of group i be $[a_i, 2a_i)$. The difference to Lemma 3.10 now is that we can only choose ℓ as $\lceil W/a_i \rceil$ for some group i. This necessitates moving entire groups between the left and right parts of the estimation (with left and right as illustrated in Figure 3.6). Pick a group i, ie, $\ell := \lceil W/a_i \rceil$, and consider the effects of choosing group $i+1$ instead, ie, $\ell' := \lceil W/a_{i+1} \rceil = \lceil W/(2a_i) \rceil = \ell/2$. This results in moving group i from the right part of the estimation to the left part. The contribution of any group $j > i$ is the same for t_ℓ and $t_{\ell'}$ because their items are expected to be sampled in both instances. Furthermore, the contribution of groups 1 to $i-1$ is halved, as $\ell' = \ell/2$. It remains to bound the contribution of group i, which is moved from the right part of the estimation to the left part: it contributes $|G_i|$ to t_ℓ and $\xi := \ell' \sum_{j \in G_i} w_j/W$ to $t_{\ell'}$. However, as all items in G_i are in the weight range $[a_i, 2a_i)$ and $\ell' = \lceil W/a_{i+1} \rceil = \lceil W/(2a_i) \rceil$, we have $\xi \geq \ell' a_i |G_i| / W \geq a_i/a_{i+1} \cdot |G_i| = |G_i|/2$, and thus the expected number of samples contributed by group i at most halves, too. Thus, $t_\ell \leq 2t_{\ell'}$, and for any ℓ of Lemma 3.10, we can find an $\hat{\ell} \geq \ell$ that yields no fewer, but also at most twice as many unique items in expectation while only relying on information about the groups, not the w_i. □

3.5.1 Distributed Sampling without Replacement

We again use the owner-computes approach and adapt the distributed data structure for problem WRS–R from Section 3.4.1. However, to find the right estimate for the number ℓ of samples with replacement, we need to perform a global estimation taking all items into account. This can be achieved by finding the global sum of all the local prefix sums in each step of the binary search. Also, in each iteration, we need a nested binary search to find corresponding buckets in the local bucket arrays.

If the global number of group u is not too large, we can consider a different trade-off between preprocessing cost and query cost. We precompute replicated arrays of global group sizes and weights. This allows finding the right value of ℓ without communication at query-time. Local sorting is performed using bucket sort with u groups. The global group sizes can then be computed as a sum all-reduction on the local arrays of group sizes.

During a query, after sampling with replacement, we need a global parallel selection algorithm to reduce the sample size to k. This can be done using the unsorted selection algorithm of Section 2.3.1. We get the following result:

Theorem 3.13 (Weighted Sampling without Replacement with Random Allocation)
Consider an arbitrary set of item sizes and let $u = \log(\max_i w_i / \min_i w_i)$. *If items are randomly assigned to the PEs initially, then preprocessing takes expected time* $\mathcal{O}\left(\text{isort}^1_u(n/p) + \alpha \log p\right)$

where $\text{isort}_u^1(n/p)$ *denotes the time for sequential integer sorting of* x *elements using keys from the range* $[0\,..\,u]$. *Using this data structure, sampling* k *items without replacement can be done in expected time* $\mathcal{O}(k/p + \alpha \log(n) \log(p))$.

A variant that uses a replicated array of global group sizes works with $\mathcal{O}(n/p + \beta u + \alpha \log p)$ *preprocessing time and query time* $\mathcal{O}(k/p + \log u + \alpha \log kp)$.

Proof. Finding the right estimate for ℓ with binary search increases the latency of the algorithm to $\mathcal{O}(\alpha \log(p) \log(n))$.

With a replicated array of global group sizes, we incur additional preprocessing time $\mathcal{O}(n/p + u)$ for bucket sort and $\mathcal{O}(\beta u + \alpha \log p)$ for the sum all-reduction on the arrays of local group sizes. At query time, this allows finding the right value of ℓ with local work $\mathcal{O}(\log u) \leq \mathcal{O}(\log n)$.

Drawing the sample with replacement takes expected time $\mathcal{O}(k/p + \log p)$ by Theorems 3.8 and 3.9.

Using the unsorted selection algorithm of Section 2.3.1, the selection at the end of the query takes time $\mathcal{O}(k/p + \beta \min(k/p, \sqrt{p} \log_p k) + \alpha \log kp)$ w.h.p. for general inputs. However, for randomly distributed data, this becomes $\mathcal{O}(k/p + \alpha \log kp)$ by Corollary 2.2.

The claimed bounds then follow with Theorem 3.8. □

3.6 Poisson Sampling (Problem WRS–P)

Poisson sampling, also known as subset sampling, is a generalisation of Bernoulli sampling to the weighted case. The unweighted case can be solved in linear expected time with regard to the output size by computing the geometrically distributed distances between elements in the sample [AD85]. The naïve algorithm for the weighted problem, which consists of throwing a biased coin for each item, requires $\mathcal{O}(n)$ time. Bringmann and Panagiotou [BP17] show that this is optimal if only a single subset is desired, and present a sequential algorithm that is also optimal for multiple queries.

The difference between WRS–P on the one hand and WRS–1/WRS–R on the other hand is that we do not have a fixed sample size but rather treat the item weights as independent inclusion probabilities in the sample (this requires $w_i \leq 1$ for all i). Hence, different algorithms are required. Observe that the expected sample size is $W \leq n$. Then our goal is to devise a parallel preprocessing algorithm with work $\mathcal{O}(n)$ which subsequently permits sampling with work $\mathcal{O}(1 + W)$.

We now parallelise the approach of Bringmann and Panagiotou [BP17, Theorem 1.6]. Similar to our algorithm for sampling with replacement, this algorithm is based on grouping items into sets with similar weight. In each group, one can use ordinary Bernoulli sampling in connection with rejection sampling. Load balanced division between PEs can be done with a prefix sum calculation over the weights in each group.

Theorem 3.14 (Poisson Sampling)

Preprocessing for problem WRS–P can be done with work $\mathcal{O}(n)$ *and span* $\mathcal{O}(\log n)$. *A query can then be implemented with expected work* $\mathcal{O}(W + 1)$ *and span* $\mathcal{O}(\log n)$.

Proof. Approximately sort the items into $L + 1$ buckets with $L := \lceil \log n \rceil$, where bucket i is $B_i := \{ j \mid 2^{-i} \geq w_j \geq 2^{-(i+1)} \}$ for $i \in [0 \,..\, L-1]$ and $B_L := \{ j \mid 2^{-L} \geq w_j \}$ contains all sufficiently improbable elements. This can be done with linear work and logarithmic span on a CREW PRAM [RR89, Lemma 3.1]. Let σ denote the permutation of the items implied by this re-ordering.

Next, we precompute an assignment of consecutive ranges of permuted items to PEs. Observe that we need not care about bucket boundaries regarding the assignment; there are no dependencies between elements, and the result remains a valid Poisson sample. Therefore, we compute a prefix sum $\hat{w}_i := \sum_{j \leq i} w_{\sigma(j)}$ over the inclusion probabilities in their new order. PE i then handles the items whose $\hat{w}_j$ fall into the range $[(i-1)W/p, \, i \cdot W/p)$. These can be found with linear work and constant span by checking for every neighbouring pair of items whether a boundary falls between them, and if so, which PEs.

Sampling then proceeds on all buckets in parallel using the assignment calculated during preprocessing. In bucket i, all items have weight at most $\overline{w}_i := \max_{j \in B_i} w_j \leq 2^{-i}$, and, with the exception of bucket L, at least $\overline{w}_i/2$. PEs generate geometrically distributed skip values $v := \lfloor \ln(\text{rand}())/\ln(1-\overline{w}_i) \rfloor$ and then consider the $j := v/\overline{w}_i$-th item. The algorithm then uses rejection to output the item with probability $w_j/\overline{w}_i$. This process is repeated until a PE exceeds its allotted item range. In all buckets $i < L$, the acceptance probability is at least $1/2$ for every element, leading to an efficient algorithm in expectation. In bucket L, the smaller acceptance probability is not a problem, as with high probability only a constant number of items is ever considered to begin with. This is because the probability of being sampled is at most $1/n$ for items in bucket L. In total, this requires work $\mathcal{O}(1 + W)$ in expectation.

Although all PEs have to do about the same expected amount of work, there are some random fluctuations between the actual amount of work depending on the actual exponential variates that are computed. Using Chernoff bounds once more, these can be bounded to $\mathcal{O}(W/p + \log n)$ with high probability, leading to a (conservative) $\log n$ term for the span.

Each PE returns an array containing its part of the sample. An additional prefix sum computation can be used to rearrange the output into one contiguous array. This requires work $\mathcal{O}(n)$ and span $\mathcal{O}(\log n)$. □

3.6.1 Distributed Poisson Sampling

For the distributed setting, observe that problem WRS–P can be solved entirely independently over disjoint subsets of the input if we are not interested in load balancing. No communication is needed. We thus can directly adapt the result of Bringmann and Panagiotou [BP17]. Note that for a PE with n_i items, it suffices to sort them into $\log n_i$ categories now which is possible in linear time. A query on a PE with total weight W_i will take expected time $\mathcal{O}(1 + W_i)$. As expected parallel query time we get $\mathcal{O}(\max_i W_i + \log n)$ using an argument analogous to the proof of Theorem 3.14.

Once more, we analyse load balancing for the case that the items are distributed randomly.

Theorem 3.15 (Distributed Poisson Sampling for Random Allocation)
When items are distributed randomly, Poisson sampling (problem WRS–P) can be done in a communication-free way with expected preprocessing overhead $\mathcal{O}(n/p + \log p)$ and expected sampling time $\mathcal{O}(W/p + \log p)$.

Proof. The bound for the preprocessing follows by applying balls-into-bins bounds to the distribution of the number of items as in the proof of Theorem 3.9. The query bound follows similarly by exploiting that the expected maximum load is maximised when all the weight is concentrated in W items of weight 1 [San96]. □

3.7 Sampling with a Reservoir (Problem WRS–B)

We adapt the streaming algorithm of Efraimidis and Spirakis [ES06] (see also Section 3.1.3) to a distributed mini-batch streaming model as described in Section 1.2.4, where PEs process variable-size batches of items one at a time. After processing a batch, the task is to update the sample to be a uniform (or weighted) random sample without replacement of size $\min(k, n')$ of all n' items seen so far, up to and including the items of the current batch. Only the current mini-batch is available in memory, as the PEs' memory is too small to store previous batches.

3.7.1 Fully Distributed Weighted Reservoir Sampling

The basic idea of our algorithm is to combine the exponential clocks method, associating with each item an exponentially distributed key (*cf.* Section 3.1.3), with a communication-efficient bulk priority queue as described in Section 2.4. Using these two main ingredients, we maintain the set of the k items with the smallest keys, ie, the sample. Each PE is solely responsible for the items that were seen in its input, and no PE gets a special role (such as a coordinator node). During each batch, a PE inserts into its local reservoir all items whose key is smaller than the largest key of any item in the sample (the global *threshold*). When a batch finishes, the PEs perform a distributed selection for the k-th smallest key, which becomes the new global threshold, and discard all items with larger keys. The remaining elements form the new sample. During a mini-batch, the threshold remains unchanged.

Skip Value Computation. First, we show how to adapt the sampling method with skip values ('exponential jumps') of Efraimidis and Spirakis [ES06, Section 4] to the exponential variates described in Section 3.1.3. Recall that the difference between the variates associated with items in [ES06] and here is a simple $x \mapsto -\ln(x)$ mapping. Let v_i denote the key of item i, that is, the exponentially distributed variate associated with it, and define $T := \max_{i \in R} v_i$ as the threshold value, ie, the largest key of any item in the reservoir. Then the amount of weight to be skipped is exponentially distributed with rate T, which can be computed as $X := -\ln(\text{rand}())/T$, where rand() is a uniformly distributed random value from the interval $(0, 1]$. The key associated with the newly sampled item j is then $v_j := -\ln(\text{rand}(e^{-Tw_j}, 1))/w_j$, with $\text{rand}(a, b) := a + \text{rand}()(b - a)$. The range of this variate has been chosen so that v_j is less than T (as at this stage, it has already been determined that item j must be part of the reservoir, we need to compute a suitable variate from the distribution associated with its weight). In the

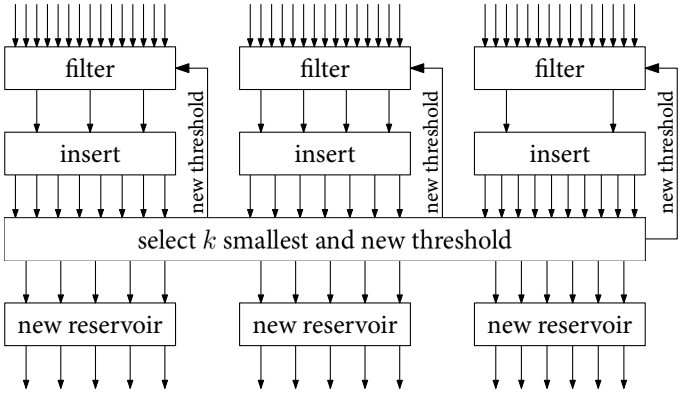


Figure 3.7: Schematic view of data flow in our reservoir sampling algorithm.

sequential setting, item j then replaces the item with the largest key in the reservoir, and the threshold T is updated to the now-largest key in the reservoir.

Algorithm Details. We now flesh out the remaining parts of our algorithm. Firstly, the reservoir is maintained in a distributed fashion using our bulk priority queue of Section 2.4. This requires each PE's *local reservoir* to be implemented as a B+ tree that is augmented to also support the *split*, *rank*, and *select* operations in logarithmic time (see Section 1.3.4). The *split* operation is used to quickly discard the items that are no longer part of the sample at the end of a batch, while *rank* and *select* are required for the threshold selection process.

Secondly, the algorithm *globally* maintains the threshold T, which is the same at all PEs and does not change during a mini-batch. The PEs process their items using the skip distance method described above, inserting the new candidate sample items into their *local* reservoirs.

Lastly, once all items of the mini-batch are processed, the PEs jointly select the globally k-th smallest key (see Section 3.1.5) in the union of all local reservoirs. This key becomes the insertion threshold for the next mini-batch. Each PE then discards all items with larger keys using a *split* operation on its local reservoir. The remaining k items in the union of all local reservoirs are a weighted sample without replacement of size k of all items seen so far. Algorithm 3.6 gives pseudocode and Figure 3.7 shows a schematic view of the algorithm.

Theorem 3.16 (Weighted Reservoir Sampling)

For weighted reservoir sampling with sample size k, processing a mini-batch of up to b items per PE is possible in time $\mathcal{O}(b + (b^* + 1)\log(b^* + k) + T_{sel})$*, where* $b^* \leq b$ *is the maximum number of items from the mini-batch below the insertion threshold on any PE, and* T_{sel} *is the time for selection from sorted sequences of size at most* $b^* + k$ *per PE (see Section 3.1.5).*

Proof. By definition of b^*, the local insertions require time $\mathcal{O}(b^* \log(b^* + k))$ in total because each local reservoir has size at most k at the start of the batch. Since we have to process each item's weight even when using the above skip value technique, $\mathcal{O}(b)$ time is required to identify the items to be inserted into the reservoir. The selection operation takes time T_{sel} which varies

Algorithm 3.6 : Weighted reservoir sampling in a Single-Program Multiple-Data style.

Input : A, the local portion of the mini-batch consisting of items with weight $w \in \mathbb{R}_+$ and index $i \in \mathbb{N}$; T, the previous mini-batch's threshold (initially $-\infty$); R, the local reservoir (initially empty), a B+ tree mapping keys from $\mathbb{R}$ to item IDs

Output : T', the updated threshold; R', the updated local reservoir

```
1  Function processBatch(A, T, R)
2      if T < 0 then                                  — fewer than k items seen globally before this batch
3          foreach (w, i) ∈ A do
4              R.insert(−ln(rand())/w, i)             — exponentially distributed keys
5      else
6          j := 0                                     — 1-based index of next item, initially invalid
7          while j ≤ |A| do
8              X := −ln(rand())/T                     — weight to be skipped
9              while X > 0 do                         — skip X amount of weight in total
10                 j := j + 1
11                 if j > |A| then break from both loops
12                 X := X − A[j].w
13             x := exp(−T · A[j].w)
14             v := −ln(rand(x, 1))/A[j].w            — new key
15             R.insert(v, A[j].i)
16     (T', i) := select(R, k) ∈ ℝ × ℕ                — select k globally smallest and new threshold
17     (R', _) := R.splitAt(i)                        — discard local items with larger keys
18     return (T', R')                                — return new threshold and reservoir
```

depending on the specifics of the input. The number of candidate items per PE for the selection is clearly bounded by the local reservoir size of at most $k + b^*$. Discarding the items with keys exceeding the new threshold using a *split* operation takes time logarithmic in the local reservoir size, ie, $\mathcal{O}(\log(k + b^*))$ time.

Now consider the implications of splitting the single stream of the sequential case into multiple independently-handled streams without carrying over any remaining skip weight X at the end of the stream (line 11 of Algorithm 3.6). This maintains correctness because the process is designed so that each unit of weight has the same probability of spawning a sample, regardless of when the procedure was started. Thus, partitioning the stream does not affect correctness.

Keeping the threshold fixed for the duration of each mini-batch also maintains correctness because the set of all items above *any* threshold always forms a weighted random sample (whose size is not known a priori). Here, the threshold is chosen so that the sample size is guaranteed to be at least k, and the selection process determines a new threshold to restore a fixed sample size of k items. □

The question now is how many items we (unnecessarily) insert into the local reservoirs by leaving the threshold unchanged during a mini-batch. We first analyse the number of insertions into *each* PE's local reservoir in Lemma 3.17, before considering the expected maximum number of insertions into *any* PE's local reservoir in Theorem 3.18.

Lemma 3.17 (Weighted Reservoir Sampling, Average Case)
If the item weights are independently drawn from a common continuous distribution and all batches have the same number of items on every PE, then our algorithm inserts $\mathcal{O}\left(\frac{k}{p}\left(1+\log\frac{n}{k}\right)\right)$ *items into each local reservoir in expectation.*

Proof. Efraimidis and Spirakis [ES06, Proposition 7] show that if the w_i are independent random variates from a common continuous distribution, their sequential reservoir sampling algorithm inserts $\mathcal{O}(k\log(n/k))$ items into the reservoir in expectation. We adapt this to mini-batches of b items per PE. Let X_i denote the number of insertions on a PE for batch i. We obtain a binomially distributed random variable with expectation

$$\mathbf{E}[X_i] = \sum_{j=1}^{b} \mathbf{P}[\text{item } j \text{ is inserted}] = b \cdot \frac{k}{n_{\text{pre}}} \leq b \cdot \frac{k}{ipb} = \frac{k}{ip},$$

where n_{pre} is the number of items seen globally before the batch began. For the initial $i_0 = \frac{k}{bp}$ iterations, this probability exceeds one, and we account for this with b insertions per PE, ie, $b \cdot \frac{k}{bp} = k/p$ overall. For mini-batches $i_0 \leq i < \frac{n}{pb}$ we obtain

$$\begin{aligned}\mathbf{E}[\textstyle\sum X_i] &\leq \sum_{\frac{k}{bp} \leq i \leq \frac{n}{bp}} \frac{k}{ip} = \frac{k}{p} \sum_{\frac{k}{bp} \leq i \leq \frac{n}{bp}} \frac{1}{i} = \frac{k}{p}\left(H_{\frac{n}{bp}} - H_{\frac{k}{bp}}\right) \\ &\leq \frac{k}{p}\left(1 + \ln\frac{n}{bp} - \ln\frac{k}{bp}\right) = \frac{k}{p}\left(1+\ln\frac{n}{k}\right),\end{aligned}$$

where H_n is the n-th harmonic number. □

Theorem 3.18 (Weighted Reservoir Sampling, Number of Insertions)
If the item weights are independently drawn from a common continuous distribution and all batches have the same number of items on every PE, then our algorithm inserts no more than $\mathcal{O}\left(\frac{k}{p}\log\frac{n}{k} + \log p\right)$ *items into* any *local reservoir in expectation.*

Proof. To obtain the maximum load over all PEs, we apply a Normal approximation to the bound on the X_i from the proof of Lemma 3.17. This yields the following approximation Y_i:

$$Y_i \sim \mathcal{N}\left(\frac{k}{ip}, \frac{k}{ip}\left(1 - \frac{k}{ipb}\right)\right).$$

Summing these over the mini-batches as above, we again obtain a normal distribution whose mean and variance are the sum of its summands' means and variances. We then apply a bound

on the maximum of independent and identically distributed (i.i.d.) Normal random variables obtained using the Cramér-Chernoff method [BLM13, Chapter 2.5],

$$\mathbf{E}\Big[\max_{j\in[1\,..\,p]} Z_j\Big] \le \mu + \sigma\sqrt{2\ln p} \quad \text{for} \quad Z_j \sim \mathcal{N}(\mu,\sigma^2) \ .$$

Using the mean of the Y_i as an upper bound to their variance, we obtain $\mu + \sqrt{2\mu \ln p}$ as an upper bound to the maximum per-PE load for $\mu = \frac{k}{p}(1 + \ln \frac{n}{k})$. Thus, the expected *bottleneck* number of insertions into any local reservoir is $\mathcal{O}(\frac{k}{p} \log \frac{n}{k} + \log p)$. □

Reservoir Sampling with Variable Reservoir Size. For any given threshold T, the items with keys less than or equal to T form a sample without replacement of all items seen so far. The size of this sample—call it s—is not known a priori, and in the previous sections, we used selection from the local reservoirs to determine the threshold so that $s = k$. If, however, the precise sample size is not important to the application, and s is allowed to vary in some range $\big[\underline{k}\,..\,\overline{k}\big]$, we can do better than for fixed k. By using the *amsSelect* approximate selection algorithm of Section 2.3.2.b) (see also Section 3.1.5), selection converges much faster if $\overline{k} - \underline{k}$ is sufficiently large (eg, a constant factor apart).

Corollary 3.19 (Reservoir Sampling with Variable Reservoir Size)
If the sample size is allowed to vary between $\underline{k}$ and $\overline{k}$, and $\overline{k} - \underline{k} \in \Omega\big(\overline{k}\big)$, then $T_{sel} \in \mathcal{O}(\alpha \log p)$ in the running time of Theorem 3.16 and Corollary 3.20.

Observe that if the items come from a common distribution, once $n \gg k$ items have been processed, turnover in the sample is very low. Accordingly, only few items have keys below the threshold to enter the reservoir in each batch. As a result, we can forego selection and let the sample grow until $s > \overline{k}$, potentially for multiple mini-batches. Only then does the selection have to find a new threshold. Additionally, the selection is faster because it does not have to find the item with a particular precise rank, but only *some* item in a given range of ranks.

3.7.2 Uniform Reservoir Sampling

The above algorithm can be easily adapted to uniform items by using the well-known skip distances for uniform reservoir sampling [Dev86, p. 640, *'Reservoir sampling with geometric jumps'*]. Here, we adapt Devroye's algorithm to our notation and model. Initially, when no threshold is known, the keys of the items are simply uniform random variates between 0 and 1, generated by rand( ). The number of items to be skipped then follows a geometric distribution with success probability T and can be computed as $X := \lfloor \ln(\text{rand}())/\ln(1-T) \rfloor$ for a given threshold T. The key of the $X+1$-th item, which is inserted into the local reservoir, is then simply $v := \text{rand}() \cdot T$. The remainder of the algorithm works analogously to the weighted case. Note that skipping X items is a constant-time operation, whereas skipping X' amount of weight in the weighted case requires examining every item that is skipped. As a result, the asymptotic local processing time for a batch of uniform items is the number of items inserted into the reservoir times the time to insert them.

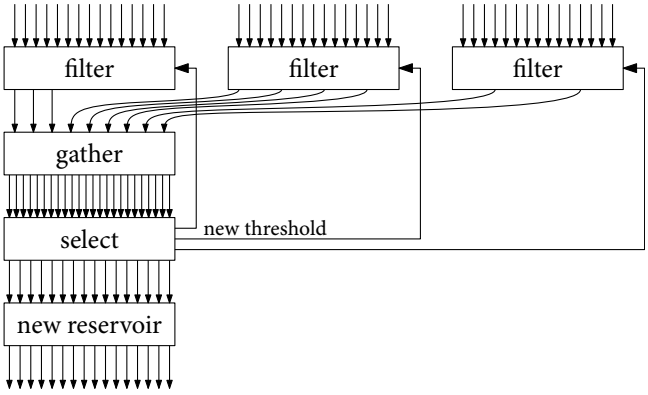


Figure 3.8: Schematic view of data flow in the centralised gathering algorithm.

Corollary 3.20 (Uniform Reservoir Sampling, Number of Insertions)
For unweighted (uniform) reservoir sampling with sample size k, processing a mini-batch is possible in $\mathcal{O}((b^* + 1)\log(b^* + k) + T_{sel})$ *time, where* $b^* \leq b$ *is the maximum number of items from the mini-batch on any PE that is below the insertion threshold, and* T_{sel} *is the time for selection from sorted sequences of size at most* $b^* + k$ *per PE (see Section 3.1.5).*

Observe that the criteria for random distribution of the input (see Section 3.1.5) are much easier to satisfy for uniform sampling. For example, a uniform arrival rate for all PEs suffices, as the keys associated with the items are uniformly random.

Refer to Lemma 3.17 and Theorem 3.18 above for an analysis of the number of items below the insertion threshold over all mini-batches.

3.7.3 A Centralised Approach

As a comparison point to our fully decentralised algorithms, we now describe an approach that designates a single PE as the coordinator, which maintains the sample. Again, the PEs use a threshold to immediately discard any items that cannot be part of the sample (in the first batch, if a PE receives more than k items, only the k items with the smallest keys need to be retained). The remaining candidates are gathered at the coordinator, which uses a standard sequential selection algorithm (eg, quickselect) to sequentially select the k smallest items, discards the rest, and broadcasts the new threshold. Figure 3.8 shows a schematic view of the algorithm.

Observe that the number of items gathered in later batches is small in expectation, as only few new items enter the reservoir if the items follow the same distribution in all mini-batches. This algorithm can be seen as an adaptation of Jayaram et al.'s [Jay+19] method to a mini-batch model, which renders the level set construction used therein unnecessary.

3.8 Experimental Evaluation

We now report experiments on alias tables (problem WRS–1, Section 3.3) and the closely related problem of sampling with replacement (problem WRS–R, Section 3.4) as well as reservoir sampling (problem WRS–B, Section 3.7). Our focus is on these problems because our algorithms for problems WRS–N and WRS–P are basically reductions to sorting and selection problems that have been studied elsewhere (WRS–N also needs WRS–R, which we do study here). We are also not aware of any competing parallel approaches that we could compare to.

Our experiments are split into two parts. First, we consider sampling with replacement (problems WRS–1 and WRS–R) in a shared-memory parallel setting. We limit ourselves to shared-memory experiments for these problems because we expect the shared-memory setting to be much more common in non-streaming workloads at present. Furthermore, there are simply no other published results on—or implementations of—parallel algorithms of any kind for these problems to the best of our knowledge. As a result, we judge the value of shared-memory implementations and evaluations to eclipse that of distributed-memory ones for these problems. Thereafter, we report distributed-memory experiments on reservoir sampling (WRS–B), which is a much more on-line problem where the need for distributed-memory codes and evaluations is much more obvious.

All of our implementations are publicly available under the GNU General Public Licence (version 3). The code and scripts for the first part of our experiments, problems WRS–1 and WRS–R, can be found at `https://github.com/lorenzhs/wrs`, while the artefacts for problem WRS–B are available at `https://github.com/lorenzhs/reservoir`.[8]

Experimental Platform. We use machines with Intel and AMD processors in our experiments. The Intel machine has four Xeon Gold 6138 CPUs (4×20 cores, 160 hyper-threads, of which we use up to 158 to minimise the influence of system tasks on our measurements) and 768 GiB of DDR4-2666 main memory. The AMD machine is a single-socket system with a 32-core AMD EPYC 7551P CPU (64 hyper-threads, of which we use up to 62) and 256 GiB of DDR4-2666 RAM. While single-socket, this machine also has non-uniform memory access (NUMA) characteristics, as the CPU consists of four dies internally, with part of the memory attached to each die. Both machines run Ubuntu 18.04. The distributed reservoir sampling experiments use MPI (OpenMPI 4.0) for communication and were conducted on ForHLR II, a general-purpose high-performance computing cluster located at Karlsruhe Institute of Technology, using up to 256 nodes. Each node is equipped with two deca-core Intel Xeon E5-2660 v3 processors for a total of 20 cores per node, and 64 GiB of DDR4 main memory. We use one MPI process (PE) per core, ie, 20 PEs per node, for a total of up to 5120 PEs. All nodes are attached to an InfiniBand 4X EDR interconnect using an InfiniBand FDR adaptor [KIT20]. All implementations are in C++ and compiled with GNU g++ 9.2.0 (flags `-O3 -flto -march=native`).

Our measurements do not include time spent on memory allocation and mapping where sizes are known in advance.

[8] Archived copies of the code are available at `https://web.archive.org/web/20200925083205/https://codeload.github.com/lorenzhs/wrs/zip/master` and `https://web.archive.org/web/20200925083355/https://codeload.github.com/lorenzhs/reservoir/zip/master`, respectively.

Table 3.3: Overview of algorithms used in the evaluation.

Abbreviation	Section	Description
PSA	3.3.2	parallel alias tables
PSA+	3.3.2.a)	optimisation of PSA
OS	3.4	output-sensitive sampling with replacement
OS-ND	p. 89	non-deduplicating optimised variant of OS
2lvl	3.3.3	two-level alias tables
2lvl-classic		2lvl with Vose's algorithm (Algorithm 3.1) as base case
2lvl-sweep		2lvl with our sweeping algorithm (Algorithm 3.2) as base case
ours	3.7.1	our weighted reservoir sampling algorithm
ours-8		*ours* with 8 and the selection algorithm of Theorem 2.6
gather	3.7.3	centralised gathering reservoir sampling algorithm

Implementation Details. Table 3.3 lists the algorithms which we evaluate in this section. We implemented alias table construction using our parallel splitting algorithm (PSA, Section 3.3.2, as well as PSA+, the practical optimisation described in Section 3.3.2.a)). For WRS–R, we implemented the distributed output-sensitive sampling algorithm (OS, Section 3.4) in shared memory using a sequential implementation of the base method. We chose this mode of implementation as it is simpler and does not depend on shared memory, which should help on NUMA machines. A variant of OS called OS-ND, which we describe below, is also included in our implementation. We also implemented the two-level alias table construction algorithm in shared memory (2lvl, Section 3.3.3, Theorem 3.3 (i)) and sequential alias table using both Vose's method (Algorithm 3.1) and our sweeping construction (Algorithm 3.2). The 2lvl algorithm can use either of these as its base case. When using Vose's method, we refer to it as 2lvl-classic, and 2lvl-sweep when using our sweeping algorithm.

Both of the machines used require some degree of Non-Uniform Memory Access (NUMA) awareness in memory-bound applications like ours. Thus, in our SM-parallel implementations, all large arrays are distributed over the available NUMA nodes, and threads are pinned to NUMA nodes to maintain data locality. The same amount of memory and the same number of threads is assigned to each NUMA node.

For reservoir sampling, we implemented our algorithm of Section 3.7 with the amsSelect approximate multisequence selection algorithm of Section 2.3.2.b), using one (labelled *'ours'*) or eight pivots (labelled *'ours-8'*) and exact bounds ($\underline{k} = \overline{k} = k$, see Section 2.3.2.c)). We compare it to a centralised algorithm which uses the same thresholding procedure but gathers all candidate items at a designated root PE, where it uses sequential selection to maintain the sample, subsequently labelled *'gather'*. The local reservoirs are implemented as B+ trees, based on an implementation of Bingmann [Bin18b] augmented with join/split operations from an implementation by Akhremtsev [AS16] and rank/select support (see Section 1.3.4).

All pseudo-random numbers are generated with 64-bit Mersenne Twisters [MN98], using the Intel Math Kernel Library (MKL) [Int19] on the Intel machine and the cluster, and dSFMT[9] on the AMD machine.

Optimisations. For the output-sensitive algorithm OS, we use an additional optimisation that aborts the tree descent and uses the base case bucket table when fewer than 128 samples are to be drawn from at least half as many items. The resulting elements are then deduplicated using a hash table to ensure that each element occurs only once in the output. A variant without this deduplication is called OS-ND and can be useful for applications that allow the total multiplicity of sampled items to be split over several occurrences.

All alias table queries are implemented in a branchless manner. To do so, we store the bucket index and its alias in a temporary array A of size two and use indexing with the conditional to return the correct item. Using zero-based indexing and the notation of Section 3.1.2, let $A[0] = r$ and $A[1] = a_r$. The query then returns A indexed with whether the coin came up heads (0) or tails (1), which uses the result of a comparison as an index into A instead of a conditional branch. The result is an improvement of 20 % to 25 % in sequential query times (measured for $n = 10^8$ items). When using all threads, there are enough concurrent random accesses to memory at any time to hide the latency of the conditional branch, yielding a more modest single-digit percentage point increase in total throughput.

We implemented special handling of the first mini-batch of reservoir sampling to avoid inserting too many items into the reservoir if the local size b of the first mini-batch is large compared to the sample size k. While processing the first mini-batch, use the key of *local rank* k as a local threshold, and update this threshold periodically. More concretely, if $b \geq \max(1.5k, k + 500)$, we use the key of *local rank* k as the threshold for subsequent items, and refresh this local threshold every time the local reservoir exceeds $\max(1.1k, k + 250)$ items, discarding those that are larger. This maintains correctness: at no point is a local reservoir pruned to a size smaller than k, so the union of all local reservoirs is guaranteed to be of size at least k. It also maintains the property that each local reservoir is a sample without replacement of some size k' of all items seen so far at this PE.

Furthermore, to speed up the innermost loop of our reservoir sampling algorithm (Algorithm 3.6), we compute the sum of weights of 32 items at a time, check whether the amount of weight to be skipped (X) is larger than this sum, and skip all 32 items at once if this is the case. This reduces the number of conditional branches and allows for vectorisation using the CPU's Single Instruction Multiple Data (SIMD) units. Both of these factors speed up the processing of the items in a batch significantly, especially as typically, only a few items per batch end up contributing to the reservoir (and X is much larger than the average weight) once $n \gg k$ items have been processed.

Input Data. Unless specified otherwise, we use uniformly random weights from the interval $(0, 1]$ as inputs. In our query experiments, we additionally use random permutations of the weights $\{1^{-s}, 2^{-s}, \ldots, n^{-s}\}$ for a parameter s to obtain more skewed 'power-law' or Zipf-like distributions. For reservoir sampling, the weights were scaled to the interval $(0, 100]$.

[9] http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/SFMT/, version 2.2.3; an archived copy of the code is available at https://web.archive.org/web/20200515023309/http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/SFMT/dSFMT-src-2.2.3.tar.gz.

3.8.1 Sequential Performance of Alias Tables

Surprisingly, many common existing implementations of alias tables (eg, `gsl_ran_discrete_preproc` in the GNU Scientific Library (GSL) [Gal+09] or `sample` in the R project for statistical computing [R C20]) use a struct-of-arrays memory layout for the alias table data structure. By using the array-of-structs paradigm instead, we can improve memory locality, incurring one instead of up to two cache misses per query. Combined with branchless selection inside the buckets and a faster random number generator, our query is four times faster than that of GSL version 2.5 (measured for $n = 10^8$). At the same time, alias table construction using our implementation of Vose's method is 30 % faster than GSL on the Intel machine and 44 % faster on the AMD machine. Other popular statistics packages, such as NumPy (version 1.5.1, function `np.choice`) or Octave (Statistics package version 1.4.0, function `randsample`) employ algorithms with superlinear query time for WRS–R. We therefore use our own implementation of Vose's algorithm as the baseline in our evaluation.

Among our sequential implementations, construction with Vose's method is 20–25 % faster than our sweeping algorithm,[10] with the gap being slightly smaller on the AMD machine than on the Intel machine. Including the time for memory allocations narrows the gap by around two percentage points. Thus, when used purely sequentially, Vose's method remains the fastest, and we use it as the sequential baseline for our speedup experiments in Section 3.8.2.

3.8.2 Construction

Speedups compared to an optimised sequential implementation of Vose's alias table construction algorithm are shown in Figure 3.9 (strong scaling with fixed $n = 10^8$ and $n = 10^9$ as well as weak scaling with $n/p = 10^7$ uniform random variates).[11] Observe that for $n = 10^8$, the per-thread input size is quite small for the high thread counts achievable on the Intel machine, resulting in somewhat noisy measurements (Figure 3.9 (a)). Note that OS-ND is not shown, as it only differs from OS in the query phase, not during construction. Speedups do not increase further once the machine's memory bandwidth is saturated, limiting the speedup that can be achieved with techniques that require multiple passes over the data (PSA, 2lvl-classic). Unlike PSA (and what PSA+ attempts to mitigate), 2lvl can be constructed almost independently by the PEs and requires much fewer accesses to memory. Sequentially, Vose's method outperforms our sweeping algorithm (see Section 3.8.1). However, when used as the base case of 2lvl, our algorithm scales much better to high thread counts because it reduces the memory traffic and since hyper-threading (HT) helps to hide the overhead of branch mispredictions. For large per-thread inputs, 2lvl-sweep achieves more than twice the speedup of 2lvl-classic. On random inputs such as the ones used here, PSA+ can greedily process almost the entire input, and thus achieves excellent speedups.

[10] In the conference paper this chapter is based on, Ref. [HS19d], we reported a somewhat smaller gap. We have since further optimised our implementation of Vose's method. Specifically, the optimisation to make queries branchless was previously implemented by storing both the index and the alias in the table, instead of using a temporary array of size two at query time.

[11] As Figure 3.9 shows only speedup values, we here provide the running times of the sequential baseline with regard to which they are given. These are, from (a) to (f), 1.14 *s*, 1.35 *s*, 11.4 *s*, 13.5 *s*, 114 *ms*, and 136 *ms*.

Preprocessing for OS introduces some overhead but requires fewer passes over memory than PSA and achieves approximately twice the speedups as a result on the Intel machine. On the AMD machine, its scaling behaviour is initially similar to that of PSA or slightly worse, however, where PSA's speedups begin to stagnate, OS continues to scale well. This continues even into hyper-threading, which yields much larger relative improvements for OS construction on the AMD machine than on the Intel machine (compare Figures 3.9 (c) and 3.9 (d)). However, due to the Intel machine's higher overall memory bandwidth, the AMD machine does not achieve the same speedups for memory-bound construction algorithms, even when using the same number of cores on both machines. We can also see that by processing each group of each thread independently, OS makes good use of the cache: the large non-inclusive L3 caches of the Intel machine gives it a boost for $n = 10^8$ (Figure 3.9 (a)). Once groups no longer fit into the CPU's caches ($n = 10^9$, Figure 3.9 (c)), speedups are somewhat lower. To a lesser extent, we can also observe this effect on the AMD machine (Figures 3.9 (b) and 3.9 (d)).

In the weak scaling experiments (Figures 3.9 (e) and 3.9 (f)), we again see clearly how 2lvl-classic and PSA are limited by memory bandwidth. Using more than two threads per available memory channel (4×6 for the Intel machine, 8 for the AMD machine) yields nearly no additional benefit for these algorithms. Meanwhile, 2lvl-sweep, PSA+, and—to a lesser extent—OS are not so much limited by the available memory bandwidth as by the latency of memory accesses. As a result, they continue to scale well, even for the highest thread counts.

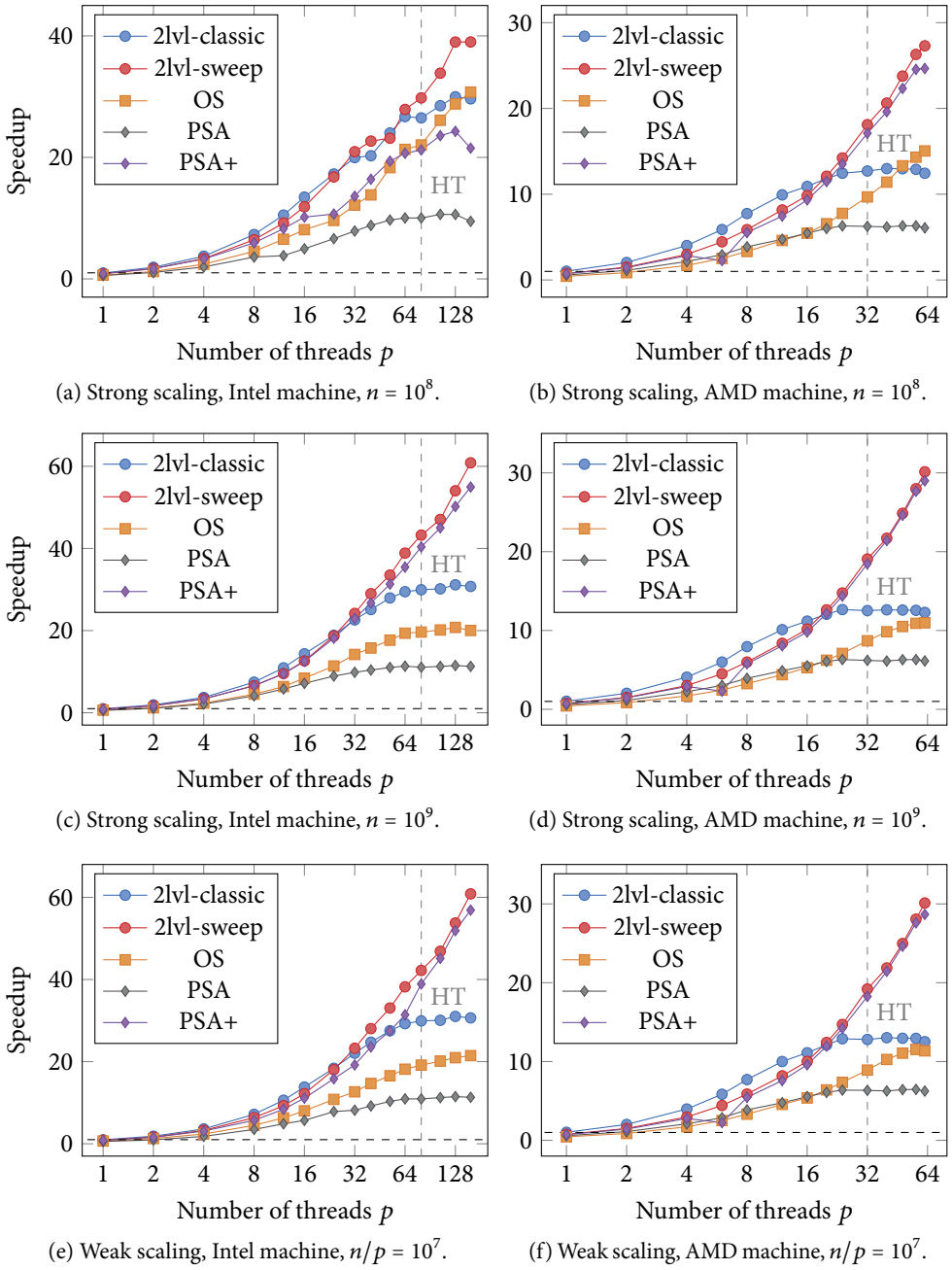
3.8.3 Queries

We performed strong and weak scaling experiments for queries (Figure 3.10) as well as throughput measurements for different sample sizes (Figure 3.11). Observe that the different variants of our 2lvl, 2lvl-classic and 2lvl-sweep, differ only in how they approach construction and have identical query behaviour. The same is true of PSA and PSA+. Thus, we do not consider them separately in the query evaluation.

3.8.3.a) Scaling

First, consider the scaling experiments of Figure 3.10.[12] These experiments were conducted on the Intel machine, as its highly non-uniform memory access characteristics highlight the differences between the algorithms. All speedups are given relative to sampling sequentially from an alias table. The strong scaling experiments (Figures 3.10 (a) and 3.10 (b)) deliberately use a small sample size to show scaling to low per-thread sample counts ($\approx 64\,000$ samples per thread for 158 threads). We can see that all algorithms have good scaling behaviour. Hyper-threading (marked 'HT' in the plots) yields additional speedups, as it helps to hide memory latency. This already shows that the bottleneck is random access latency to memory. Sampling from PSA and 2lvl is done completely independently by all threads, with no interaction apart from reading from the same shared data structures. Because the Intel machine has four NUMA nodes, most queries have to access another NUMA node's memory. This limits the speedups achievable using alias tables (PSA, 2lvl).

[12] The sequential processing times with regard to which the speedups are reported are, from (a) to (d), 215 *ms*, 220 *ms*, 21.3 *ms*, and 21.4 *ms*.



(a) Strong scaling, Intel machine, $n = 10^8$.

(b) Strong scaling, AMD machine, $n = 10^8$.

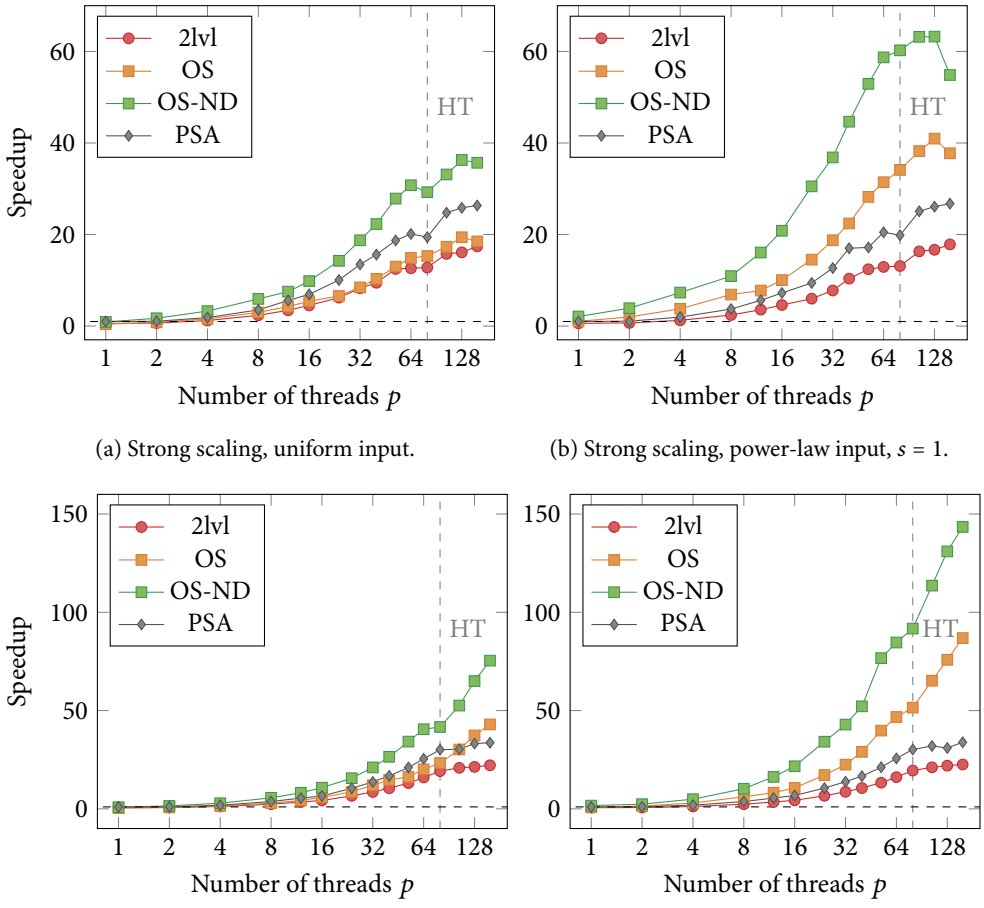
(c) Strong scaling, Intel machine, $n = 10^9$.

(d) Strong scaling, AMD machine, $n = 10^9$.

(e) Weak scaling, Intel machine, $n/p = 10^7$.

(f) Weak scaling, AMD machine, $n/p = 10^7$.

Figure 3.9: Strong (top and middle) and weak (bottom) scaling evaluation of parallel alias table construction techniques. Strong scaling with input sizes $n = 10^8$ (top) and $n = 10^9$ (middle), weak scaling with $n/p = 10^7$. Speedups are relative to our optimised implementation of Vose's method, Algorithm 3.1. Algorithm names as in Table 3.3.



(a) Strong scaling, uniform input.

(b) Strong scaling, power-law input, $s = 1$.

(c) Weak scaling, uniform input.

(d) Weak scaling, power-law input, $s = 1$.

Figure 3.10: Query strong and weak scaling for $n = 10^9$ input elements. Sample size for strong scaling $k = 10^7$, per-thread sample size for weak scaling $k/p = 10^6$. All speedups are relative to sampling from an alias table sequentially. Intel machine. Algorithm names as in Table 3.3.

In contrast, for OS and OS-ND, threads access only local memory after a shared top-level sample assignment stage. This benefits scaling, especially on NUMA machines and machines with local last-level caches, such as newer AMD CPUs. As a result, OS-ND achieves the best speedups, despite this benchmark producing very few samples with multiplicity greater than one (Figure 3.10 (a), where the sample size is 1 % of the input size). On the other hand, deduplication in the base case of OS has considerable overhead, making OS roughly 25 % slower than sampling from an alias table for such inputs, even sequentially. However, preliminary experiments show

that implementing OS with hash-table based deduplication remains significantly faster than fully descending the DC-trees, ie, omitting the optimisation described on page 89.

The weak scaling experiments of Figures 3.10 (c) and 3.10 (d) show even better speedups because many more samples are drawn here than in our strong scaling experiment, reducing overheads. Sampling from a classical alias table (PSA) achieves a speedup of over 30 here, again limited by memory accesses rather than computation. Meanwhile, the output-sensitive methods (OS, OS-ND) reap the benefits of accessing only local memory.

Comparing the results for uniformly random weights (Figures 3.10 (a) and 3.10 (c)) with those for power-law inputs (Figures 3.10 (b) and 3.10 (d)) clearly shows that the query performance of alias tables (PSA, 2lvl) is independent of the input's weight distribution. Meanwhile, for OS and its variant OS-ND, uniformly random weights are a difficult input because they result in few items with multiplicity greater than one (see the throughput measurements below for further details). Thus, they achieve significantly higher speedups for the power-law input.

3.8.3.b) Throughput

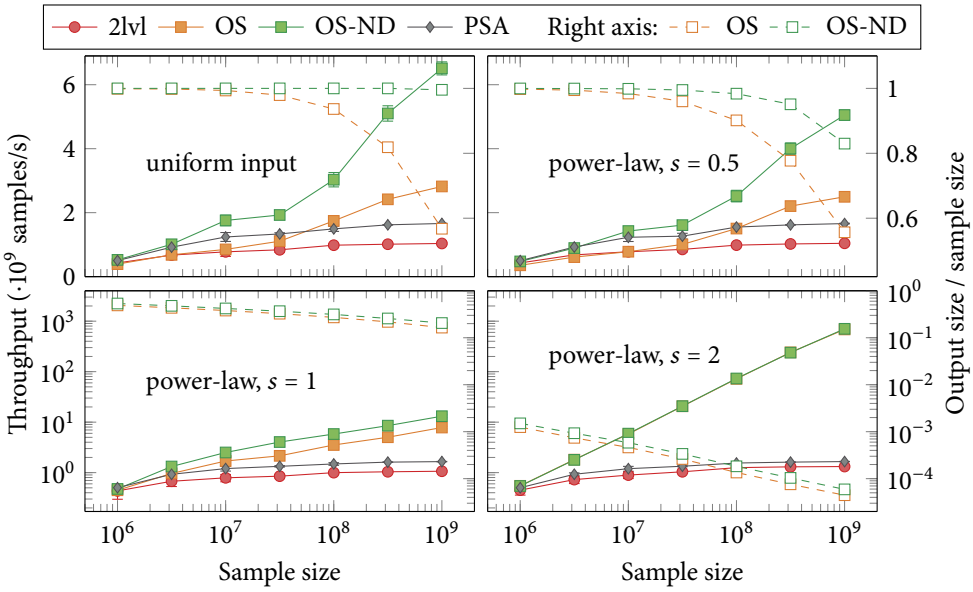
Figure 3.11 shows the query throughput of the different approaches (note the logarithmic y-axes in the lower plots). We can see that 2lvl suffers a significant slowdown compared to PSA on all inputs since an additional query for a meta-item is needed (this is also clearly visible in Figure 3.10). Its throughput is around 40 % lower than sampling from an alias table. Nonetheless, both algorithms are limited by the latency of random accesses to memory for the (bottom) alias tables on both machines.

As long as the sample contains few duplicates (*cf.* the dashed lines with the scale on the right y-axis, which belong to the solid lines of the same colour and marker shape), the cost of base case deduplication in OS exceeds the benefits of increased memory locality. On the AMD machine, where memory access locality is less important, this results in higher throughput for 2lvl than for OS for small sample sizes when inputs are not too skewed (Figure 3.11 (b)). As expected, when there are many duplicates, the output-sensitive algorithms (OS and OS-ND) perform very well. Omitting base case deduplication (OS-ND) doubles throughput for uniform inputs and does no harm for skewed inputs, making OS-ND the consistently fastest algorithm.

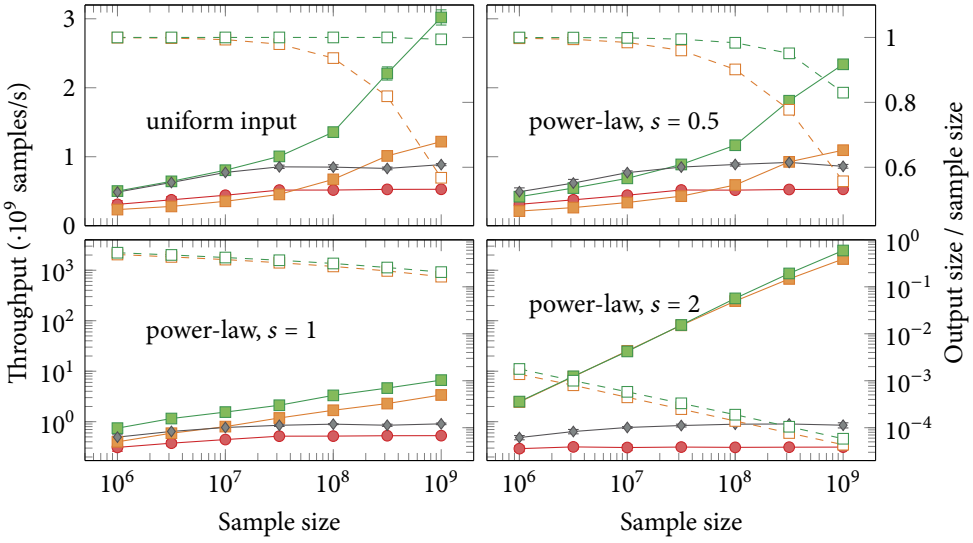
Even when OS is slower than sampling from an alias table, adding sequential deduplication to normal alias tables using a fast hash table (Google's `dense_hash_map`[13]) is not competitive. A preliminary experiment with uniform inputs ($n = 10^8$ items and $k = 10^7$ samples) resulted in sequential deduplication increasing the time for sampling by a factor of five to six compared to simply storing samples in an array without deduplication. This clearly demonstrates that locally deduplicating already partitioned data with many small hash tables, as is done in OS, is much faster than global deduplication using a single large hash table.

Lastly, we observe that 2lvl and PSA throughput levels off after $10^{7.5} \approx 3 \cdot 10^7$ samples on the AMD machine (3.11 (b)), whereas it keeps increasing slightly on the Intel machine (3.11 (a)). This is likely due to the Intel machine's higher overall memory bandwidth.

[13] `https://github.com/sparsehash/sparsehash`, version 2.0.2, archived at `https://web.archive.org/web/20200925092936/https://codeload.github.com/sparsehash/sparsehash/tar.gz/sparsehash-2.0.2`

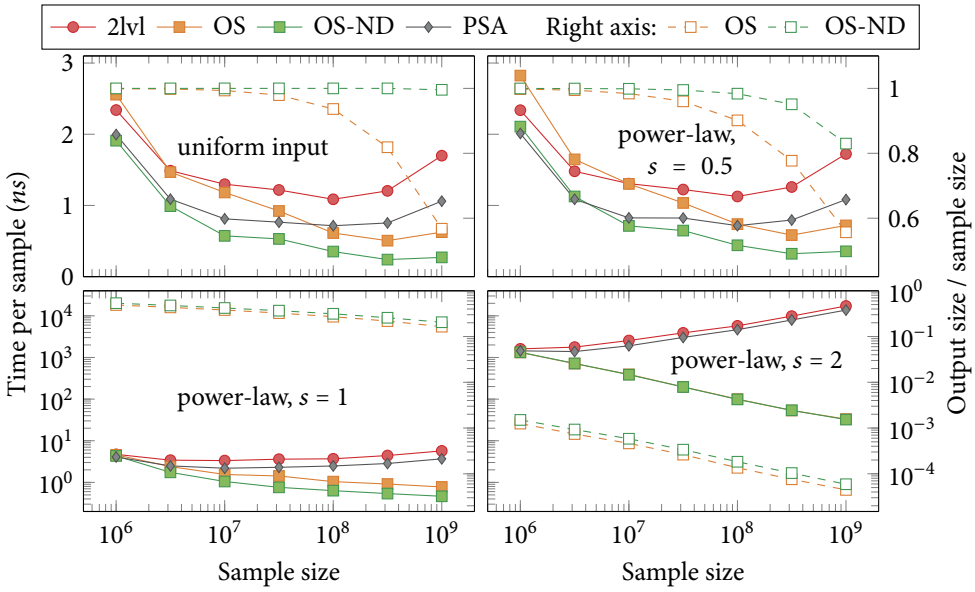


(a) Intel machine (158 threads). Note the logarithmic y-axes for the bottom plots.

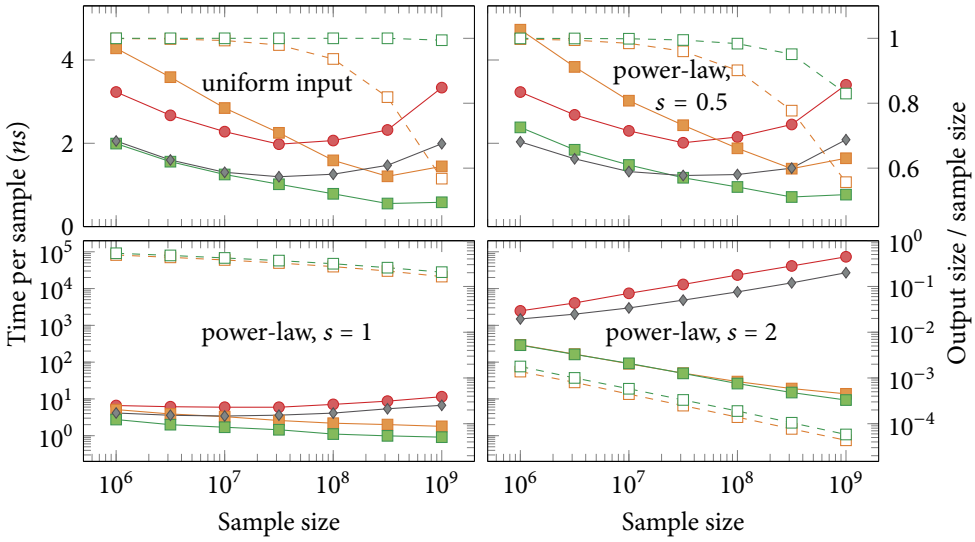


(b) AMD machine (62 threads). Note the logarithmic y-axes for the bottom plots.

Figure 3.11: Query throughput of the different methods for $n = 10^9$, using all available cores. Top left: uniform inputs, top right: power-law with $s = 0.5$, bottom left: power-law $s = 1$, bottom right: power-law $s = 2$. Dashed lines on the right y-axis belong to the same-coloured solid lines on the left y-axis and show fraction of output size over sample size for output-sensitive algorithms. Algorithm names as in Table 3.3.



(a) Intel machine (158 threads). Note the logarithmic y-axes for the bottom plots.



(b) AMD machine (62 threads). Note the logarithmic y-axes for the bottom plots.

Figure 3.12: Time per unique output item for the different methods for $n = 10^9$ using all available cores on the Intel (top) and AMD (bottom) machines. Top left: uniform inputs, top right: power-law with $s = 0.5$, bottom left: power-law $s = 1$, bottom right: power-law $s = 2$. Dashed lines on the right y-axis belong to the same-coloured solid lines on the left y-axis and show fraction of output size over sample size for output-sensitive algorithms. Algorithm names as in Table 3.3.

3.8.3.c) Time per Unique Item

Figure 3.12 shows the time per unique item in the sample. This metric penalises algorithms for emitting an item with multiplicity greater than one multiple times, which affects the methods based on alias tables considerably when the number of unique samples is low. We can see that the 2lvl and PSA approaches do well as long as few items have multiplicity larger than one, ie, the dashed lines are close to 1 (right scale). In these cases, what OS gains from having higher locality of memory accesses is lost in base case deduplication, especially on the AMD machine. Because it may split items' total multiplicity over several occurrences by omitting base case deduplication, OS-ND does not suffer from this and is the fastest algorithm. The same is true for the power-law inputs with $s = 0.5$ and $s = 1$ (observe that as in Figure 3.11, the y-axes for the lower two plots are logarithmic). For power-law inputs with $s = 2$, the running time of OS and OS-ND is nearly constant regardless of sample size. This is because the number of unique items is very low for this input (measured in the low thousands), and thus what little time is spent on sampling is dominated by thread synchronisation and scheduling overhead. These overheads are particularly problematic with the 158 threads on the Intel machine (Figure 3.12 (a)), where they add up to several milliseconds, around ten times as much as on the AMD machine (Figure 3.12 (b)).

3.8.4 Reservoir Sampling

We now present the results of strong and weak scaling experiments with our weighted reservoir sampling algorithm on a supercomputer and provide detailed measurements on the composition of the running time of the algorithms. Throughout this section, our weighted reservoir sampling algorithm with single-pivot selection is referred to as *'ours'*, its version with multi-pivot selection with 8 pivots as *'ours-8'*, and the centralised gathering algorithm as *'gather'* (see also Table 3.3).

Each experiment was repeated ten times, with each run lasting 30 seconds, completing as many mini-batches as possible in that time. Input generation is not included in the reported times. We use uniformly random floating-point weights from the interval $(0, 100]$ as inputs. Preliminary experiments with skewed weights—normally distributed with the mean increasing based on the mini-batch sequence number and the PEs' ranks—showed no significant differences in running time. Speedups are reported relative to our algorithm with single-pivot selection (*'ours'*) on a single node ($p = 20$ cores/PEs). Based on preliminary experiments, we chose $d = 8$ as the number of pivots used in the multi-pivot amsSelect selection algorithm with exact output rank (see Section 2.3.2.c)).

3.8.4.a) Weak Scaling

The results of a scaling experiment are shown in Figure 3.13, with plots in three rows for per-PE mini-batch sizes b—from top to bottom—10^6, 10^5, and 10^4, and three columns with sample sizes k of 10^3, 10^4, and 10^5 items, from left to right.[14] The figure shows the efficiency of the

[14] The single-node times per batch with regard to which the speedups are reported are, row-wise from top left to bottom right, 3.10 *ms*, 3.14 *ms*, 3.24 *ms*; 116 μs, 122 μs, 131 μs; 16.2 μs, 16.6 μs, 19 μs.

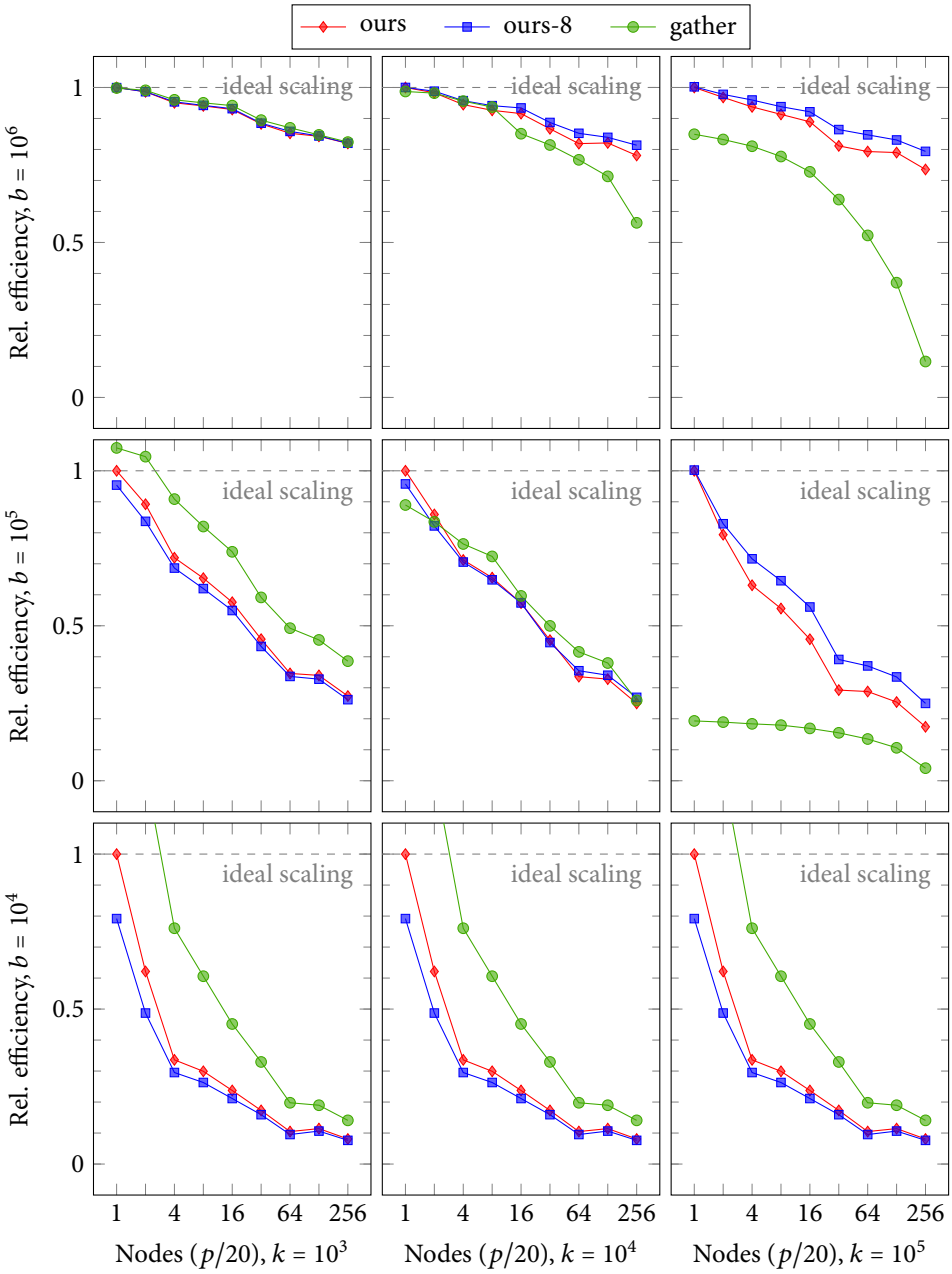


Figure 3.13: Reservoir sampling, weak scaling with different batch (rows) and sample sizes (columns). Efficiency (speedup divided by number of PEs) is measured relative to our algorithm with single-pivot selection (‘*ours*’) on 1 node (20 cores).

algorithm, measured relative to our algorithm with single-pivot selection (*'ours'*) for the same batch and sample sizes on a single node ($p = 20$ cores/PEs).

We can see that our algorithm shows good scaling, especially for larger mini-batch sizes. Using multiple pivots for selection (*'ours-8'*, in blue) is especially beneficial for larger sample sizes (centre and right columns), where it reduces average recursion depth by a factor of around 2.5—from 7.3 to 2.7 for $k = 10^5$ and from 4.3 to 1.8 for $k = 10^4$—compared to a much smaller improvement from 1.9 to 1.1 for $k = 10^3$ (left column), where the average recursion depth is already very low when using a single pivot. This results in selection running time improvements of up to 35 % for $k = 10^5$ and around 20 % for $k = 10^4$, with no significant improvement for $k = 10^3$. Because local processing is a significant part of overall processing time (refer to the running time composition analysis of Section 3.8.4.c) for details), the actual overall running time improvement is only around 8 % ($k = 10^5$ and $b = 10^6$). As expected, smaller samples (left column) achieve slightly better speedups than larger ones (right column). The causes for this lie in the $\mathcal{O}(\log k \log p)$ latency of the selection algorithm and increased local processing time due to larger local reservoirs.

For $b = 10^4$, the costs for a single collective communication operation exceed local processing time for larger numbers of PEs, and all algorithms struggle. Because it only needs to execute a single gather operation on extremely few new candidates—most PEs do not contribute any new candidates in most batches—, algorithm *'gather'* has lower communication overhead in this scenario. However, its efficiency also leaves much to be desired. When comparing the performance of all algorithms for the different batch sizes, our key takeaway is that processing fewer than 10^5 items per PE per mini-batch should be avoided with a large number of nodes if throughput is important. These inputs are too small for distributed processing to be efficient.

We also see clearly that the centralised algorithm (*'gather'*, in green) performs well only when the batch size is unrealistically small (bottom row) or the sample size is very small ($k = 10^3$, left column), where few candidates need to be gathered per batch. It begins struggling even with $k = 10^4$ for large batch sizes. For larger sample sizes ($k = 10^5$, right column), it performs poorly unless batches are tiny (bottom right). Its performance for $b = 10^5, k = 10^5$ is a reproducible outlier and is likely due to the MPI library's internal algorithm choice for the gather step. All algorithms show better—and in the case of our algorithm, near-optimal—scaling for large batch sizes, as communication overhead is much less noticeable than for small batches, where local processing is fast.

Overall, *'ours-8'* is the most consistently fast algorithm, as we see that selection using multiple pivots yields better results than the variant using single-pivot selection (*'ours'*) for large batch and sample sizes. Its performance trails that of *'ours'* only when all three of batch size, sample size, and number of PEs are comparatively small.

3.8.4.b) Strong Scaling

Our strong scaling experiments use the same sample sizes of $k = 10^3$, 10^4, and 10^5 items, while keeping the global batch size $B = b \cdot p$ fixed at $B = 2^{10} \cdot 10^4 \approx 10^7$, $2^{10} \cdot 10^5 \approx 10^8$, and $2^{10} \cdot 10^6 \approx 10^9$ items. These sizes are chosen so that the number of PEs p divides them evenly for the power-of-two numbers of 20-core machines used in our experiments.

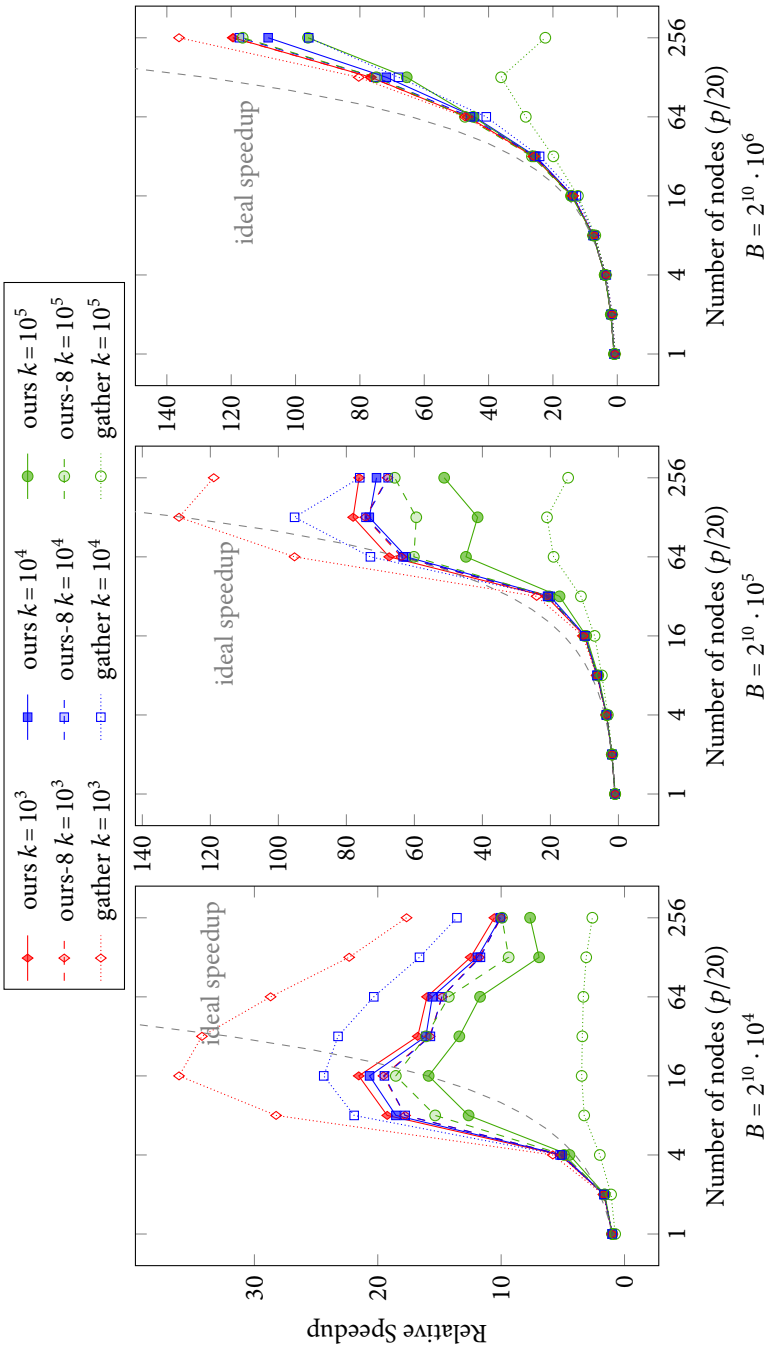


Figure 3.14: Strong scaling, speedups relative to our algorithm with single-pivot selection (*'ours'*) for the same sample size on 1 node (20 cores).

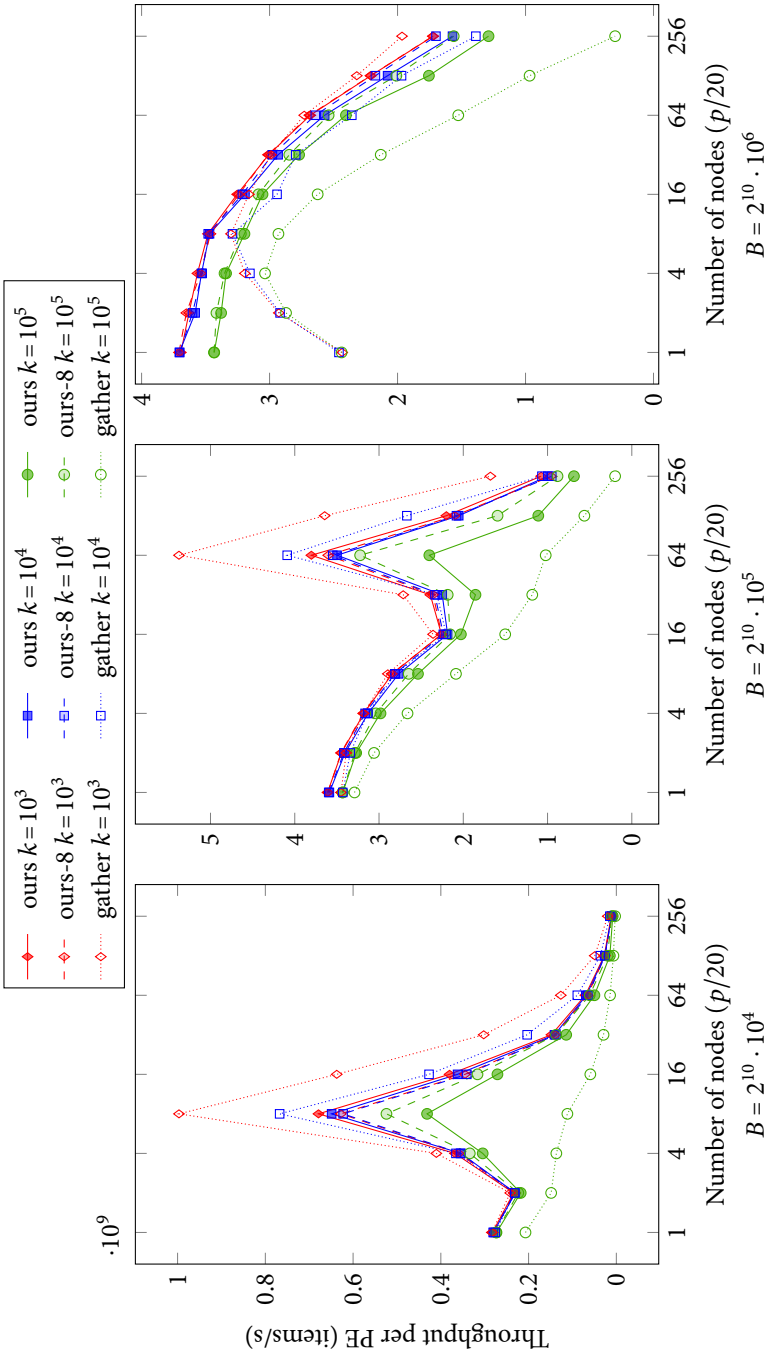
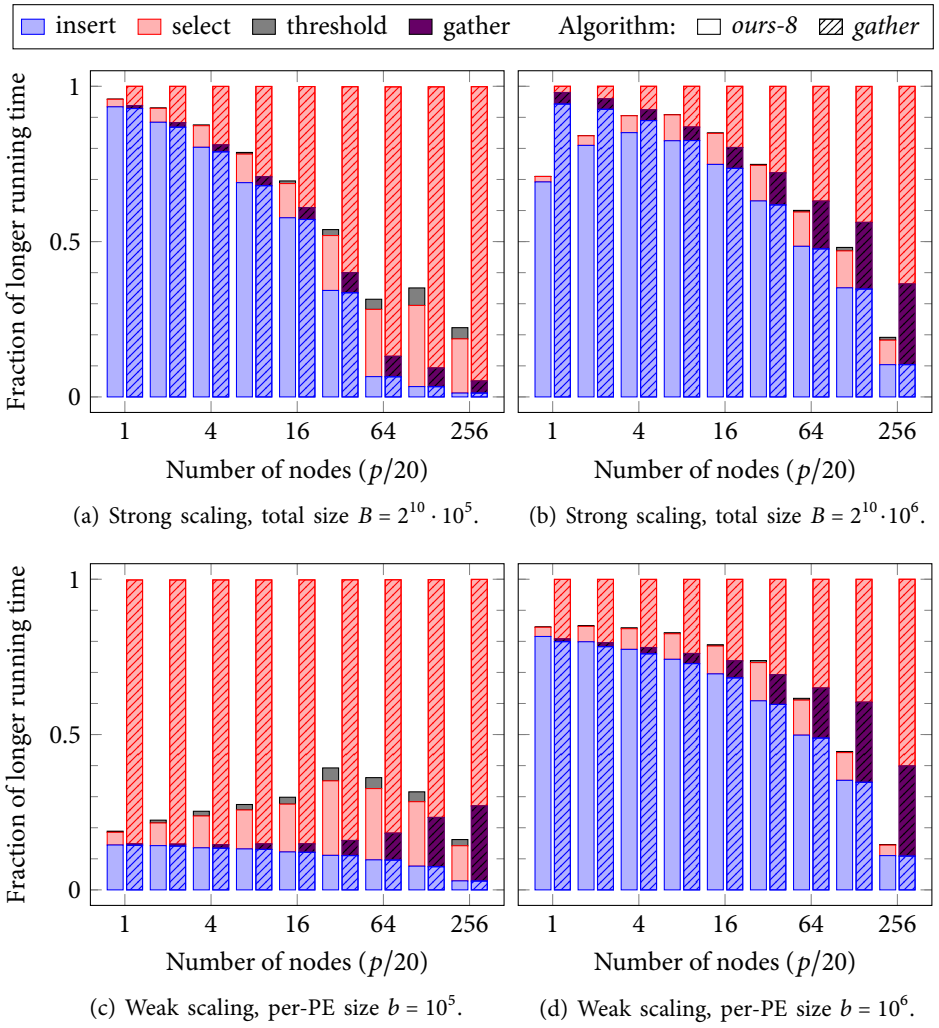


Figure 3.15: Strong scaling, items per PE per second (excluding input generation).



(a) Strong scaling, total size $B = 2^{10} \cdot 10^5$.

(b) Strong scaling, total size $B = 2^{10} \cdot 10^6$.

(c) Weak scaling, per-PE size $b = 10^5$.

(d) Weak scaling, per-PE size $b = 10^6$.

Figure 3.16: Composition of running time, normalised to the slower algorithm; strong scaling with $B = bp = 2^{10} \cdot 10^5$ (top left) and $2^{10} \cdot 10^6$ (top right) items; weak scaling with $b = 10^5$ (bottom left) and 10^6 (bottom right) items per PE; sample size $k = 10^5$. Left bars: our algorithm using selection with 8 pivots (*'ours'*), right bars with diagonal lines: centralised algorithm (*'gather'*).

Figure 3.14 shows the relative speedups for a strong scaling experiment with the above configurations.[15] We again observe that using multiple pivots for selection (*'ours-8'*) provides

[15] The single-node times per batch with regard to which the speedups are reported are, with the sample sizes in ascending order, 1.813 *ms*, 1.828 *ms*, 1.875 *ms* for $B = 2^{10} \cdot 10^4$; 14.18 *ms*, 14.27 *ms*, 14.92 *ms* for $B = 2^{10} \cdot 10^5$; 138.6 *ms*, 138.2 *ms*, 149.2 *ms* for $B = 2^{10} \cdot 10^6$.

significant benefits for large sample sizes (green lines) and does not make much difference for smaller ones (red and blue lines). The centralised algorithm again works well only for small sample sizes and fails to provide any significant speedup for large samples.

We also see that as long as the local batch size (ie, input size per PE) is too large to fit into the CPU's caches—more than around 10^5 items—speedups increase well, before abruptly jumping once local processing happens in cache. For the smallest batch size, this effect even exceeds what would be the ideal speedup. This is a classical superlinear speedup due to larger available cache resources. Once the data fits into cache, local processing time—which previously was a significant factor—collapses, and now represents only a very small part of the overall time. Initially, this leads to a superlinear speedup. As communication in the selection process becomes the dominant factor in overall running time, speedups slowly decline as the $\mathcal{O}(\log k \log p)$ messages required for the selection dominate running time as the number of PEs grows. Figure 3.15 shows the throughput per PE, ie, how many items are processed at every PE per second and confirms this. It clearly shows the momentary advantage of processing inputs that just fit into cache, but are large enough to keep the fraction of running time spent on selection low. Once this advantage passes, the decline in throughput per PE once again follows the predicted curve, dominated by the communication cost of selection.

3.8.4.c) Running Time Composition

Figure 3.16 shows the composition of running times for our weak and strong scaling experiments with two different batch sizes and $k = 10^5$ samples. Each pair of bars—our algorithm using selection with 8 pivots (*'ours-8'*), and to its right, the centralised gathering algorithm (*'gather'*), marked with diagonal lines—is normalised to the slower of the two algorithms, which is always *'gather'* in these experiments.

Figures 3.16 (a) and 3.16 (b) present the results for strong scaling. We can see that the running time share of locally processing the input declines as expected, and selection becomes the dominant factor in our algorithm. In the centralised algorithm, however, the amount of time spent on gathering the candidate items grows rapidly, especially for the larger batch size (Figure 3.16 (b)). For the smaller batch size, sequential selection dominates the running time of *'gather'* when using many nodes, as only $b = B/p = 20\,000$ items are processed per PE and batch when using 256 nodes (5120 PEs). This is much faster than selecting the $k = 10^5$ smallest values out of little more than k candidates (the 10^5 previously best items plus fewer than 300 new candidates per mini-batch on average for 128 and 256 nodes in this experiment).

The results for weak scaling are shown in Figures 3.16 (c) and 3.16 (d). For the smaller batch size (Figure 3.16 (c)), selection dominates the gathering algorithm's running time from the beginning. Our algorithm is consistently more than twice as fast, and typically up to four times faster. While this gap shrinks for 32 and 64 nodes (640 and 1280 PEs, respectively), it grows again as the centralised algorithm's time spent on gathering the candidates increases, even though on average, less than 0.1 new candidates per PE are gathered. The results for the larger batch size, shown in Figure 3.16 (d), are also as expected. While our algorithm requires slightly more time for sequential processing—the candidates have to be inserted into a B+ tree instead of stored in an array—the centralised algorithm's selection and gathering become unsustainably slow for large numbers of nodes.

3.9 Application: Generating R-MAT Graphs

In this section, we demonstrate how weighted sampling can be applied to the problem of generating graphs according to the R-MAT ('Recursive MATrix') model. We only provide a short overview here, as the full details of the technique are not pertinent to the focus of this dissertation. Instead, we refer the reader to our paper, Ref. [HS20b], for additional details and discussion. The paper is joint work with Peter Sanders, and parts of this section are copied verbatim from this paper.

R-MAT is a simple, widely used model for generating graphs with a power-law degree distribution, a small diameter, and a community structure that is somewhat similar to complex real-world networks. Its simplicity also allows theoretical analysis of its properties, eg, diameter, clustering coefficient, and degree distribution [MX11; Les+10]. It is particularly attractive for generating very large graphs because edges can be generated independently by an arbitrary number of processors. Therefore, R-MAT is used in the Graph 500 benchmark [Mur+10], which is the most well-known supercomputer benchmark for graph algorithms and even nonnumerical algorithms in general.

However, current R-MAT generators need time logarithmic in the number of nodes for generating an edge, as they generate one bit at a time for node IDs of the connected nodes in constant time each. Recent results on communication-free graph generation [SS16; Fun+19; Blä+19] put the role of R-MAT as the most scalable graph generator into question. Other models—refer to Refs. [Gol+10; DT20; Pen+20] for an overview of the broader literature on generating graphs—with similar properties, such as Barabasi–Albert (BA) preferential attachment graphs [BA99; SS16] or random hyperbolic graphs (RHG) [Kri+10; Fun+19; Blä+19], can now be generated in a massively parallel fashion using only linear work. By accelerating R-MAT generation by a logarithmic factor, we put things back to what one would expect. We achieve constant time per edge by precomputing pieces of node IDs of logarithmic length.

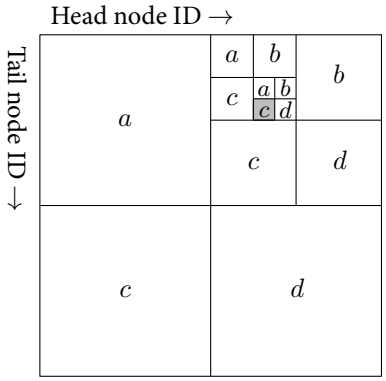


Figure 3.17: Example of R-MAT's recursive partitioning process to a depth of four. The marked square corresponds to a node ID prefix of $\texttt{0011}_2$ for the tail node and $\texttt{1010}_2$ for the head node. This square has probability $b \cdot a \cdot d \cdot c$ of being chosen.

These pieces can then be sampled in constant time using an alias table. This simple technique leads to practical improvements by an order of magnitude. This further pushes the limits of attainable graph size and makes generation overhead negligible in most situations.

Formal Definition. The simplest variant of R-MAT defines a directed multi-graph $G = (V, E)$ with $n = 2^k$ nodes and m edges based on 4 parameters a, b, c, and d with $a + b + c + d = 1$. Each edge is drawn independently at random using the following recursive process. Consider the $n \times n$ adjacency matrix M of G where $M_{ij} = 1$ if $(i, j) \in E$ and $M_{ij} = 0$ otherwise. M is split into four quadrants. The edge is placed in the upper left quadrant with probability a, in the upper right with probability b, in the lower left with probability c, and in the lower right quadrant with probability d. The process repeats this subdivision k times until a single entry of the adjacency matrix is determined. An example is shown in Figure 3.17.

Our Algorithm. For generating R-MAT graphs, we use an alias table to store precomputed subpaths of the recursive process described above. For some constant $\varphi < 1$, we precompute $u = n^\varphi$ paths together with their probabilities. By sampling and concatenating the resulting paths, we generate $\Theta(\log n)$ address bits of adjacency array entries in each iteration. The simplest such strategy generates, for a fixed length $\ell < 0.5 \log n$, all $u = 4^\ell$ paths of a length ℓ. These paths can be generated using a simple recursive procedure in time $\mathcal{O}(u)$. The function enumItems in Algorithm 3.7 outputs each path representing it as a tuple (i, j, p) where i stands for ℓ bits of the row index of the adjacency matrix, j for ℓ bits of the column index, and p for the probability that this path is generated.

The resulting sequence of partial paths E is then preprocessed into an alias table A that allows constant time sampling. Full row and column indices of adjacency array entries can then be generated by repeatedly sampling pieces of row and column indices and concatenating them until $k = \log n$ bits are available. After at most $\lceil k/\ell \rceil \in \mathcal{O}(1)$ iterations, a new edge will be completed. By reusing leftover bits from generating previous edges, we can indeed see that generating B edges will take $\lceil kB/\ell \rceil$ sampling operations. The function genRMAT in Algorithm 3.7 implements this approach to generate a batch of B edges.

Overall, we get the following result:

Theorem 3.21 (Linear-Work R-MAT generation)

The above algorithm computes an R-MAT graph with $m \in \Omega(n)$ edges in time $\mathcal{O}(m)$.

Proof. The correctness of the algorithm is evident from the above description. Since the number of items is $4^\ell < 4^{0.5 \log n} = 2^{\log n} = n \in \mathcal{O}(m)$, generating items and preprocessing them into an alias table can be done in time $\mathcal{O}(m)$ [Vos91]. As argued above, generating each edge takes constant time. □

In the technical report [HS19b], we also describe a generalisation that generates bit strings of non-uniform length. This might be faster for highly skewed generator matrices.

Experiments. We perform experiments on a machine with two Intel Xeon E5-2683 v4 processors with 16 processing cores each. With hyper-threading (two hardware-supported threads on each core) this means we can use up to 64 threads. The machine uses Ubuntu 18.04 as its operating system. Our implementation is written in C++ and compiled with the GNU C++

Algorithm 3.7 : Our R-MAT graph generator. The values a, b, c, and d are parameters of the distribution. This pseudocode uses strings of bits for clarity where a real implementation would process machine words in a bit-parallel fashion in constant time. Operator '$\cdot$' stands for string concatenation.

Input : i, j prefixes of the current path, initially empty bit strings; p the current path's probability, initially 1.0; ℓ, the number of levels to expand
Output : triples (i, j, p) of partial paths

```
1  Function enumItems(i, j, p, ℓ)
2      if ℓ = 0 then output item (i, j, p)
3      else
4          enumItems(i·0, j·0, ap, ℓ − 1); enumItems(i·0, j·1, bp, ℓ − 1)
5          enumItems(i·1, j·0, cp, ℓ − 1); enumItems(i·1, j·1, dp, ℓ − 1)
```

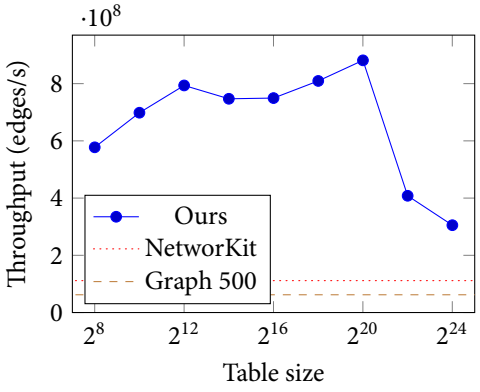
Input : B, the number of edges to generate

```
6  Function genRMAT(B)
7      ℓ := 0.25 log n                    — tuning parameter, constraint: ℓ = c log n, c < 0.5
8      A := alias table for the items generated by enumItems(⟨⟩, ⟨⟩, 1.0, ℓ)
9      i := j := ⟨⟩                       — fragments of row and column index bit strings, initially empty
10     e := 1
11     while e ≤ B do                                          — Generate a batch of B edges.
12         (i', j') := A.sample()                              — get more bits using alias table A
13         i := i·i';   j := j·j'                              — append them to known bits
14         if |i| ≥ k then                                     — enough bits for a new edge
15             output edge (i[1..k], j[1..k])
16             i := i[k+1..|i|];   j := j[k+1..|j|]            — reuse remaining bits for next edge
17             e := e + 1
```

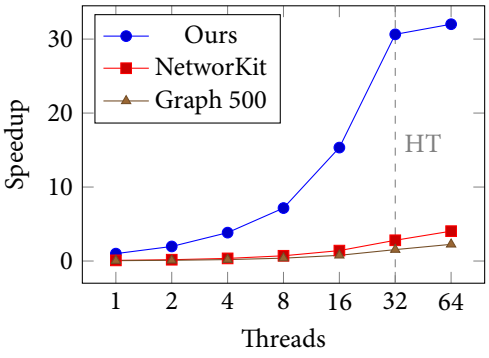
compiler g++ in version 8.2.0. Its source code is available at `https://github.com/lorenzhs/wrs/tree/rmat`.[16] We compare with the R-MAT generators used in the Graph 500 benchmark reference implementation [Bad+20] and the NetworKit toolkit for large-scale network analysis [SSM16]. We removed the code for making the graph undirected and for scrambling node IDs to concentrate on the core task of R-MAT: edge generation. The modified code is also available under the above link.

Figure 3.18 (a) shows the throughput using 64 threads as a function of the available alias table size. For $n = 2^{30}$ nodes, we generate 10^{11} edges overall, assigning blocks of 2^{16} edges at a time to the threads. We see two peaks in throughput which correspond to fully using L2 cache and L3 cache, respectively. Overall, the best throughput is up to 881 million edges per second. This is 14.2 times faster than the Graph 500 generator and 7.9 times faster than NetworKit. On

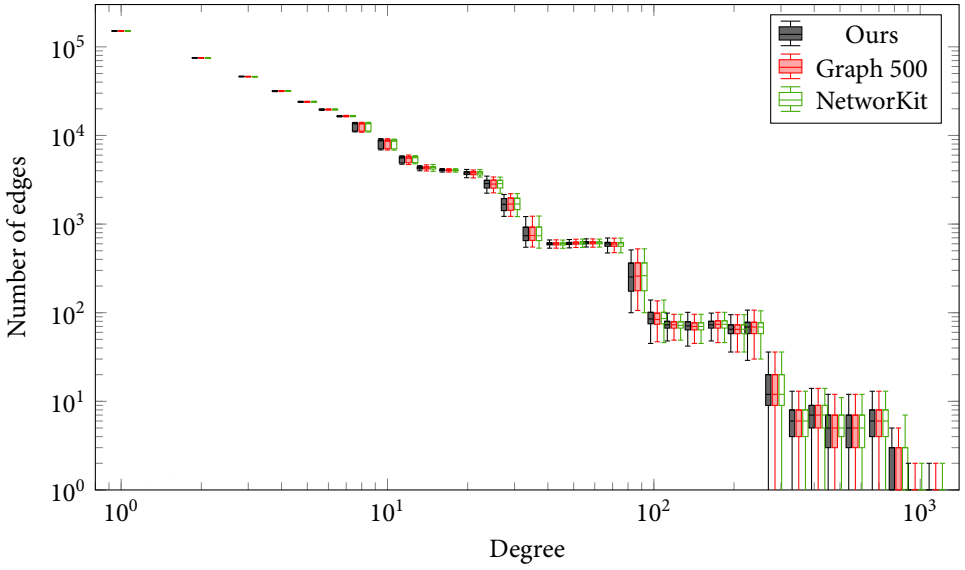
[16] An archived copy of the code is available at `https://web.archive.org/web/20200925083536/https://codeload.github.com/lorenzhs/wrs/zip/rmat`.



(a) Throughput of R-MAT generator as a function of table size, using 64 threads.



(b) Speedup relative to the fastest sequential algorithm.



(c) Degree distribution of the generated graphs. Aggregated over bins of width $\lfloor \log_2(d) - 2 \rfloor$ for degree d, 500 runs with $m = 10^7$ edges and $n = 2^{20}$ nodes each. Box plot with quantiles 0.25, 0.5, and 0.75, with whiskers to 1.5 × inter-quartile range, with small horizontal offsets to *Ours* and *NetworKit* for readability.

Figure 3.18: R-MAT experiments.

an AMD Epyc 7551P with 32 cores (64 hardware threads; see page 87 for more details on the machine), we get similar results: slightly better overall performance than on the Intel machine, with the best performance when the L3 cache on each chiplet is fully used.

Figure 3.18 (b) gives the speedup over our algorithm on a single thread. We achieve speedup 30.6 on 32 threads and speedup 32 on 64 threads. Hyper-threading is of little help because the memory bandwidth to L3 cache is becoming a limiting factor in our code. In contrast, the Graph 500 and NetworKit generators benefit significantly from hyper-threading since the number of arithmetic operations far dominate the memory accesses.

Figure 3.18 (c) shows the degree distribution of the generated graphs for all three generators. It is clearly visible that the output graphs of all generators have identical degree distributions. All three generators show the plateaus in the degree distribution that are characteristic of R-MAT graphs. This experimentally corroborates our claim that our method generates graphs according to the R-MAT model.

To demonstrate the extreme scale of the graphs that can be generated using our method, we additionally executed it on 256 nodes of SuperMUC-NG, the 13-th fastest supercomputer according to the June 2020 Top 500 list[17] and 4th in the June 2020 Graph 500 BFS list[18]. Each node has two Intel Xeon Platinum 8174 processors with 24 cores (48 hardware threads) and 96 GiB of RAM. We ran both our generator and the Graph 500 generator for 20 minutes each, for $n = 2^{44}$ nodes. Our generator produced $2^{47.995}$ edges, 12.4 times more than the $2^{44.361}$ edges produced by the Graph 500 generator. Thus, generating an R-MAT graph as used in the Graph 500 BFS benchmark for scale 44 ($n = 2^{44}$, $m = 2^{48}$) takes just 20 minutes on 256 nodes of SuperMUC-NG. This graph is four times larger than the largest previously used graph in the official Graph 500 benchmark, at scale 42 ($n = 2^{42}$, $m = 2^{46}$) as of the June 2020 list.

The speed of our generators compares favourably to generators for other models. For example, the sequential Erdős-Renyi generator of Funke et al. [Fun+19] is only around 10 % faster than our sequential generator, but for a much simpler model (the special case $a = b = c = d$). On the other hand, the streaming hyperbolic graph generator sRHG of Funke et al. [Fun+19], a more complicated model, takes around 5 times more time per edge.

3.10 Conclusions and Future Work

We have presented parallel algorithms for a wide spectrum of weighted sampling problems running on a variety of machine models. The algorithms are at the same time efficient in theory and sufficiently simple to be practically useful. Our experiments show that alias table computation can be parallelised efficiently. For random inputs, the variant PSA+ combines the high construction speedup of 2lvl with the query performance of a normal, single-level alias table. For skewed inputs, OS and even more so its variant OS-ND show excellent query performance. Our reservoir sampling algorithm shows good scaling and performance in a distributed evaluation, proving the usefulness of our mini-batch model.

[17] https://www.top500.org/lists/top500/list/2020/06/, archived at https://web.archive.org/web/20200817083528/https://www.top500.org/lists/top500/list/2020/06/ on 17th Aug. 2020

[18] https://graph500.org/?page_id=834, archived at https://web.archive.org/web/20200809164212/https://graph500.org/?page_id=834 on 9th Aug. 2020

As an application of weighted sampling, we give a simple and practical algorithm for generating R-MAT graphs with constant time per edge in an embarrassingly parallel way. This makes this widely used family of graphs even more easily usable for experiments with huge graphs. By applying weighted sampling to precomputed chains consisting of many random decisions, we improve per-core throughput by an order of magnitude over the reference implementation.

Future Work. Future work could consider further implementations, such as the block-wise algorithm of Section 3.3.2.b) or implementations for GPGPUs. The latter is the topic of an ongoing master's thesis by Hans-Peter Lehmann, supervised mainly by the author of this dissertation and showing promising preliminary results. For implementing WRS–N one could perhaps obtain constant factor improvements by studying more accurate estimators for the output size of a sample with replacement.

Our algorithms are formulated for explicitly parallel models of computing. It could also be interesting to adapt them to the more abstract, implicitly parallel models used in databases or big data tools such as MapReduce [DG08], Spark [Zah+10], or Thrill [Bin+16]. For example, using a prefix sum operation, we could build a table based on rejection sampling similar to the one used by Bringmann and Larsen [BL13] in Thrill or Spark. Other systems that are based on sets or relations could emulate arrays by explicitly storing the array index as the first component of a tuple. Batched queries can be supported by join or merge operations with a set of random indices. However, for this to be efficient, the batch needs to have size $\Omega(n)$ or the join operation must keep the larger join partner in-memory allowing sublinear time access to the data. We can also consider a more coarse-grained approach emulating the distributed-memory approach of Section 3.3.3 with $p = \sqrt{n}$ PEs. The table of $\sqrt{n}$ meta-items would be replicated while each distributed object would store a small alias table for $\sqrt{n}$ objects.

Regarding reservoir sampling, a natural avenue for future work would be to study whether our approach could be applied to sampling from a sliding window, ie, sampling only from the (mini-batches containing the) w most recently seen items. Additionally, we find it curious that while using multiple pivots in the selection reduces the selection's average recursion depth by a factor of around 2.5, the running time benefit is much more limited at around 35 % of selection running time (see Section 3.8.4.a)). Preliminary measurements suggest that the reduced number of MPI collective operations does not translate into a corresponding reduction in running time. Careful engineering might be able to improve this. Furthermore, we might be able to improve the work bound of WRS–B by exploiting integer keys for the distributed priority queue.

Acknowledgements

This work was performed on the supercomputer ForHLR funded by the Ministry of Science, Research and the Arts Baden-Württemberg and by the Federal Ministry of Education and Research.

The authors gratefully acknowledge the Gauss Centre for Supercomputing e.V. (`www.gauss-centre.eu`) for funding this project by providing computing time on the GCS Supercomputer SuperMUC-NG at Leibniz Supercomputing Centre (`www.lrz.de`).

Chapter 4

4 Probabilistic Checking of Fundamental Big Data Operations

We propose fast probabilistic algorithms with low (ie, sublinear in the input size) communication volume to check the correctness of operations in Big Data processing frameworks and distributed databases. Our checkers cover many of the commonly used operations, including sum, average, median, and minimum aggregation, as well as sorting, union, merge, and zip. An experimental evaluation of our implementation in Thrill (Bingmann et al. [Bin+16]) confirms the low overhead and high failure detection rate predicted by theoretical analysis.

Motivation. Recently, Big Data processing frameworks like Apache Spark [Zah+10], Apache Flink [Ale+14] and Thrill [Bin+16] have surged in popularity and are widely used to process large amounts of data. Computation and data are distributed over a large number of machines connected by a network, and processing is based on collective operations that transform datasets. Combined with the advances in computing performance and memory capacity brought about by Moore's Law (see Chapter 1), this lets users process enormous amounts of data quickly.[1]

Yet as the size of the data and the number of machines increases, so does the rate of hardware failures and thus the importance of *fault tolerance*, as well as the difficulty of writing correct programs that handle all edge cases. Some frameworks, eg, Apache Spark, can deal with failing machines [Zah+10], but none of the popular solutions can detect *silent data corruption*. These errors can be the result of subtle mistakes in the programming, but also spontaneous bitflips in memory, caused for example by manufacturing defects (*hard errors*) [SPW11] or high-energy particles ('cosmic rays' causing *soft errors*) [ZL79; Har+20]. The latter concern can largely be mitigated by the use of error-correcting code (ECC) memory at the cost of increased hardware expenditure. While ECC memory can *detect* most hard errors, there is a limit to its mitigation capacity, which is exceeded much more frequently than commonly assumed [SPW11]. Moreover, the frequency of occurrence of either kind of errors is expected to continue to increase as hardware miniaturisation progresses [Bor05]. More concerningly, malicious users have been shown to be able to flip bits in memory without accessing them (dubbed 'row hammer' in popular reporting) [Kim+14], posing problems on multi-user HPC systems. Thus, there are numerous causes of silent data corruption in big data systems. We propose probabilistic algorithms to detect such silent failures in the frameworks' operations

[1] When MapReduce experienced its surge in popularity around the start of the so-called 'big data revolution', the primary bottleneck was I/O (input/output) bandwidth, ie, the speed with which data could be read or written to disk. Yet fast solid-state drives (SSDs) have proliferated in the last decade. Today, even a single consumer-grade SSDs can read or write multiple gigabytes per second. A commodity server with multiple of these drives can achieve I/O bandwidths that were previously unimaginable at this price point. Combined with the growing scarcity of the network (see Chapter 1), this means that the bottleneck in Big Data processing frameworks is now often the network. This again emphasises the need for communication efficiency.

Table 4.1: Our main results. Parameters as in Table 4.2: input size n; number of PEs p; failure probability δ; machine word size w; trade-off parameter d; communication startup cost α; communication cost per word β. Operations are defined in Section 4.1.

Operation	Broadcast Result?	Certificate Required?	Checker Running Time $\mathcal{O}(\cdot)$
sum/count aggregation	no	no	$\left(\frac{n}{p} + \beta d\right) \log_d \frac{1}{\delta} + \alpha \log p$
average aggregation	no	dist.	same as above
median aggregation	yes	yes*	same as above
minimum aggregation	yes	yes	$\frac{n}{p} + \alpha \log p$
permutation, sort, union, merge, zip, GroupBy†, join†	no	no	$\left(\frac{n}{pw} + \beta\right) \log \frac{1}{\delta} + \alpha \log p$

* no certificate required if input elements are distinct.
† invasive checker for input redistribution phase.

with little overhead, which we refer to as *checkers*. These checkers can be used to verify the integrity of the main computation while treating the operation as a black box, ie, independent of its implementation. To ensure that their overhead is low and does not impact the system's overall scalability, they need to be highly communication-efficient. Our checkers cover many of the commonly used operations of current-generation data-parallel computing frameworks.

Consider the options available to a programmer who wants to increase users' confidence in the correctness of her program. Formal verification would be ideal, but is very difficult for complex programs and does not address hardware errors. Testing can help the programmer to avoid mistakes, but cannot prove the absence of bugs. Lastly, she can write a small and fast program that verifies the output of the main program: a *checker*. The checker must be very fast to avoid large slowdowns, much faster than the main program. The probabilistic checkers we consider here make a favourable trade-off between confidence and speed.

References. This chapter is based on a conference paper [HS18] published jointly with Peter Sanders. The paper was mainly written by the author of this dissertation, with editing by Peter Sanders. Large parts of this chapter were copied verbatim from the paper and the technical report [HS17] accompanying it, which contains additional detail not present in the conference version. The author of this dissertation is also the author of all implementations and conducted all evaluations.

Results. Table 4.1 lists our main results. For each operation, it states whether the entire result needs to be available at all PEs (or a distributed result suffices), whether the checker requires a certificate—and if yes, whether a distributed certificate suffices—and the checker's running time. Note that the majority of our results also apply to distributed database systems. The

Table 4.2: Summary of notation used in this chapter.

Symbol	Meaning
n	input size
δ	maximum permitted failure rate
d	trade-off (tuning) parameter, $d \in \mathbb{N}$
w	machine word size, see Section 1.2.1
p	number of PEs
α, β	communication characteristics of the machine model, see Section 1.2.3

proximity between data-parallel big data processing frameworks and distributed databases is exemplified by the SQL layer of Apache Spark [Arm+15].

4.1 Preliminaries

Let an operation's input data set consist of n elements, each represented by a fixed number of machine words[2], and let k denote the number of elements in the output of a particular operation, and let w be the machine word size in bits. A summary of the notation used in this chapter is provided in Table 4.2.

In this chapter, we adopt the terminology of Thrill [Bin+16] for operations. We also design our checkers to become part of Thrill, and implemented them within Thrill for our experiments.

Error Model. Checkers have one-sided errors: they are never allowed to reject the result of correct computations. Only in the case of an incorrect result are they allowed to erroneously accept with a small probability, limited by the parameter $\delta > 0$.

Certificates. Some operations become much easier to check if the result of the operation is accompanied by a *certificate* to facilitate checking. However, when an algorithm provides an output along with a certificate for said output, the certificate might also be faulty. We need to take care not to accept incorrect results because the certificate contains the same flaw, ie, we need to verify the correctness of the certificate as well.

Hashing. To simplify the analysis, we assume the availability of (idealised) *random hash functions*, ie, hash functions chosen uniformly at random from the set of all mappings between the input and output value types. Then, for any input, its (fixed) hash value can be treated like a number chosen uniformly at random from the hash functions' image space. We explain separately when weaker guarantees on the randomness of the hash function suffice.

Result Integrity. Ordinarily, the output is provided in distributed form, ie, the PEs hold disjoint parts of the output. However, some checkers require the output of an operation or a certificate to be provided in a full at all PEs. If this is the case, we must additionally ensure

[2] Frequently, adaptation to variable-sized objects is possible. Our focus on fixed-size objects is mainly to simplify notation.

that all PEs received the *same* output or certificate. This can be achieved by hashing the data in question with a random hash function and comparing the hash values of all other PEs. This can be achieved in time $\mathcal{O}(k + \alpha \log p)$ by broadcasting the hash of PE 1, which every PE can compare to its own hash, and aborting if any PE reports a difference.

Operations

We now define the operations which we consider for checking.

Reductions. A *reduce* operation (Figure 4.1 (a)) collects elements by their keys, processing all elements of the same key in an arbitrary order using an associative reduce function f. This function maps two values to a new value of the same type and describes how to combine two elements into one. To reduce the local elements of a PE, we use a hash table h. Process elements one-by-one, denoting the current element by $e = (k, v)$ and set $h[k] = f(v, h[k])$ or $h[k] = v$ if $h[k]$ is empty. We then use a simple collective reduction algorithm (see Section 1.3.3) to obtain the final result at PE 1. The total time taken is $\mathcal{O}(n/p + T_{coll}(k)) = \mathcal{O}(n/p + \beta k + \alpha \log p)$.

Most aggregation operators which we consider are all special cases of this generic reduction framework. In *sum* or *count aggregation*, $f(a, b) = a + b$ is the arithmetic addition operator. *Minimum* or *maximum aggregation* computes the smallest or largest value associated with the keys, $f(a, b) = \min(a, b)$.

The *average aggregation* of Table 4.1 does not directly fall under this reduction framework. To compute per-key averages, the value type is extended by a count, and $f((v_1, c_1), (v_2, c_2) = (v_1 + v_2, c_1 + c_2)$ adds values as well as counts. Finally, in a post-processing step, the value-count-pair (v, c) is mapped to v/c to obtain the average.

GroupBy Aggregations. A more general approach to aggregation is *GroupBy* (Figure 4.1 (b)), where all elements with a certain key are collected at one PE and processed by the group function $g : [Value] \to Value$. This enables the use of more powerful operators such as computing median, but requires more communication. Total running time is $\mathcal{O}(n/p + T_{all\text{-}to\text{-}all}(n/p)) = \mathcal{O}(n/p + \beta n + \alpha p)$.

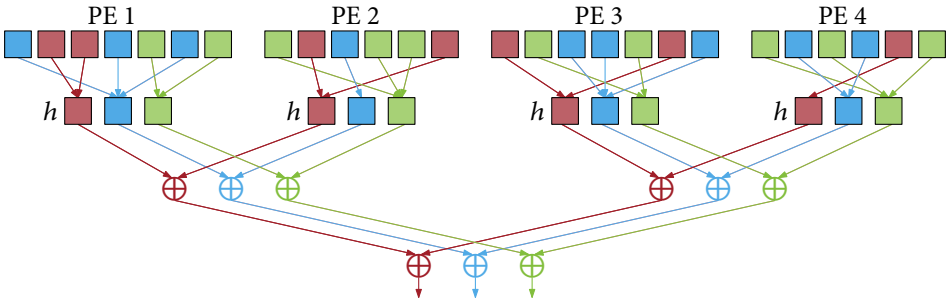
In a *median aggregation*, we compute for each key the value that is larger than one-half of the keys' values and smaller than the other half's. More precisely, for an odd number of elements n_i associated with key i, the median is the element of rank $\lceil n_i/2 \rceil$. For an even number of elements, it is the arithmetic mean of the elements of ranks $n_i/2$ and $n_i/2 + 1$.

Other Operations. A *zip* operation (Figure 4.1 (c)) combines two sequences $S_1 = \langle x_1, \ldots, x_n \rangle$ and $S_2 = \langle y_1, \ldots, y_n \rangle$ of equal length n index-wise, producing as its result a sequence of pairs $S = \langle (x_1, y_1), \ldots, (x_n, y_n) \rangle$.

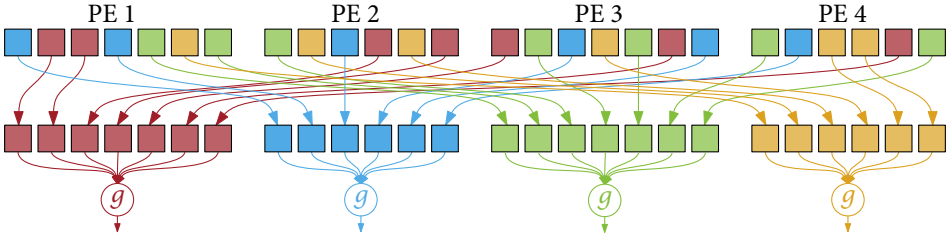
The *union* (Figure 4.1 (d)) operation combines two multisets so that each item's multiplicity in the output is exactly the sum of its multiplicities in the two input sets.

A *merge* (Figure 4.1 (e)) operation combines two sorted sequences into a single sorted sequence whose length is exactly the sum of the input sequences' lengths.

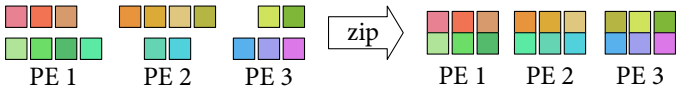
In a *(natural) join*, two data sets A and B indexed by a common key column are combined. For each key, the join outputs all combinations of values from A and B associated with the key.



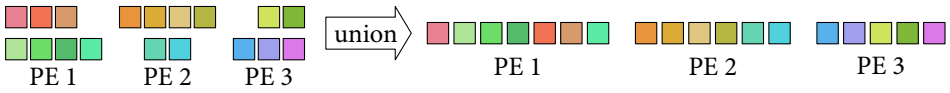
(a) Illustration of a reduction with reduce function $\oplus$, four PEs, and three keys.



(b) Illustration of a GroupBy aggregation with group function g, four PEs, and four keys.



(c) Zipping two sequences.



(d) Union of two sequences: no ordering guarantees.

(e) Merging two sorted sequences into one sorted sequence.

Figure 4.1: Illustration of selected operations considered for checking.

4.2 Related Work

The literature on the wider field of verifying the correctness of a program's output is vast and to attempt to give an overview with any semblance of completeness would be an enormous undertaking. However, we were surprised that distributed—let alone communication-efficient—probabilistic checkers appear to be largely unstudied. We therefore first report on the most directly related works, and then give a short overview of the techniques for increasing confidence in the output of a program that we consider the most relevant.

Yi et al. [Yi+09] present PIRS, an aggregation checker in the context of database management systems. Their algorithm is based on polynomial identities using techniques that are very similar to one of our permutation checkers. This necessitates arithmetic modulo a prime number for each item in the input, which is a rather costly operation. We compare our results to their method in more detail in the appropriate sections.

Nath and Venkatesan [NV13] present a cryptographically secure aggregation checker. However, due to the cryptographic primitives involved being rather expensive to compute, their checker is one to two orders of magnitude slower than PIRS. Other cryptographically secure checkers are based on Merkle hash trees, eg, Ref. [Li+07]. Since cryptographically secure checkers consider a very different security model, we do not explore this avenue in more detail here. Instead, we refer the interested reader to Refs. [Pap+13; Pap+14] for another approach with cryptographic guarantees that is also slower than PIRS.

Tests. In practice, programs are often tested against a set of test cases of known good input/output pairs, ensuring that the program computes the correct result without crashing for all test cases. However, testing can never demonstrate the *absence* of bugs (see, eg, Ref. [Mye04]).

Formal Verification. From a theoretical standpoint, formally verifying the correctness of a program is the ultimate goal. While tremendous progress has been made in this area (eg, Ref. [Ahr+16]), verifying implementations of complex distributed algorithms in popular programming languages is still infeasible.

Certifying Algorithms. Blum and Kannan [BK89; BK95] introduce probabilistic checkers in a sequential setting (although individual algorithms that meet their definitions have existed far longer, eg, Freivalds's [Fre77] matrix multiplication checker). McConnell et al. [McC+11] build upon this to design (sequential) *certifying algorithms*, stressing the importance of simplicity of the checker, and putting increased focus on certificates (also termed *witnesses*) that prove that the output is correct. While formally verifying the main algorithm may be infeasible, doing so for the (far simpler) verifier may be within reach.

Verifiable Computing. Verification of arbitrary computations performed by a single machine or outsourced to a set of untrusted machines is a well-studied problem, dating back at least 30 years and published under names such as 'verifiable computing' [GGP10], 'checking computations' [Bab+91], or 'delegating computations' [GKR15]. All of these are computationally expensive beyond the limits of feasibility, despite recent efforts to make verifiable computing more practical [Par+16; CMT12; Set+12; Bra+13]. A recent survey by Walfish and Blumberg [WB15] concludes that *'[t]he sobering news, of course, is these systems are basically toys'* due to

orders-of-magnitude overhead. In contrast, our work focuses on checking specific operations, allowing us to develop checkers that are fast in practice.

Algorithm-Based Fault Tolerance. An approach that has received significant attention for linear algebra kernels is *Algorithm-Based Fault Tolerance* (ABFT) [HA84; Bos+09]. By encoding the input using checksums, and modifying the algorithms to work on encoded data, ABFT techniques detect and correct any failure on a single processor. This allows error detection with little overhead in many numerical applications, see, eg, Ref. [Zha+20] for a recent result in machine learning. However, the approach fundamentally requires the algorithms to be redesigned to operate on the encoded data.

Helpful Advisor. Cormode et al. [CMT13; CMT12] and Chakrabarti et al. [Cha+14] study outsourced computation in data stream models. A powerful 'helper' (the entity to which the computation has been outsourced) annotates the stream. However, the helper is untrusted and the algorithm must verify the correctness of the annotation while also using the information contained therein to compute the solution. This algorithm is called the 'verifier', and its role is similar to that of a checker in our model. However, the focus of these papers lies on single-pass streaming algorithms, which precludes computing and comparing fingerprints of input and output, and is thus rather different from the setting we study.

Self-Stabilisation. The defining property of self-stabilising systems [Dij74; Sch93] is that they return to correct behaviour within bounded time from being started in any state. Thus, they can recover from arbitrary faults. However, this design goal is intrinsically linked to the design of the algorithm that computes the results in the first place. A self-stabilising algorithm is (usually) designed so that its state is easy to verify. Much like algorithm-based fault tolerance, this requires not just the design of a verification tool, but a redesign of the algorithm that computes the result. Thus, it is a rather invasive approach that may not be desirable.

Proof Labelling Schemes. For graph problems, a successful theoretical approach to locally verifying global properties (also called 'local distributed verification') are proof labelling schemes [KKP10]. Much like our checkers (and a key distinction from self-stabilising algorithms), they are designed to be independent of the result's computation. Instead, for problems that are hard to verify, they use 'advice' that is provided to the verifier. The task of the verifier is then to both convince itself that the advice provided is truthful and to use it to ascertain the result's correctness.[3] However, there are also significant differences to our model of checking. Notably, proof labelling schemes consider graph problems in a *network-centric* model of computation, where each node of the graph is also a node of the network. To establish correctness, for each vertex, the verifier is restricted to working with the vertex's state and the advice on vertices that are at most a constant distance away from the vertex (eg, its neighbours). Similar ideas that do not use external advice, relying only on the solution, were considered earlier by Naor and Stockmeyer [NS95], also in the context of graph problems. Korman and Kutten [KK06] give an introduction and provide an overview of some noteworthy results in both settings as well as self-stabilisation (see above).

[3] Under different names, such advice pops up in numerous places throughout computer science. For example, in a common characterisation of the complexity class NP, a *witness* allows an algorithm from P to verify that the decision problem can be solved. The certificates used by some of our checkers are another form of advice.

Fault Tolerance. Note that while some existing systems, eg, Apache Spark [Zah+10], implement some fault tolerance measures such as detecting and handling the failure of individual nodes, there are large classes of failures that no existing big data processing system appears to detect or mitigate. These include data corruption caused by *soft errors*, ie, incorrectly handled edge cases in the programming, or bitflips in CPU or main memory caused by cosmic rays, leak voltage, as well as *hard errors*, ie, device defects causing repeated failure at the same memory locations. A common solution to memory errors is the use of ECC RAM [BCH13], which can correct most soft errors and detect (but often not correct) many hard errors [HSS12]. This is orthogonal to our checkers: a checker can never detect errors introduced into the input data before the algorithm is invoked, so the use of ECC RAM remains advisable even when computations can be checked. An alternative approach is to use redundant computations, eg, Ref. [Fia+12], which comes at a rather high cost, though it may be an option if failure rates are high enough for recovery to dominate overall running time.

4.3 Checking Aggregations

Reductions are perhaps the most important operation in the kind of Big Data systems we consider, and the first instances of these systems even carried them in their name: MapReduce. The checker we describe works not only for sum aggregation but also other operations on integers that fulfil certain properties. We require the reduce operator $\oplus$ to be associative, commutative, and satisfy $x \oplus y \neq x$ for all $y \neq 0$, ie, every element except the neutral element changes the result. Examples include count aggregation, which conceptually equals sum aggregation where the value of every element is mapped to 1, and exclusive or (*xor*). Without loss of generality, we therefore only discuss sum aggregation in this section. Several other common choices of reduce functions violating the above requirements and aggregations requiring the additional power of GroupBy are discussed in Section 4.5.

In sum aggregation, we are given a distributed set of (*key*, *value*)-pairs. Let K be the (unknown) set of keys in the input and $k := |K|$ its equally unknown size. The output of sum aggregation then consists of a single value for each key, which is the sum of all values associated with it in the input. In SQL, this operation would be expressed as

```
SELECT key, SUM(value) FROM table GROUP BY key.
```

To check the result of such an aggregation, we apply a naïve sum reduction algorithm to a condensed version of the input. Due to the small size of the data to which it is applied, we sometimes refer to this as the *minireduction*. The condensed version of the input which we apply it to is conceptually similar to a counting Bloom filter [Fan+00] or a count-min sketch [CM05]. Just as is done there, we use a random hash function to map items to a new, smaller key space and then sum up the values of all items with the same hash key (see Figure 4.2 for an example of our checker). Conceptually, a count-min sketch is 'just' a counting Bloom filter that is too small to reliably answer membership queries. Similarly, our condensed version of the input is the same data structure, but even smaller, so that it would not be able to reliably answer the queries that a count-min sketch is commonly used for (eg, quantiles or frequent items). We then merge the local summaries obtained this way to produce a 'fingerprint' of the input, and

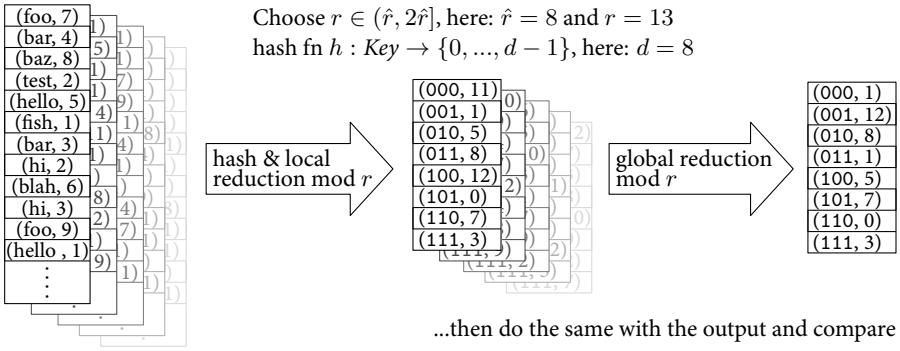


Figure 4.2: *Word Count* example of the sum aggregation checker. On the left, the PEs' inputs, shown as columns of (word, count) pairs. Inputs are first hashed and reduced locally modulo r (centre) followed by a global reduction over the local tables. The resulting fingerprint is shown on the right and fits into $d \cdot \lceil \log(2\hat{r}) \rceil = 8 \cdot 4 = 32$ bits. This configuration of the checker fails with probability $\delta \leq 1/\hat{r} + 1/d = 1/4$.

compare it to the fingerprint of the output. We will show that if the fingerprints match, the output was likely computed correctly.

Theorem 4.1 (Aggregation Checker)

Let $\oplus$ be an associative and commutative reduce function on integers satisfying $x \oplus y \neq x$ for all elements x, y with $y \neq 0$. Then $\oplus$-aggregation of n elements with failure probability at most δ can be checked in time $T_{\text{check-sum}}(n, p, \delta) := \mathcal{O}\left(\left(\frac{n}{p} + \beta d\right) \log_d \frac{1}{\delta} + \alpha \log p\right)$, where $d \in \mathbb{N}$ is a tuning parameter.

Observe that only the local work depends on the input size and that the running time is independent of the number of keys that appear in the input. Let d from the statement of Theorem 4.1 be the size of the condensed key space, with $2 \leq d \ll k$. We then use a random hash function $h : K \to [1 \,..\, d]$ to map the original keys to the reduced key space. Let $\hat{r} \geq d$ be a modulus parameter, with r chosen uniformly at random from the range $[\hat{r} + 1 \,..\, 2\hat{r}]$. Apply a naïve sum reduction modulo r to the thus-remapped input and—separately—the output of the aggregation algorithm. If both produce the same result, the operation was likely conducted correctly. Pseudocode is shown in Algorithm 4.1.

Lemma 4.2 (Aggregation Checker Failure Probability)

A single iteration of the above sum aggregation checker fails with probability at most $\frac{1}{\hat{r}} + \frac{1}{d}$.

Proof. If the aggregation was performed correctly, then the per-bucket results in the reduced keyspace are the same for input and output, and thus the checker always accepts a correct result. This follows from the properties we require the reduction operator to have, which carry over to the ring in which we perform the mini-reduction. Thus assume from now on that the output of the aggregation operation is *incorrect*, ie, the checker *should* fail.

Algorithm 4.1 : A single iteration of the sum aggregation checker.

Input : v the local input to the sum aggregation operation, an array of size n_i, o the local part of the asserted result of the sum aggregation, an array of size k_i, $\delta \in \mathbb{R}$ the maximum allowed failure rate

Output : whether o is likely the correct result of the sum aggregation

Function checkSumAgg(v, o, δ)
- $(d, \hat{r}) : \mathbb{N} \times \mathbb{N}$:= numerically determined parameters (see Table 4.3)
- $r : \mathbb{N}$:= random number from $[\hat{r} + 1 \,..\, 2\hat{r}]$ — *modulus parameter*
- $h : K \to [1 \,..\, d]$:= random hash function — *maps keys to buckets*
- w_v := cRed(v, d, r, h) — *apply condensed reduction to input*
- w_o := cRed(o, d, r, h) — *...and asserted result of sum aggregation*
- **return** $w_v = w_o$ — *significant only at PE 1*

Input : *arr* an array of input items, $d \in \mathbb{N}$ the result size, $r \in \mathbb{N}$ the modulus, $h : K \to [1 \,..\, d]$ a hash function

Output : a d-dimensional sum modulo r of the inputs' hash values

Function cRed(arr, d, r, h)
- t : Value$[d]$:= $\langle 0, \ldots, 0 \rangle$
- **foreach** $(k, v) \in arr$ **do**
  - $t[h[k]] := (v + t[h[k]]) \bmod r$ — *local reduction*
- Reduce$(t, +_r, 1)$ — *reduce to PE 1 with addition modulo r*
- **return** t — *significant only at PE 1*

For $i \in K$, let n_i be the (correct, unknown) $\oplus$-aggregate of all values with key i, and n_i' be the asserted result of the corrupted computation. Then, because the result is incorrect, there exists at least one i with $n_i \neq n_i'$. Let $I := \{i \in K \mid n_i \neq n_i'\}$ be the keys whose results were computed incorrectly by the operation.

Let $h : K \to [1 \,..\, d]$ be the random hash function used to map keys to the condensed keyspace. We use this hash function to map elements to the d buckets by their keys, and reduce the values in associated counters modulo r, where r is chosen uniformly at random from $[\hat{r} + 1 \,..\, 2\hat{r}]$. Effectively, we operate in the residue class ring $\mathbb{Z}/r\mathbb{Z}$. Thus the checker fails if

$$\forall_{j \in [1 \,..\, d]} : \bigoplus_{\substack{i \in I \\ h(i) = j}} n_i = \bigoplus_{\substack{i \in I \\ h(i) = j}} n_i' \mod r \ .$$

We prove the claimed failure probability by first considering the failure probability introduced by the modulus, and then analysing a version of the checker without a modulus. For the first part, we bound the probability of such an error by $1/\hat{r}$. In the second part, we injectively map each hash function h for which the checker fails to $d - 1$ distinct hash functions for which it does not fail, yielding the $1/d$ term of the failure probability.

(1) If there is an $i \in I$ with $n_i = n_i' \mod r$, the checker cannot detect the error in this key. Because the modulus is chosen randomly, this occurs with probability $r^{-1} \leq \hat{r}^{-1}$ by definition

of r for a single key. However, for the checker to fail, $n_i = n_i' \mod r$ must hold for all $i \in I$. Thus, the probability of failure declines exponentially with $|I|$, and $|I| = 1$ is the hardest case. Therefore, $\hat{r}^{-1}$ bounds the total additional failure probability introduced by the modulus.

(2) Define $F := \{h : K \to [1\,..\,d] \mid \text{checker fails for } h\}$ as the set of hash functions for which the checker fails and let $\check{\imath} := \min I$ be the first key whose aggregate value is incorrect. For each $h \in F$, we define the $d-1$ hash functions $\overline{h_j}$ with

$$\forall_{j \in [1\,..\,d] \setminus \{h(\check{\imath})\},\, i \in K} : \overline{h_j}(i) = \begin{cases} h(i) & i \neq \check{\imath}, \\ j & \textit{else} \end{cases}$$

and let $\overline{F} := \left\{\overline{h_j} \mid h \in F, j \in [1\,..\,d] \setminus \{h(\check{\imath})\}\right\}$. Clearly, the checker does not fail for any $\overline{h_j} \in \overline{F}$, as exactly one element with different values is being remapped in a hash function for which it does fail (the case of $n_{\check{\imath}} = n_{\check{\imath}}' \mod r$ is treated in **(1)**). By the assumption that $x \oplus y \neq x$ for $y \neq 0$, the result will differ in exactly two buckets, and the checker will notice.

We also need to show that the mapping is injective, ie, that the new hash functions are unique and thus $|\overline{F}| = (d-1)\,|F|$. Clearly, for a given h all of its $\overline{h_j}$ are different, so assume that there exists an $h' \neq h$ and j, j' so that $\overline{h_j} = \overline{h'_{j'}} \in \overline{F}$. Then, by definition, $j = \overline{h_j}(\check{\imath}) = \overline{h'_{j'}}(\check{\imath}) = j'$ and thus $j = j'$. For $\overline{h_j} = \overline{h'_j}$, we furthermore need

$$\forall_{i \in K} : \overline{h_j}(i) = \overline{h'_j}(i) \overset{def.}{\iff} \forall_{i \in K \setminus \{\check{\imath}\}} : h(i) = h'(i)$$

Thus $h = h'$ if and only if $h(\check{\imath}) = h'(\check{\imath})$. But this must hold, for otherwise we would have $h' = \overline{h_{h'(\check{\imath})}} \in \overline{F}$ by definition of the $\overline{h_j}$, which contradicts the assumption that $h' \in F$. Therefore, such an h' cannot exist and $|\overline{F}| = (d-1) \cdot |F|$. □

Observe that unlike the failure probability bound in **(1)**, the bound in **(2)** is tight: if the only difference between the two inputs is the key of a single item, then the probability that the hash values of the new and old key are the same (and the modification thus goes unnoticed) is $1/d$. This follows from the uniformity of random hash functions.

Lemma 4.3 (Aggregation Checker Running Time)

Checking $\oplus$-reduction with $\oplus$ as in Theorem 4.1, n input pairs, and k keys, using d buckets, with moduli in $[\hat{r}+1\,..\,2\hat{r}]$ and probability of failure at most $\delta > 0$, is possible in time

$$\mathcal{O}\left(\left(\frac{n}{p} + \beta\frac{d\log(\hat{r})}{w}\right)\log_{\left(\frac{1}{\hat{r}}+\frac{1}{d}\right)^{-1}}\delta^{-1} + \alpha\log p\right).$$

Proof. From the above description we can see that a single iteration of the checker requires time $\mathcal{O}(n/p)$ to hash and locally reduce the input. A single iteration's table that can be expressed in $d\lceil\log(2\hat{r})\rceil$ bits, ie, $\lceil d\log(2\hat{r})/w\rceil$ machine words. Thus, the reduction takes time $T_{coll}(\lceil d\log(2\hat{r})/w\rceil) \in \mathcal{O}(\beta d\log(\hat{r})/w + \alpha\log p)$. Repeating the procedure $\lceil\log_{\left(\frac{1}{\hat{r}}+\frac{1}{d}\right)^{-1}}\delta^{-1}\rceil$ times increases the probability of detecting an incorrect result to at least $1-\delta$ by accepting only

if all repetitions of the procedure declare the result to be likely correct. To keep the number of messages low, we can execute all instances of the checker simultaneously (this also means that we only have to read the input once), and perform their reductions at the same time. □

We can instantiate this checker in different ways to obtain the characteristics we wish. First, we use this to show Theorem 4.1:

Proof (Theorem 4.1). Setting $d \leq \hat{r} \leq 2^{w-1}$ and $(\frac{1}{\hat{r}} + \frac{1}{d})^{-1} < d$ in Lemma 4.3 yields $\log(2\hat{r}) \leq w$ and running time $\mathcal{O}\big((n/p + \beta d)\log_d \delta^{-1} + \alpha \log p\big)$, which can be simplified to the claimed bound. □

We can also minimise bottleneck communication volume and find that minimum at $d = 2$ buckets, $\hat{r} = 8$ for a modulus range of $[9\,..\,16]$, and thus a minireduction result size of 8 bits with $\log_{1.6} \delta^{-1}$ repetitions. However, this high number of repetitions causes a lot of local work. Further, the practical usefulness of sending single bytes across the network is questionable. In effect, real-world interconnects have an effective minimum message size b, such that sending fewer than b bits is not measurably faster than sending a message of size b bits. Thus, our goal should be to minimise the number of iterations—$\lceil \log_{\frac{1}{\hat{r}}+\frac{1}{d}} \delta \rceil$—under the constraint that the result size shall be close to b bits: $d\lceil \log(2\hat{r}) \rceil \lceil \log_{\frac{1}{\hat{r}}+\frac{1}{d}} \delta \rceil \leq b$. This relation can be used to (numerically) compute optimal choices of $\hat{r}$ and d for a given b. Table 4.3 shows such values for some interesting choices of b and δ. However, in practice, keeping local work low might

Table 4.3: Select numerically determined optimal values for the sum aggregation checker's bucket count d and modulus parameter $\hat{r}$ given a message size of b bits.

b	δ	d	$\hat{r}$	#iter	achieved δ
1024	10^{-4}	37	2^8	3	$3.0 \cdot 10^{-5}$
1024	10^{-6}	25	2^7	5	$2.5 \cdot 10^{-7}$
1024	10^{-8}	18	2^7	7	$4.1 \cdot 10^{-9}$
1024	10^{-10}	14	2^6	10	$2.5 \cdot 10^{-11}$
1024	10^{-20}	6	2^4	32	$3.3 \cdot 10^{-21}$
4096	10^{-6}	124	2^{10}	3	$7.4 \cdot 10^{-7}$
4096	10^{-10}	68	2^9	6	$2.1 \cdot 10^{-11}$
4096	10^{-20}	32	2^8	14	$4.4 \cdot 10^{-21}$
16 384	10^{-7}	420	2^{12}	3	$1.8 \cdot 10^{-8}$
16 384	10^{-10}	273	2^{11}	5	$1.2 \cdot 10^{-12}$
16 384	10^{-20}	148	2^{10}	10	$7.6 \cdot 10^{-22}$
16 384	10^{-30}	93	2^{10}	16	$1.3 \cdot 10^{-31}$
65 536	10^{-10}	1170	2^{13}	4	$9.1 \cdot 10^{-13}$
65 536	10^{-20}	630	2^{12}	8	$1.3 \cdot 10^{-22}$
65 536	10^{-30}	420	2^{12}	12	$1.1 \cdot 10^{-31}$
65 536	10^{-40}	321	2^{11}	17	$2.9 \cdot 10^{-42}$

be more important than these solutions to minimise δ admit, and one might prefer to trade a reduced number of iterations for a larger value of d and perhaps choose $\hat{r} = 2^{31}$.

A Note on Related Work. Yi et al. [Yi+09] claim that an approach based on the count-min sketch cannot work. However, their argument is based on using the sketch for its original purpose, estimating the count of each key. This would indeed, as the authors claim, necessitate an excessively large sketch without providing good guarantees. However, we do not query the sketch at all, but treat the entire sketch as a fingerprint of the input and compare these fingerprints. Therefore, their considerations do not apply to us.

Optimisations. Multiple instances of this algorithm can be executed concurrently by using a hash function that computes $c \cdot \lceil \log d \rceil$ bits. Its value can then be interpreted as c concatenated hash values for separate instances, enabling bit-parallel implementation. It is also possible to use Single Instruction Multiple Data (SIMD) techniques to further reduce local work. Refer to Section 4.6 for specific implementation details.

4.4 Checking Permutations

Many approaches for permutation checking exist in the sequential case, and often directly imply communication-efficient equivalents. Perhaps the most elegant approach was first described by Wegman and Carter [WC81], albeit in a less general manner than we prove here. The idea is to use a random hash function and compare the sum of hash values in the input and output sequences. Checking permutations is perhaps most useful for verifying the output of sorting algorithms. Once we have established that a sequence is a permutation of another, verifying sortedness of the output sequence only requires that each PE receive the smallest element of its successor PE and compare its local maximum to it.

Lemma 4.4 (Hash-Based Permutation Checker)

Let $E = \langle e_1, \dots, e_n \rangle$ and $O = \langle o_1, \dots, o_n \rangle$ be two sequences of n elements from a universe U. For a random hash function $h : U \to [0\,..\,H-1]$, define $\lambda := \sum_{i=1}^{n} h(e_i) - h(o_i)$. Then $\lambda = 0$ if E is a permutation of O, and $\mathbf{P}[\lambda = 0] \leq H^{-1}$ otherwise. If the items of E and O are pairwise different within the respective sequence, ie, each item occurs only once, then the computation can also be performed in $\mathbb{Z}/H\mathbb{Z}$ (ie, mod H*) without affecting the result.*

Proof. Define $h(X) := \sum_{x \in X} h(x)$ for any set or sequence X. If E and O are permutations of each other, then $h(E) = h(O)$ due to commutativity of addition, and thus $\lambda = 0$ by associativity.

Otherwise, let e be an element that—without loss of generality—occurs more frequently in E than in O, say k times in E and $k' < k$ in O (k' can, of course, be 0). Then we can express $h(E)$ and $h(O)$ as $h(E) = h(E \setminus e) + k \cdot h(e)$ and $h(O) = h(O \setminus e) + k' \cdot h(e)$. Assuming $h(E) = h(O)$ (so that $\lambda = 0$) and solving for $h(e)$, we obtain $h(e) = (h(O \setminus e) - h(E \setminus e))/(k - k')$. But the right side of this equation is independent of the value of $h(e)$, which is uniformly distributed over $[0\,..\,H-1]$. Thus, the event $h(E) = h(O)$—and therefore $\lambda = 0$—occurs with probability at most $1/H$.

For pairwise unique items, $k = 1$ and $k' = 0$, and thus the calculation works in $\mathbb{Z}/H\mathbb{Z}$ without affecting the probability. □

The above proof is, in effect, a solution to Exercise 5.8 of Ref. [San+19].

Every PE can compute the sum for its $\mathcal{O}(n/p)$ local elements independently. A sum all-reduction over these values then yields s. The algorithm can thus be executed in time $\mathcal{O}(n/p + \beta \log(H)/w + \alpha \log p)$ for unique items, and $\mathcal{O}(n/p + \beta \log(nH)/w + \alpha \log p)$ otherwise. Similar to Section 4.3, the success probability can be boosted arbitrarily by repeating the algorithm with different hash functions. By executing instances simultaneously, this can be done without affecting the number of messages.

However, this algorithm requires confidence in the randomness of the hash function h. Blum and Kannan [BK95, Section 6.2] give an elegant construction for a family of sufficiently random hash functions for the case $\delta = \frac{1}{2}$. If we do not have access to a sufficiently trusted hash function, we can use a different approach based on constructing a polynomial from E and O, which is originally due to Lipton [Lip89] and essentially a solution to Exercise 5.7 of Ref. [San+19].

Lemma 4.5 (Polynomial-Based Permutation Checker)
For two sequences $E = \langle e_1, \ldots, e_n \rangle$ and $O = \langle o_1, \ldots, o_n \rangle$ from a universe $[0\,..\,U-1]$ and any $\delta > 0$, choose a prime $r > \max(n/\delta, U-1)$ and define the polynomial

$$q(z) := \prod_{i=1}^{n} (z - e_i) - \prod_{i=1}^{n} (z - o_i) \mod r.$$

Then for random $z \in [0\,..\,r-1]$, $q(z) = 0$ if E is a permutation of O, and $\mathbf{P}[q(z) = 0] < \delta$ otherwise.

Proof. If E is a permutation of O, then both products have the same factors and thus $q(z) = 0$ for all z by commutativity of multiplication. Otherwise, because r is larger than any e_i or o_i (this is important to ensure that no pair i, j with $e_i \equiv o_j \mod r$ exists) is prime, q is a non-zero polynomial of degree n in the field $\mathbb{F}_r$. Such a polynomial has at most n roots.[4] Thus the probability of $q(z) = 0$ is at most $\frac{n}{r} < \frac{n}{n/\delta} = \delta$. □

Instead of doing $2n$ expensive multiplication-modulo-prime operations, one could also consider using carry-less multiplication in a Galois Field $\mathrm{GF}(2^\ell)$ with an irreducible polynomial. Multiplication in Galois Fields can be implemented very efficiently, eg, using Intel SIMD instructions [PGM13].

Theorem 4.6 (Permutation Checker)
It is possible to check whether a sequence of n elements is a permutation of another such sequence with probability at least $1 - \delta$ in time $T_{\text{check-perm}}(n, p, \delta) \subseteq \mathcal{O}\left(\left(\frac{n}{pw} + \beta\right) \log \frac{1}{\delta} + \alpha \log p\right)$.

[4] This can easily be seen by induction. It is easy to verify for $n \leq 1$. Now let q be a polynomial of degree $n \geq 2$, and a be a root of q (if none exists, we are finished). Then $q = q' \cdot (X - a)$ for some polynomial q' of degree $n-1$, and by the induction hypothesis, q' has at most $n-1$ roots. For some $b \in \mathbb{F}_r$, $q(b) = q'(b) \cdot (b - a)$ is zero if and only if $a = b$ or $q'(b) = 0$. Thus q has at most n roots.

Proof. We can boost the success probability of the algorithm of Lemma 4.4 arbitrarily by executing several independent instances and accepting only if all instances do. Batching communication keeps latency to $\log p$. Choose $H = 2^w$ in Lemma 4.4 to obtain failure probability at most 2^{-w} with cost $\mathcal{O}(n/p + \beta \log n + \alpha \log p)$ for a single round. Executing $1/w \cdot \log(1/\delta)$ instances then yields failure probability at most δ in time $\mathcal{O}\big((n/p + \beta \log n)\frac{1}{w} \log \frac{1}{\delta} + \alpha \log p\big)$. The claimed running time then follows because the machine's word size w is assumed to be able to hold an input item's index, which has $\lceil \log n \rceil$ bits.

For unique items, the $\log n$ factor in the communication volume is removed because the range of values in Lemma 4.4 is $[0\,..\,2^{w-1}]$ and thus a single machine word suffices. □

If we want to use the polynomial checker of Lemma 4.5, choose $\delta = 2^{-w+1}n$, which is less than one as we can assume $n < 2^{w-1}$. Then, as we can assume $U \leq 2^w$, r can always be chosen in $[2^{w-1}\,..\,2^w]$ by Bertrand's postulate, and a single machine word can hold the necessary values. This gives cost $\mathcal{O}(n/p + \beta + \alpha \log p)$ per round and $\log(1/\delta)/(w - 1 - \log n)$ rounds, which is worse than the hash-based checker.

Theorem 4.7 (Sort Checker)
Checking whether a sequence of n elements is a sorted version of another such sequence with probability $\geq 1 - \delta$ is possible in time $T_{\text{check-sort}}(n, p, \delta) := \mathcal{O}\big(T_{\text{check-perm}}(n, p, \delta)\big)$.

Proof. After verifying the permutation property using Theorem 4.6, it remains to be checked whether the elements are sorted. First, verify that the local data is sorted in time $\mathcal{O}(n/p)$. Then, transmit the locally smallest element to the preceding PE, and receive the smallest element of the next PE. Compare this to the locally largest element. Lastly, verify that no PE rejected using a sum all-reduction on an indicator flag. In total, this requires time $T_{\text{check-perm}}(n, p, \delta) + \mathcal{O}(n/p) + \mathcal{O}(\alpha) + T_{\text{coll}}(\mathcal{O}(1)) \subseteq \mathcal{O}\big(T_{\text{check-perm}}(n, p, \delta)\big)$. □

4.5 Further Checkers

There are many operations for which we can construct checkers from the sum aggregation and permutation checker. We discuss several in the following subsections. Afterwards, we turn to some *invasive* checkers, ie, checkers that do not treat the operation as a black box, and instead check only part of the operation. While less desirable than true checkers, these checkers allows us to broaden the range of covered operations.

4.5.1 Average Aggregation

Computing the per-key averages is impossible to express in a scalar reduction, but becomes easy when replacing $(key, value)$-pairs with $(key, value, count)$-triples and using the reduce function $\oplus$ with $(k_1, v_1, c_1) \oplus (k_1, v_2, c_2) := (k_1, v_1 + v_2, c_1 + c_2)$. The computation is then followed up by an output function h with $h(k_1, v_1, c_1) := (k_1, v_1/c_1)$. This avoids the use of the much more communication-expensive GroupBy function.

Checking average aggregation is easy when these per-key element counts are available in a certificate, for then we can reconstruct the result of a sum aggregation by undoing the final

division—termed h above—by multiplying the average value with the count for every key. As described above, this certificate naturally arises during computation anyway and thus does not impose overhead on the average computation. Now we can leverage the sum checker, applying it both to the input sequence and the reconstructed sums. As the product is computed component-wise, both the asserted averages and the certificate can be supplied in distributed form, as long as both values are available at the same PE for any key.

To prevent accidental mismatches when both averages and counts are scaled in a way that yields the same reconstructed sums—eg, double the averages and halve the counts—, we also need to check the correctness of the counts. Therefore, we also need to apply the *count aggregation checker* to the input and the counts in the certificate. The same can also be achieved in a single step by applying the $(key, value, count)$-triple aggregation trick using the reduce function $\oplus$ described above to the checker.

Corollary 4.8 (Average Aggregation Checker)

For a given input sequence of n key-value pairs $\langle e_1, \ldots, e_n \rangle$ with $e_i = (k_i, v_i)$, keys $k_i \in K$, and $k := |K|$, it is possible to check whether a set of k asserted per-key averages $\langle a_1, \ldots, a_k \rangle$ is correct if the number of values associated with each key is available as a certificate $\langle c_1, \ldots, c_k \rangle$, by supplying $\langle (e_1, 1), \ldots, (e_n, 1) \rangle$ as input and $\langle (a_1 \cdot c_1, c_1), \ldots, (a_k \cdot c_k, c_k) \rangle$ as output to the sum aggregation checker of Section 4.3, achieving the time complexity and success probability of Theorem 4.1.

4.5.2 Minimum and Maximum Aggregation

Now consider computing the minimum or maximum value per key (without loss of generality, we shall henceforth only consider minima). This is a surprisingly difficult operation to check. Clearly, the sum aggregation checker of Section 4.3 is not directly applicable, as the min function does not satisfy the requirement $\min(a, b) \neq a$ if $b \geq a$. To check min-aggregation, we need to verify for each key that (a) no elements smaller than the purported minimum exist, as well as determine whether (b) the minimum value does indeed appear in the input sequence. Both subproblems seem to require the asserted result to be known at all PEs. Let $S = \langle x_1, \ldots, x_n \rangle$ be the input sequence, and $M = \langle m_1, \ldots, m_k \rangle$ the asserted output. Given M, property (a) is easy to verify by iterating the locally present part of S and verifying that no element exists with a value smaller than the entry of its key in M.

However, property (b) is surprisingly hard to check using $o(k)$ bits of communication.[5] A certificate in the form of the locations of the minima, available in full at every PE, remedies this. Each PE then needs to verify its set of local asserted minima. The certificate is required to be available in full at every PE so that we can ensure that all keys are covered—otherwise, a faulty algorithm might simply 'forget' a key, and the checker would not be able to notice.

Theorem 4.9 (Minimum/Maximum Aggregation Checker)

Checking minimum (maximum) aggregation is possible in time $\mathcal{O}(n/p + \alpha \log p)$ provided that the asserted output and a certificate specifying which PE holds the minimum (maximum) element for any key are available at all PEs.

[5] It is easy to verify in time $\mathcal{O}(n/p + \beta k/w + \alpha \log p)$ using a bitwise or reduction—see Section 1.3.3—on a bitvector of size k specifying which keys' minima are present locally, and testing whether each bit is set in the result.

Proof. Verifying property (a) takes a linear scan over the input with a table lookup for each key, and can be done in time $\mathcal{O}(n/p)$. To check property (b), each PE needs to ensure that the minima (maxima) which are claimed to occur at this PE are actually present. This is another local operation which can be implemented in linear time. Lastly, testing whether all PEs accepted their part of the output takes time $\mathcal{O}(\log p)$ with a simple all-reduction. To ensure that the certificate is the same at every PE, we can use a (cryptographic) hash function and check that each PE's certificate has the same hash value. This is another all-reduction on a constant number of machine words. □

Note that this checker is not probabilistic and thus guaranteed to notice any errors in the result. We discuss possible improvements and lower bounds in the paragraph on future work at the end of this chapter.

4.5.3 Median Aggregation

We use the common definition of the median of an even number of elements as the mean of the two middle elements. Thus, there may not exist an element that is equal to the median, but—assuming unique values—the number of elements *smaller* than the median is always equal to the number of *larger* elements.

If the asserted medians are available at every PE, checking a median aggregation operation on unique values can be reduced to the sum aggregation problem by verifying the above property.[6] Simply map elements smaller than their key's median to -1, and larger elements to $+1$. Then, the sum over all of these values must be 0 for every key. This property can be checked probabilistically using the sum aggregation checker of Theorem 4.1. Pseudocode of our algorithm is given in Algorithm 4.2.

[6] Requiring that each value occur no more than once for each key is without loss of generality because it can be enforced by an appropriate tie-breaking scheme, which needs to be known to both the median aggregation operation and the checker.

Algorithm 4.2 : Median checker, assuming unique values. The function checkSumAgg is defined in Algorithm 4.1. For simplicity of the presentation, we use associative arrays indexed by key for o and s.

Input : v, the local part of the input to the median aggregation, an array of size n_i, o, the asserted result, an array of size k, δ, the maximum allowed failure rate
Output : Whether o is likely the correct result of the median aggregation

```
1 Function checkMedian(v, o, δ)
2     s : Map of Key → ℤ using default value 0 for unknown keys, size k
3     foreach (key, value) ∈ v do                    — combine mapping and local reduction
4         if val < o[key] then s[key] = s[key] − 1
5         if val > o[key] then s[key] = s[key] + 1
6     return checkSumAgg(s.as_array(), ⟨0, ..., 0⟩, δ)          — see Algorithm 4.1
```

Theorem 4.10 (Median Aggregation Checker)
Median aggregation on a sequence of n elements can be checked with failure probability at most $\delta > 0$ in time $\mathcal{O}(T_{\text{check-sum}}(n, p, \delta))$ provided the asserted result is available at every PE. For non-unique values, tie breaking information on the median values is required as a certificate.

Proof. By definition, an element is the median of a set of unique elements if and only if the number of elements smaller than it is exactly equal to the number of elements that is larger. This is verified using the sum aggregation checker by mapping smaller values to -1 and larger values to $+1$. Ties are broken using an appropriate tie breaking scheme and the certificate. The claim then follows from Theorem 4.1. □

4.5.4 Zip

Zipping combines two sequences $S_1 = \langle x_1, \ldots, x_n \rangle$ and $S_2 = \langle y_1, \ldots, y_n \rangle$ of equal length n index-wise, producing as its result a sequence of pairs $S = \langle (x_1, y_1), \ldots, (x_n, y_n) \rangle$. However, since S_1 and S_2 need not have the same data distribution over the PEs, this is nontrivial. Thus, the elements of (at least) one sequence need to be moved in the general case. Checking a Zip operation hence requires verifying that the order of the elements is unchanged in both sequences. For this, we require a hash function that can be evaluated in parallel on distributed data, independent of how the input is split over the PEs. One example would be the inner product of the input and a sequence of n random values, $R = \langle r_1, \ldots, r_n \rangle$, where $r_i = h'(i)$ for some high-quality hash function h' [San+19]. This way, R can be computed on the fly and without communication. The PEs only need to compute the global indices of their local input, which can be done with a prefix sum in time $\mathcal{O}(\alpha \log p)$.

Theorem 4.11 (Zip Checker)
Checking $\text{Zip}(S_1, S_2)$ *with* $|S_1| = |S_2| = n$ *with a false-positive probability of at most* $\delta > 0$ *can be achieved in time* $\mathcal{O}\big((n/p + \beta)\frac{1}{w} \log \frac{1}{\delta} + \alpha \log p\big)$.

Proof. For a single iteration of the checker, apply the hash function to S_1 and the first part of the elements of S. This hash value can be computed in time $\mathcal{O}(n/p + \beta \log(H)/w + \alpha \log p)$ for the families of hash functions discussed above, where the output of the hash function is in $[0\,..\,H-1]$. The same is applied to S_2 and the second part of the elements of S. Accept if both pairs of hashes match. Again, the success probability can be boosted arbitrarily to $1 - \delta$ by executing $\log_H \delta^{-1}$ instances of the checker in parallel, and the claimed running time follows for $H = 2^w$ and $\log(\delta^{-1})/w$ concurrent instances. □

4.5.5 Other Operations

We now briefly describe several further operations to which the permutation and sort checker can be adapted. For the latter two, we present *invasive* checkers for the element redistribution phase. The rest of the operation needs to be checked with an appropriate local checker.

4.5.5.a) Union

Verifying whether a multiset S is the union of two other multisets S_1 and S_2 of size n_1 and n_2, respectively, is equivalent to checking whether S is a permutation of the concatenation of S_1 and S_2. We can therefore adapt the permutation checker of Section 4.4 to iterate over *two* input sets.

Corollary 4.12 (Union Checker)

The $\mathrm{Union}(\mathrm{S}_1, \mathrm{S}_2)$ *operation can be checked in the time it takes to check permutations of size* $|S_1| + |S_2|$ *to the same error detection probability* $1 - \delta$*, ie, time* $\mathcal{O}(T_{\textit{check-perm}}(|S_1| + |S_2|, p, \delta))$*, with* $T_{\textit{check-perm}}(n, p, \delta)$ *defined in Theorem 4.6.*

4.5.5.b) Merge

A Merge operation combines two sorted sequences S_1 and S_2 of length n_1 and n_2, respectively, into a single sorted sequence S of length $n_1 + n_2$. Checking it is thus equal to verifying that S is sorted and the union of S_1 and S_2, *c.f.* the previous subsection.

Corollary 4.13 (Merge Checker)

Checking $\mathrm{Merge}(\mathrm{S}_1, \mathrm{S}_2)$ *with error detection probability at least* $1 - \delta$ *is possible in time*

$$\mathcal{O}(T_{\textit{check-sort}}(|S_1| + |S_2|, p, \delta)).$$

4.5.5.c) GroupBy

The GroupBy operation is conceptually similar to aggregation (see Section 4.3), but passes all elements with the same key to the *group function* $g : [\mathit{Value}] \to \mathit{Value}$. Thus, for every key, all elements associated with it need to be sent to a single PE. This stage of a GroupBy operation can therefore be checked in a two-stage process similar to the sort checker. First, we verify that the redistributed data is a permutation of the input, ie, no items were lost, added, or changed. Then we need to verify that each item is where it should be using the GroupBy operation's mapping of keys to PEs (this can, for example, be implemented using a hash function mapping keys to PEs' indices). The group function needs to be checked separately by an appropriate *local checker*, which falls outside the scope of this work.

Corollary 4.14 (GroupBy Checker)

Checking the redistribution phase of GroupBy *on a sequence of n elements with probability at least* $1 - \delta$ *is possible in time* $\mathcal{O}(T_{\textit{check-sort}}(n, p, \delta))$.

4.5.5.d) Join

Similar to GroupBy, we can design an invasive checker for element redistribution in Join operations. The two common approaches to joins are the *sort-merge join* and *hash join* algorithm [GUW09]. Note that the sort checker can be used for both approaches because as far as data redistribution is concerned, a hash join is essentially a sort-merge join using the

hashes of the keys for sorting. To further verify that the distribution of keys to PEs is the same for both input sequences, we exchange the locally largest (smallest) keys with the following (preceding) PE and check that those are larger (smaller) than the local maximum (minimum). The correctness of the element redistribution then follows from their global sortedness.

Corollary 4.15 (Join Checker)

Checking the input redistribution phase of a hash or sort-merge join on two sequences of n_1 and n_2 elements with success probability at least $1 - \delta$ is possible in time $\mathcal{O}(T_{\text{check-sort}}(n_1 + n_2, p, \delta))$.

4.6 Experimental Evaluation

We developed implementations of our core checkers, sum aggregation and sorting. These were integrated into Thrill [Bin+16], an open-source data-parallel big data processing framework using modern C++ which is being developed in the group of Peter Sanders at Karlsruhe Institute of Technology.[7] Our code is available at `https://github.com/lorenzhs/thrill/tree/checkers`.[8] Thrill provides a high-performance environment that allows for easy implementation and testing of our methods while offering a convenient high-level interface to users.

Goals. The aim of our experiments is twofold. First, to show that our checkers achieve the predicted detection accuracy, and second, to demonstrate their practicability by showing that very little overhead is introduced by using a checker.

Manipulators. To test the efficacy of our checkers, we implemented *manipulators* that purposefully interfere with the computation and deliberately introduce faults. Manipulators are a flexible way to introduce a wide variety of classes of faults, allowing us to test our hypotheses on which kinds of faults are hardest to detect. It is easy to convince oneself that large-scale manipulation of the result is much easier to detect than subtle changes. Thus, our manipulators focus on the latter kind of change in the data. The specific manipulation techniques used with an operation are described in the respective subsections.

Implementation Details. As hash functions, we used CRC-32C, which is implemented in hardware in newer x86-64 processors with support for SSE 4.2 [Gop+11], and tabulation hashing [WC81; PT12] with 256 entries per table and four or eight tables for 32 and 64-bit values, respectively. We abbreviate the hash functions with 'CRC', 'Tab', and 'Tab64', respectively. Where needed, pseudo-random numbers are obtained from an MT19937 Mersenne Twister [MN98].

Platform. We conducted our scaling experiments on up to 128 nodes of bwUniCluster, each of which features two 14-core Intel Xeon E5–2660 v4 processors and 128 GiB of DDR4 main memory. The CPUs have a base clock of 2.0 GHz and boost clock up to 3.2 GHz, and hyperthreading is disabled. The machine was running Red Hat Enterprise Linux 7.4, and the code

[7] See `https://project-thrill.org` and `https://github.com/thrill/thrill/` for details. An archived copy of the project website is available at `https://web.archive.org/web/20200202015413/http://project-thrill.org/`.

[8] An archived copy of the code is available at `https://web.archive.org/web/20200925083644/https://codeload.github.com/lorenzhs/thrill/zip/checkers`.

was compiled with g++ 7.1.0. We treat each core as its own PE, running with 28 PEs per node. For our overhead experiments, we used an AMD Ryzen 9 3950X 16-core machine with a base clock of 3.5 GHz and boost clock up to 4.7 GHz. The machine has 64 GiB of DDR4 RAM, runs Ubuntu 18.04, and we used g++ 9.3.0 as the compiler. In both cases, the compiler flags `-O3 -march=native -DNDEBUG` were used.

4.6.1 Sum Aggregation

We implemented the sum aggregation checker of Section 4.3. As workload, we chose integers distributed according to a power-law distribution, with frequency $f(k;N) = 1/(kH_N)$ for the element of rank k. Here, N is the number of possible elements and H_N is the N-th harmonic number. This distribution naturally models many workloads, eg, wordcount over natural languages.

The tested configurations of the checker—number of iterations, number of buckets, hash function, and modulus parameter—are listed in Table 4.4. The first set of configurations was used for testing detection accuracy, the second for scaling experiments.

Implementation Details. To minimise local work, our implementation employs *bit-parallelism* where possible, eg, instead of computing eight four-bit hash values, we compute one 32-bit hash value and partition it into eight groups of four bits, which we treat as the output of the hash functions. This is implemented generically to satisfy any partition of a hash value

Table 4.4: Configurations tested for Sum Aggregation checker.
First set was used for accuracy tests, second set for scaling tests.

Configuration #iter × d m log $\hat{r}$	**Table size** (bits)	**Failure rate** δ	**Comment**
1×2 m31	64	$5 \cdot 10^{-1}$	high $\hat{r}$ is less effective than …
1×4 m31	128	$2.5 \cdot 10^{-1}$	↪ multiple iterations
4×2 m4	40	$1 \cdot 10^{-1}$	lower δ *and* size than above
4×4 m3	64	$2 \cdot 10^{-2}$	δ = 2 % for 64-bit table
4×4 m5	96	$6 \cdot 10^{-3}$	
4×8 m3	128	$3.9 \cdot 10^{-3}$	
4×8 m5	192	$6 \cdot 10^{-4}$	
4×8 m7	256	$3.1 \cdot 10^{-4}$	
5×16 m5	480	$7.2 \cdot 10^{-6}$	
6×32 m9	1 920	$1.3 \cdot 10^{-9}$	
8×16 m15	2 048	$2.3 \cdot 10^{-10}$	
4×256 m15	16 384	$2.4 \cdot 10^{-10}$	
5×128 m11	7 680	$3.9 \cdot 10^{-11}$	
8×256 m15	32 768	$5.8 \cdot 10^{-20}$	lower local work, larger size
16×16 m15	4 096	$5.4 \cdot 10^{-20}$	higher local work, smaller size

Table 4.5: Manipulators used for sum aggregation checking.

Name	Manipulation applied
Bitflip	flips a random bit in the input
RandKey	randomise the key of a random element
SwitchValues	switches the values of two random elements
IncKey	increments the key of a random element
$IncDec_n$	acts on $2n$ elements with distinct keys, …
↪ using $n = 1$	incrementing the keys n elements and …
↪ and $n = 2$	decrementing that of n other elements

into groups. Since 64 hash bits suffice to guarantee a failure probability of nearly 10^{-20} (see Table 4.4), evaluating a single hash function suffices in all practically relevant configurations.

To minimise the cost of adding modulo r, we use 64-bit values for the buckets internally, add normally, and perform the expensive modulo step only if the addition would overflow. This can be detected cheaply with a jump-on-overflow instruction.

Detection Accuracy. The manipulators we used are listed in Table 4.5. Figure 4.3 demonstrates that our checkers achieve the theoretically predicted performance in practice, even when using hash functions with weaker guarantees. We can see that Lemma 4.2 generally overestimates the failure probability introduced by the modulus. Note that the significance of the results for configurations with low δ is limited, as the expected number of failures in 100 000 runs for eg, the 4×8 Tab m7 configuration is only 31.1.

We note that CRC-32C appears to work very well for subtle manipulations, which is likely because it was designed so that small changes in the input cause many output bits to change. This makes it likely that a change in an element's key causes it to be sent to a different bucket in at least one iteration. However, the least significant bits appear to change in similar ways for different inputs, as indicated by the $IncDec_1$ measurements, causing an elevated failure rate in this setting. Tabulation hashing performs quite uniformly well across the board, which is not entirely unexpected given its high degree of independence.

Local Processing Overhead. We measured the sequential overhead of the sum aggregation checker on the aforementioned AMD Ryzen 9 3950X machine, as single-node performance on bwUniCluster proved to be too inconsistent to yield useful results (this is likely due to thermal limitations). The workload consisted only of passing the inputs into the checker, with no aggregation performed. The results are shown in Table 4.6. The configurations used provide high confidence in the correctness of the result and cover a wide range of practically relevant parameter choices. Clearly, the number of iterations is the dominating factor in overhead, and CRC-32C is somewhat faster than 64-bit tabulation hashing. This is no surprise, as CRC-32C is implemented in hardware and provides only 32 hash bits. However, our more accurate configurations require 64 bits of hash function output and are thus implemented with tabulation hashing. We can see that $\delta \approx 2.4 \cdot 10^{-10}$ can be achieved in just 2.2 ns per element on this high-end desktop machine which achieved an average clock speed of 4.55 GHz in this benchmark, ie,

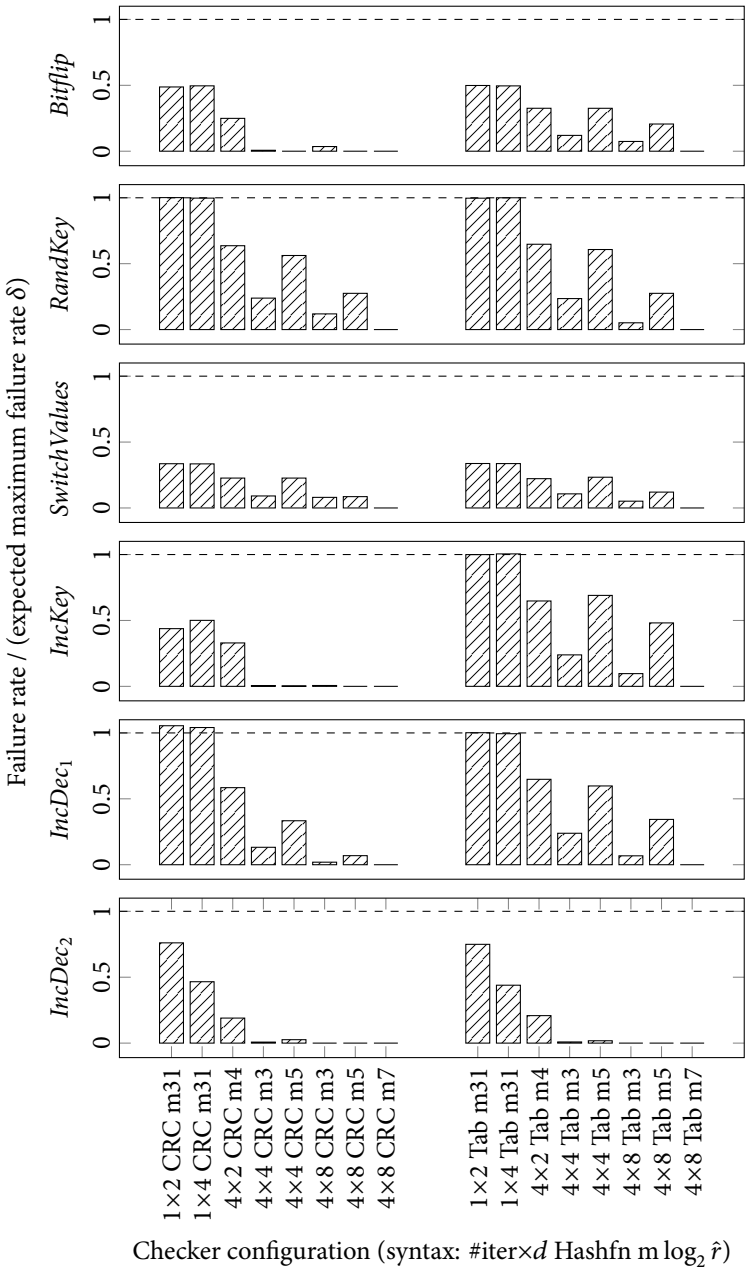


Figure 4.3: Accuracy of the Sum Aggregation checker for different manipulators. The 50 000 input elements follow a power-law distribution with 10^6 possible values, executed on 4 PEs and measured over 100 000 iterations. Configurations as in Table 4.4.

Table 4.6: Overhead of sum aggregation checker: checker local input processing time for 10^6 pairs of 64-bit integers, 1000 runs. Refer to Table 4.4 for details on the configurations.

Configuration	δ	Time per item (ns)	Approx. #cycles
5×16 CRC m5	$7.2 \cdot 10^{-6}$	2.9	13
6×32 CRC m9	$1.3 \cdot 10^{-9}$	3.1	14
8×16 CRC m15	$2.3 \cdot 10^{-10}$	3.8	17
4×256 CRC m15	$2.4 \cdot 10^{-10}$	2.2	10
5×128 Tab64 m11	$3.9 \cdot 10^{-11}$	4.1	19
8×256 Tab64 m15	$5.8 \cdot 10^{-20}$	5.8	26
16×16 Tab64 m15	$5.4 \cdot 10^{-20}$	8.5	38

approximately 10 CPU cycles. A configuration with 4 *KiB* table size and $\delta \approx 5.8 \cdot 10^{-20}$ can be realised with as little overhead as 5.8 ns per element, approximately 26 cycles. Similar accuracy with a smaller table size of just 512 Bytes and thus very little communication can be achieved in 8.5 *ns* per item, approximately 38 cycles.

To put these numbers into perspective, the main reduce operation takes approximately 59 ns per element using a single core of the same machine. This surprisingly low overhead was achieved by carefully engineering our implementation as described in Section 4.6.1. A naïve implementation would likely cause an order of magnitude more overhead. Observe that this experiment measures only the checker's *local work*. Therefore, it is not representative of the overhead in a parallel and distributed setting, where the communication required for aggregation tends to dominate overall running time. We will shortly also present experiments on a cluster computer measuring the impact of our checkers on the running times of a distributed sum aggregation workload.

We compare our checker to PIRS [Yi+09] using the original implementation and the world cup data set.[9] We had to slightly fix the code of Yi et al. so that it would compile with modern compilers. Furthermore, we removed time measurement code that represented 75 % of the overall running time, likely because the relative cost of system calls has changed since the code was originally published.[10] Executing PIRS with the input data located in main memory yielded 52 *ns* and 591 *ns* update times per item for count and sum queries, respectively, after deducting the time for input parsing.[11] We also implemented their benchmark with our checker, and the per-item update time of our checker is 3.1 *ns* for count and 3.3 *ns* for sum aggregation in a sequential execution of our checker with the '4×256 CRC m15' configuration on the same

[9] Source: `http://www.cs.utah.edu/~lifeifei/pirs/`, archived at `https://web.archive.org/web/20120519232455/http://www.cs.utah.edu/~lifeifei/pirs/` on 4th Sept. 2020.

[10] Our changes are available at `https://algo2.iti.kit.edu/huebschle/pirs.diff`, archived at `https://web.archive.org/web/20200904115701/https://algo2.iti.kit.edu/huebschle/pirs.diff`.

[11] This is a speedup of nearly 20 over the measurements that Yi et al. report on a 2.8 GHz Intel Pentium CPU, likely from early 2002. In comparison, the Ryzen 9 3950X we used was introduced in late 2019, nearly 18 years later. Yi et al. report 0.98 μs and 8.01 μs for count and sum updates, respectively. Thus, their code runs 18.8 and 13.5 times faster on our machine, respectively. This is likely at least partially due to its reliance on arithmetic modulo a prime number with 64-bit integers, which was not natively implemented in hardware in their machine.

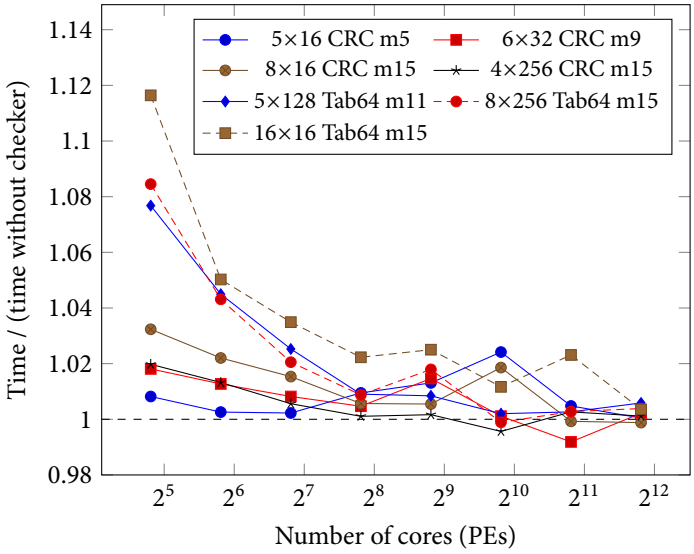


Figure 4.4: Weak scaling experiment for Sum Aggregation checker with 125 000 items per PE, following a Zipf (power-law) distribution. Average over 1000 runs. Configurations as in Table 4.4.

machine. Our checker is over 16 times faster in this benchmark for count checking, and around 180 times faster for sum aggregation checking.

Scaling Behaviour. We performed weak scaling experiments, the results of which are shown in Figure 4.4. These measurements suffer from a noticeable amount of noise because reduction and checker are interleaved—elements are forwarded to the checker as they are passed to the reduction. This is necessitated by the design of Thrill and would be similar related Big Data frameworks. As a result, we measure the entire reduce-check pipeline, the running time of which is influenced by multiple sources of variability, including the network, causing the aforementioned noise. We note that the results for a single node remain consistent with the sequential overhead measurements of the preceding paragraph.

It is clearly visible that the overhead introduced by the checkers is within the fluctuations introduced by the network and other sources of noise. Furthermore, starting with four nodes, the data exchange necessary for the reduction dominates overall running time, which becomes nearly independent of the checker's configuration and thus accuracy, although some impact of the number of iterations on running time remains visible. Nonetheless, the average overhead over all configurations is a mere 1.1 % when using more than a single node. Observe that even the slowest and most accurate configuration, '16×16 Tab64 m15' with $\delta < 6 \cdot 10^{-20}$, thus providing near-certainty in the correctness of the result, adds only 2.4 % to the average running time on two or more nodes (56 or more cores/PEs in Figure 4.4).

Table 4.7: Manipulators used for sort/permutation checking.

Name	Manipulation applied
Bitflip	flips a random bit in the input
Increment	increment some element's value
Randomise	set some element to a random value
Reset	reset some element to the default value (0)
SetEqual	set some element equal to a different one

4.6.2 Permutation and Sorting

In this subsection, we evaluate the permutation and sorting checker of Section 4.4 with a workload of 10^6 integers chosen uniformly at random from the range $[0 \,..\, 10^8 - 1]$. We implemented the hash-based permutation and sorting checker of Lemma 4.4, and truncate the output of the hash function to the $\log H$ least significant bits for different values of H. Our measurements are for a single iteration of the checker. Manipulations are applied *before* sorting to test the permutation checker and not the trivial sortedness check.

Detection Accuracy. We measured the sort checker's detection accuracy with CRC-32C and tabulation hashing, two fast real-world hash functions with limited randomness and different characteristics. This setup examines whether these hash functions provide sufficient randomness to detect manipulations and are, therefore, suitable for use in the checker. In Figure 4.5 we show the results of this experiment. We can see that tabulation hashing provides sufficient randomness to achieve the theoretically predicted detection accuracy on all tested manipulators (see Table 4.7). In contrast, CRC-32C does *not* seem to provide sufficient randomness. In particular, we observe a significantly increased failure rate using the *Increment* manipulator where a single element of the input was incremented by 1. The proportion of excessive failures rises with the number of hash bits used. This suggests that incrementing the input to the hash function affects only the least significant bits of the output with the required reliability. This is a failure of the so-called avalanche test and a well-known weakness of the CRC family [HSZ08]. No significant deviations from expected performance were observed for the other manipulators because these affect bits at different positions in the input, whereas the *Increment* manipulator disproportionately changes the least significant bits in the input.

Running Time Overhead. On the aforementioned AMD Ryzen 9 3950X machine, the overhead for local processing of input and output of the sorting operation was just under 1 ns per element for CRC32, and around 2 ns when using 32-bit tabulation hashing. However, once we use it while actually sorting the data, the overhead all but disappears. When sorting 10^8 integers using all 16 cores of the machine, we measured slowdowns of 1.2 % for the CRC-32C configurations and 2.5 % for the configurations using tabulation hashing. Because we have to compute the full hash regardless of how many of its bits we use, the configuration had no measurable influence on the running time.

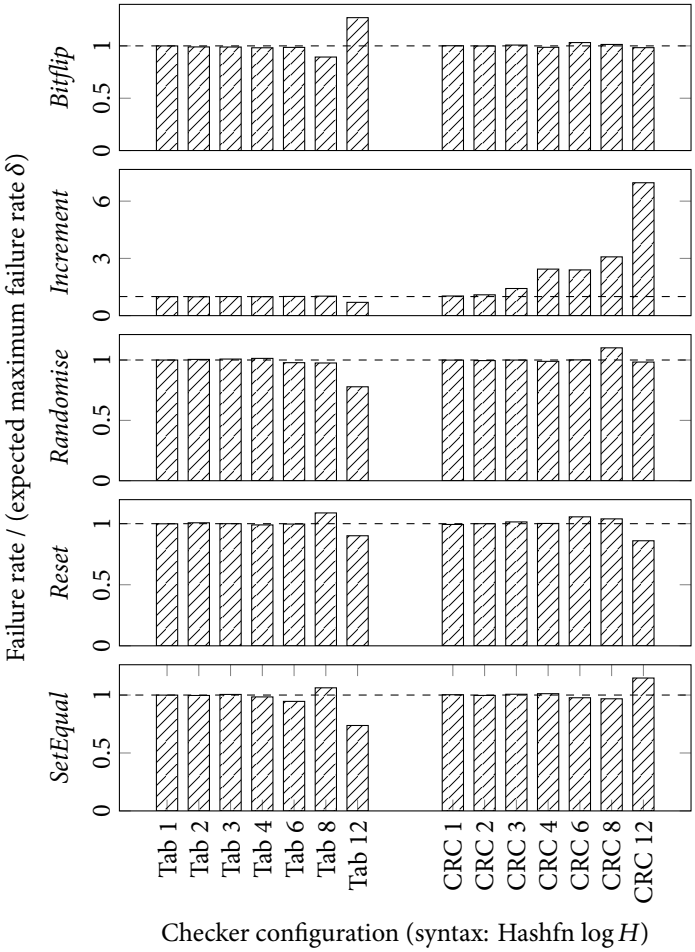


Figure 4.5: Accuracy of the Permutation/Sort checker for different manipulators. Uniformly distributed input with 10^8 possible values, 10^6 input elements, 4 PEs, 100 000 iterations for each manipulator. Configuration parameter $\log H$ describes to how many bits the hash function is truncated.

We do not present scaling experiments for the permutation checker, as the only communication is a global reduction on a single integer value and one message sent and received per PE, both also containing exactly one integer value. However, we expect the overhead in a distributed system to be even lower. This is because all time spent on the data exchange that is necessary for sorting the input is in addition to the local sorting step measured above.

4.7 Conclusions

We have shown that probabilistic checking of many distributed big data operations is possible in a communication-efficient manner. Our experiments show excellent scaling and around 2 % running time overhead in a distributed setting for sum aggregation and sorting. This is true even when near-certainty in the correctness of the result is required. Accuracy guarantees predicted by theoretical analysis are achieved in practice even when using hash functions with limited randomness. Moreover, our checkers have negligible memory overhead. The basic methods used in our checkers are extremely simple, making manual verification of the correctness of the checker much more feasible than of the aggregation to be checked. The existence of such checkers could speed up the development cycles of operations in big data processing frameworks by providing correctness checks and allowing for graceful degradation at execution time by falling back to a simpler but slower method should a computation fail.

Future Work. Lower bounds on communication volume and latency for probabilistic distributed checkers would offer some insight into how far from an optimal solution our checkers are. Furthermore, it would be interesting to see whether more operations can be checked without the need for a certificate or requiring the purported result to be available at each PE. For example, could a probabilistic minimum or maximum aggregation checker with sublinear communication exist?

It would also be interesting to know whether the sum aggregation checker can be adapted for other data types such as floating-point numbers without suffering from numerical instability issues such as catastrophic cancellation.

Acknowledgements. The authors acknowledge support by the state of Baden-Württemberg through bwHPC.

Appendix

List of Acronyms

BSP . . . . . Bulk-Synchronous Parallel (machine model) 7

CMM . . . . . Continuous Monitoring Model (machine model) 11

CRCW PRAM . . . Concurrent Read, Concurrent Write PRAM 5

CREW PRAM . . . Concurrent Read, Exclusive Write PRAM 5

HPC . . . . . High-Performance Computing 15

i.i.d. . . . . independent and identically distributed (random variables) . . . 85

I/O . . . . . input/output 111

MPI . . . . . Message-Passing Interface 2

NUMA . . . . . Non-Uniform Memory Access 87

PE . . . . . processing element (eg, CPU core or machine) 6

PRAM . . . . . Parallel Random Access Machine 5

R-MAT . . . . . Recursive MATrix (random graph model) 104

RAM . . . . . Random Access Machine 4

SM-parallel . . . . shared-memory parallel 2

SPMD . . . . . Single Program Multiple Data 12

SSD . . . . . solid-state drive (storage technology) 111

w.h.p. . . . . with high probability 14

List of Algorithms

List of Figures

List of Tables

List of Theorems

Publications and Supervised Theses

In Conference Proceedings

Moritz Baum, Julian Dibbelt, Lorenz Hübschle-Schneider, Thomas Pajor and Dorothea Wagner. 'Speed-Consumption Tradeoff for Electric Vehicle Route Planning'. In: *14th Workshop on Algorithmic Approaches for Transportation Modelling, Optimization, and Systems (ATMOS)*. Sept. 2014, pages 138–151. DOI: 10.4230/OASIcs.ATMOS.2014.138

Lorenz Hübschle-Schneider and Rajeev Raman. 'Tree Compression with Top Trees Revisited'. In: *14th International Symposium on Experimental Algorithms (SEA)*. June 2015, pages 15–27. DOI: 10.1007/978-3-319-20086-6_2

Lorenz Hübschle-Schneider and Peter Sanders. 'Communication Efficient Algorithms for Top-k Selection Problems'. In: *30th IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. May 2016, pages 659–668. DOI: 10.1109/IPDPS.2016.45

Lorenz Hübschle-Schneider and Peter Sanders. 'Communication Efficient Checking of Big Data Operations'. In: *32nd IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. May 2018, pages 650–659. DOI: 10.1109/IPDPS.2018.00074

Lorenz Hübschle-Schneider and Peter Sanders. 'Parallel Weighted Random Sampling'. In: *27th European Symposium on Algorithms (ESA)*. 2019, 59:1–59:24. DOI: 10.4230/LIPIcs.ESA.2019.59

Lorenz Hübschle-Schneider and Peter Sanders. 'Brief Announcement: Communication-Efficient Weighted Reservoir Sampling from Fully Distributed Data Streams'. In: *32nd ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. 2020, pages 543–545. DOI: 10.1145/3350755.3400287

Journal Articles

Peter Sanders, Sebastian Lamm, Lorenz Hübschle-Schneider, Emanuel Schrade and Carsten Dachsbacher. 'Efficient Parallel Random Sampling – Vectorized, Cache-Efficient, and Online'. In: *ACM Transactions on Mathematical Software (TOMS)* 44.3 (Apr. 2018), 29:1–29:14: ACM. DOI: 10.1145/3157734

Lorenz Hübschle-Schneider and Peter Sanders. 'Linear Work Generation of R-MAT Graphs'. In: *Network Science* 8.4 (2020), pages 543–550: Cambridge University Press. DOI: 10.1017/nws.2020.21

Technical Reports

Lorenz Hübschle-Schneider and Rajeev Raman. 'Tree Compression with Top Trees Revisited'. In: *Computing Research Repository (CoRR)* (June 2015). arXiv: 1506.04499 [cs.DS]

Lorenz Hübschle-Schneider, Peter Sanders and Ingo Müller. 'Communication Efficient Algorithms for Top-*k* Selection Problems'. In: *Computing Research Repository (CoRR)* (Feb. 2015). arXiv: 1502.03942 [cs.DS]

Peter Sanders, Sebastian Lamm, Lorenz Hübschle-Schneider, Emanuel Schrade and Carsten Dachsbacher. 'Efficient Random Sampling – Parallel, Vectorized, Cache-Efficient, and Online'. In: *Computing Research Repository (CoRR)* (Oct. 2016). arXiv: 1610.05141 [cs.DS]

Lorenz Hübschle-Schneider and Peter Sanders. 'Communication Efficient Checking of Big Data Operations'. In: *Computing Research Repository (CoRR)* (Oct. 2017). arXiv: 1710.08255 [cs.DS]

Lorenz Hübschle-Schneider and Peter Sanders. 'Parallel Weighted Random Sampling'. In: *Computing Research Repository (CoRR)* (Mar. 2019). arXiv: 1903.00227 [cs.DS]

Lorenz Hübschle-Schneider and Peter Sanders. 'Linear Work Generation of R-MAT Graphs'. In: *Computing Research Repository (CoRR)* (May 2019). arXiv: 1905.03525 [cs.DS]

Lorenz Hübschle-Schneider and Peter Sanders. 'Communication-Efficient (Weighted) Reservoir Sampling from Fully Distributed Data Streams'. In: *Computing Research Repository (CoRR)* (Oct. 2019). arXiv: 1910.11069 [cs.DS]

Theses

Lorenz Hübschle-Schneider. 'Speed-Consumption Trade-Off for Electric Vehicle Routing'. Bachelor's Thesis. Department of Computer Science, Karlsruhe Institute of Technology, Germany, June 2013

Lorenz Hübschle-Schneider. 'Compression Methods for Labelled Trees'. Master's Thesis. Department of Computer Science, University of Leicester, United Kingdom, Sept. 2014

Supervised Theses

Jan Ellmers. „Dynamische Baumkompression mit Top Trees". In German. Bachelor's Thesis. Department of Computer Science, Karlsruhe Institute of Technology, Germany, Nov. 2018

Collin Lorbeer. „Kommunikationseffizientes Branch-and-Bound für das Rucksackproblem". In German. Bachelor's Thesis. Department of Computer Science, Karlsruhe Institute of Technology, Germany, Apr. 2020

Bibliography

[Abt+10] Dennis Abts, Michael R. Marty, Philip M. Wells, Peter Klausler and Hong Liu. 'Energy proportional datacenter networks'. In: *37th International Symposium on Computer Architecture (ISCA)*. 2010, pages 338–347. DOI: 10.1145/1815961.1816004. [see page 3]

[AD85] Joachim H. Ahrens and Ulrich Dieter. 'Sequential Random Sampling'. In: *ACM Transactions on Mathematical Software (TOMS)* 11.2 (1985), pages 157–169: ACM. DOI: 10.1145/214392.214402. [see page 79]

[Ahr+16] Wolfgang Ahrendt, Bernhard Beckert, Richard Bubel, Reiner Hähnle, Peter H. Schmitt and Mattias Ulbrich, editors. *Deductive Software Verification - The KeY Book - From Theory to Practice*. Volume 10001. Lecture Notes in Computer Science (LNCS). Springer, 2016. ISBN: 978-3-319-49811-9. DOI: 10.1007/978-3-319-49812-6. [see page 116]

[Ale+14] Alexander Alexandrov, Rico Bergmann, Stephan Ewen, Johann-Christoph Freytag, Fabian Hueske, Arvid Heise, Odej Kao, Marcus Leich, Ulf Leser, Volker Markl et al. 'The Stratosphere platform for big data analytics'. In: *The VLDB Journal* 23.6 (2014), pages 939–964. DOI: 10.1007/s00778-014-0357-y. [see pages 15, 111]

[Arg+08] Lars Arge, Michael T. Goodrich, Michael J. Nelson and Nodari Sitchinava. 'Fundamental parallel algorithms for private-cache chip multiprocessors'. In: *20th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. 2008, pages 197–206. DOI: 10.1145/1378533.1378573. [see page 10]

[Arm+15] Michael Armbrust, Reynold S. Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K. Bradley, Xiangrui Meng, Tomer Kaftan, Michael J. Franklin, Ali Ghodsi and Matei Zaharia. 'Spark SQL: Relational Data Processing in Spark'. In: *ACM SIGMOD International Conference on Management of Data*. 2015, pages 1383–1394. DOI: 10.1145/2723372.2742797. [see page 113]

[Arr02] Richard Arratia. 'On the amount of dependence in the prime factorization of a uniform random integer'. In: *Contemporary combinatorics* 10 (2002), pages 29–91: Springer Science & Business Media. Page 36 . [see pages 56, 58]

[AS16] Yaroslav Akhremtsev and Peter Sanders. 'Fast Parallel Operations on Search Trees'. In: *23rd IEEE International Conference on High Performance Computing (HiPC)*. 2016, pages 291–300. DOI: 10.1109/HiPC.2016.042. [see page 88]

[Aue13] Felix Auerbach. „Das Gesetz der Bevölkerungskonzentration". In: *Petermanns Geographische Mitteilungen* 59 (1913), Seiten 74–76 . [see page 24]

[Axt+15] Michael Axtmann, Timo Bingmann, Peter Sanders and Christian Schulz. 'Practical Massively Parallel Sorting'. In: *27th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. 2015, pages 13–23. DOI: 10.1145/2755573.2755595. [see pages 7, 24, 29, 30, 151]

[Axt+17] Michael Axtmann, Sascha Witt, Daniel Ferizovic and Peter Sanders. 'In-Place Parallel Super Scalar Samplesort (IPSSSSo)'. In: *25th European Symposium on Algorithms (ESA)*. 2017, 9:1–9:14. DOI: 10.4230/LIPIcs.ESA.2017.9. [see page 5]

[BA99] Albert-László Barabási and Réka Albert. 'Emergence of Scaling in Random Networks'. In: *Science* 286.5439 (1999), pages 509–512: AAAS. DOI: 10.1126/science.286.5439.509. [see page 104]

[Bab+91] László Babai, Lance Fortnow, Leonid A. Levin and Mario Szegedy. 'Checking Computations in Polylogarithmic Time'. In: *23rd ACM Symposium on Theory of Computing (STOC)*. 1991, pages 21–31. DOI: 10.1145/103418.103428. [see page 116]

[Bad+20] David A. Bader, Jonathan Berry, Simon Kahan, Richard Murphy, E. Jason Riedy, Jeremiah Willcock, Anton Korzh and Marcin Zalewski. *Graph 500 reference implementation*. Code archived at https://web.archive.org/web/20200925082734/https://codeload.github.com/graph500/graph500/zip/3.0.1 on 25th Sept. 2020. 2020. URL: https://github.com/graph500/graph500/. [see page 106]

[Bal+95] Vasanth Bala, Jehoshua Bruck, Robert Cypher, Pablo Elustondo, Alex Ho, Ching-Tien Ho, Shlomo Kipnis and Marc Snir. 'CCL: A Portable and Tunable Collective Communication Library for Scalable Parallel Computers'. In: *IEEE Transactions on Parallel and Distributed Systems* 6.2 (1995), pages 154–164: IEEE. DOI: 10.1109/71.342126. [see page 13]

[Bat68] Kenneth E. Batcher. 'Sorting Networks and Their Applications'. In: *American Federation of Information Processing Societies (AFIPS) Conference*. 1968, pages 307–314. DOI: 10.1145/1468075.1468121. [see page 68]

[Bau+14] Moritz Baum, Julian Dibbelt, Lorenz Hübschle-Schneider, Thomas Pajor and Dorothea Wagner. 'Speed-Consumption Tradeoff for Electric Vehicle Route Planning'. In: *14th Workshop on Algorithmic Approaches for Transportation Modelling, Optimization, and Systems (ATMOS)*. Sept. 2014, pages 138–151. DOI: 10.4230/OASIcs.ATMOS.2014.138. [see page 153]

[BCH13] Luiz André Barroso, Jimmy Clidaras and Urs Hölzle. *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines, Second Edition*. Synthesis Lectures on Computer Architecture. Morgan & Claypool Publishers, 2013. ISBN: 978-1-62705-009-8. DOI: 10.2200/S00516ED2V01Y201306CAC024. [see page 118]

[BD18] Guy E. Blelloch and Laxman Dhulipala. *Introduction to Parallel Algorithms, 15-853: Algorithms in the Real World*. 2018. URL: https://ldhulipala.github.io/notes/parallel.pdf, archived at https://web.archive.org/web/20200921170903/https://ldhulipala.github.io/notes/parallel.pdf on 21st Sept. 2020. [see page 55]

[Ber+08] Keren Bergman, Shekhar Borkar, Dan Campbell, William Carlson, William Dally, Monty Denneau, Paul Franzon, William Harrod, Kerry Hill, Jon Hiller et al. *Exascale computing study: Technology challenges in achieving exascale systems*. Technical report TR-2008-13. Defense Advanced Research Projects Agency Information Processing Techniques Office (DARPA IPTO), 2008. URL: http://www.cse.nd.edu/Reports/2008/TR-2008-13.pdf, archived at https://web.archive.org/web/20190429051308/http://www.cse.nd.edu/Reports/2008/TR-2008-13.pdf on 29th Apr. 2019. [see page 3]

[Ber+20] Petra Berenbrink, David Hammer, Dominik Kaaser, Ulrich Meyer, Manuel Penschuck and Hung Tran. 'Simulating Population Protocols in Sub-Constant Time per Interaction'. In: *28th European Symposium on Algorithms (ESA)*. 2020, 16:1–16:22. DOI: 10.4230/LIPIcs.ESA.2020.16. [see page 58]

[BH83] Ken RW Brewer and Muhammad Hanif. *Sampling with unequal probabilities*. Volume 15. Lecture Notes in Computer Science (LNCS). Springer, 1983. DOI: 10.1007/978-1-4684-9407-5. [see page 58]

[Bil+16] Gianfranco Bilardi, Andrea Pietracaprina, Geppino Pucci, Michele Scquizzato and Francesco Silvestri. 'Network-Oblivious Algorithms'. In: *Journal of the ACM* 63.1 (2016), 3:1–3:36: ACM. DOI: 10.1145/2812804. [see page 9]

[Bin+16] Timo Bingmann, Michael Axtmann, Emanuel Jöbstl, Sebastian Lamm, Huyen Chau Nguyen, Alexander Noe, Sebastian Schlag, Matthias Stumpp, Tobias Sturm and Peter Sanders. 'Thrill: High-Performance Algorithmic Distributed Batch Data Processing with C++'. In: *IEEE International Conference on Big Data*. 2016, pages 172–183. DOI: 10.1109/bigdata.2016.7840603. [see pages 15, 109, 111, 113, 130]

[Bin18a] Timo Bingmann. 'Scalable string and suffix sorting: Algorithms, techniques, and tools'. PhD thesis. Department of Computer Science, Karlsruhe Institute of Technology, Germany, 2018. URL: https://arxiv.org/abs/1808.00963. [see page 4]

[Bin18b] Timo Bingmann. *TLX: Collection of Sophisticated C++ Data Structures, Algorithms, and Miscellaneous Helpers*. 2018. URL: https://panthema.net/tlx, archived at https://web.archive.org/web/20191103202232/https://panthema.net/tlx on 3rd Nov. 2019. [see page 88]

[BK89] Manuel Blum and Sampath Kannan. 'Designing Programs That Check Their Work'. In: *21st ACM Symposium on Theory of Computing (STOC)*. 1989, pages 86–97. DOI: 10.1145/73007.73015. [see page 116]

[BK95] Manuel Blum and Sampath Kannan. 'Designing Programs that Check Their Work'. In: *Journal of the ACM* 42.1 (1995), pages 269–291: ACM. DOI: 10.1145/200836.200880. [see pages 116, 124]

[BKS17] Paul Beame, Paraschos Koutris and Dan Suciu. 'Communication Steps for Parallel Query Processing'. In: *Journal of the ACM* 64.6 (2017), 40:1–40:58: ACM. DOI: 10.1145/3125644. [see page 8]

[BL13] Karl Bringmann and Kasper Green Larsen. 'Succinct sampling from discrete distributions'. In: *45th ACM Symposium on Theory of Computing (STOC)*. 2013, pages 775–782. DOI: 10.1145/2488608.2488707. [see pages 58, 109]

[Blä+19] Thomas Bläsius, Tobias Friedrich, Maximilian Katzmann, Ulrich Meyer, Manuel Penschuck and Christopher Weyand. 'Efficiently Generating Geometric Inhomogeneous and Hyperbolic Random Graphs'. In: *27th European Symposium on Algorithms (ESA)*. 2019, 21:1–21:14. DOI: 10.4230/LIPIcs.ESA.2019.21. [see page 104]

[Ble89] Guy E. Blelloch. 'Scans as Primitive Parallel Operations'. In: *IEEE Transactions on Computers* 38.11 (1989), pages 1526–1538: IEEE. DOI: 10.1109/12.42122. [see page 63]

[BLM13] Stéphane Boucheron, Gábor Lugosi and Pascal Massart. *Concentration Inequalities - A Nonasymptotic Theory of Independence*. Oxford University Press, 2013. ISBN: 978-0-19-953525-5. DOI: 10.1093/acprof:oso/9780199535255.001.0001. [see page 85]

[BM72] Rudolf Bayer and Edward M. McCreight. 'Organization and Maintenance of Large Ordered Indexes'. In: *Acta Informatica* 1.3 (1972), pages 173–189. DOI: 10.1007/BF00288683. [see page 14]

[BO03] Brian Babcock and Chris Olston. 'Distributed Top-K Monitoring'. In: *ACM SIGMOD International Conference on Management of Data*. 2003, pages 28–39. DOI: 10.1145/872757.872764. [see pages 23, 25–27]

[Bor05] Shekhar Y. Borkar. 'Designing Reliable Systems from Unreliable Components: The Challenges of Transistor Variability and Degradation'. In: *IEEE Micro* 25.6 (2005), pages 10–16. DOI: 10.1109/MM.2005.110. [see page 111]

[Bor13] Shekhar Borkar. 'Exascale Computing - A Fact or a Fiction?' In: *27th IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. May 2013, page 3. Keynote presentation. DOI: 10.1109/IPDPS.2013.121. [see page 3]

[Bos+09] George Bosilca, Remi Delmas, Jack J. Dongarra and Julien Langou. 'Algorithm-based fault tolerance applied to high performance computing'. In: *Journal of Parallel and Distributed Computing* 69.4 (2009), pages 410–416. DOI: 10.1016/j.jpdc.2008.12.002. [see page 117]

[BOV15] Vladimir Braverman, Rafail Ostrovsky and Gregory Vorsanger. 'Weighted sampling without replacement from data streams'. In: *Information Processing Letters* 115.12 (2015), pages 923–926: Elsevier. DOI: 10.1016/j.ipl.2015.07.007. [see pages 58, 60]

[BP17] Karl Bringmann and Konstantinos Panagiotou. 'Efficient Sampling Methods for Discrete Distributions'. In: *Algorithmica* 79.2 (2017), pages 484–508: Springer. DOI: 10.1007/s00453-016-0205-0. [see pages 54, 58, 59, 70, 79, 80]

[Bra+13] Benjamin Braun, Ariel J. Feldman, Zuocheng Ren, Srinath T. V. Setty, Andrew J. Blumberg and Michael Walfish. 'Verifying computations with state'. In: *24th ACM Symposium on Operating Systems Principles (SOSP)*. 2013, pages 341–357. DOI: 10.1145/2517349.2522733. [see page 116]

[BRN20] Altan Birler, Bernhard Radke and Thomas Neumann. 'Concurrent online sampling for all, for free'. In: *16th International Workshop on Data Management on New Hardware (DaMoN)*. 2020, 5:1–5:8. DOI: 10.1145/3399666.3399924. [see page 59]

[BSS20] Timo Bingmann, Peter Sanders and Matthias Schimek. 'Communication-Efficient String Sorting'. In: *34th IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 2020, pages 137–147. DOI: 10.1109/IPDPS47924.2020.00024. [see page 16]

[Cha+07] Ernie Chan, Marcel Heimlich, Avi Purkayastha and Robert A. van de Geijn. 'Collective communication: theory, practice, and experience'. In: *Concurrency and Computation: Practice and Experience* 19.13 (2007), pages 1749–1783: Wiley. DOI: 10.1002/cpe.1206. [see page 6]

[Cha+14] Amit Chakrabarti, Graham Cormode, Andrew McGregor and Justin Thaler. 'Annotations in Data Streams'. In: *ACM Transactions on Algorithms* 11.1 (2014), 7:1–7:30: ACM. DOI: 10.1145/2636924. [see page 117]

[Cha82] M. T. Chao. 'A general purpose unequal probability sampling plan'. In: *Biometrika* 69.3 (1982), pages 653–656: Oxford University Press. DOI: 10.1093/biomet/69.3.653. [see pages 58, 60]

[Chr+13] Michael Christ, James Demmel, Nicholas Knight, Thomas Scanlon and Katherine A. Yelick. 'Communication lower bounds and optimal algorithms for programs that reference arrays - Part 1'. In: *Computing Research Repository (CoRR)* (Aug. 2013). arXiv: 1308.0068 [math.CA]. [see page 10]

[CK07] Edith Cohen and Haim Kaplan. 'Summarizing data using bottom-k sketches'. In: *26th ACM Symposium on Principles of Distributed Computing (PODC)*. 2007, pages 225–234. DOI: 10.1145/1281100.1281133. [see page 58]

[CL03] Fan R. K. Chung and Linyuan Lu. 'The Average Distance in a Random Graph with Given Expected Degrees'. In: *Internet Mathematics* 1.1 (2003), pages 91–113. DOI: 10.1080/15427951.2004.10129081. [see page 59]

[Clo53] Charles Clos. 'A study of non-blocking switching networks'. In: *Bell System Technical Journal* 32.2 (1953), pages 406–424: Wiley Online Library. DOI: 10.1002/j.1538-7305.1953.tb01433.x. [see page 2]

[CM05] Graham Cormode and S. Muthukrishnan. 'An improved data stream summary: the count-min sketch and its applications'. In: *Journal of Algorithms* 55.1 (2005), pages 58–75. DOI: 10.1016/j.jalgor.2003.12.001. [see page 118]

[CMT12] Graham Cormode, Michael Mitzenmacher and Justin Thaler. 'Practical verified computation with streaming interactive proofs'. In: *3rd Innovations in Theoretical Computer Science Conference (ITCS)*. 2012, pages 90–112. DOI: 10.1145/2090236.2090245. [see pages 116, 117]

[CMT13] Graham Cormode, Michael Mitzenmacher and Justin Thaler. 'Streaming Graph Computations with a Helpful Advisor'. In: *Algorithmica* 65.2 (2013), pages 409–442: Springer. DOI: 10.1007/s00453-011-9598-y. [see page 117]

[CMY11] Graham Cormode, S. Muthukrishnan and Ke Yi. 'Algorithms for distributed functional monitoring'. In: *ACM Transactions on Algorithms* 7.2 (2011), 21:1–21:20: ACM. DOI: 10.1145/1921659.1921667. [see page 11]

[Col88] Richard Cole. 'Parallel Merge Sort'. In: *SIAM Journal on Computing* 17.4 (1988), pages 770–785: SIAM. DOI: 10.1137/0217049. [see page 55]

[Cor+10] Graham Cormode, S. Muthukrishnan, Ke Yi and Qin Zhang. 'Optimal sampling from distributed streams'. In: *29th ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems (PODS)*. 2010, pages 77–86. DOI: 10.1145/1807085.1807099. [see page 59]

[Cor+12] Graham Cormode, S. Muthukrishnan, Ke Yi and Qin Zhang. 'Continuous sampling from distributed streams'. In: *Journal of the ACM* 59.2 (2012), 10:1–10:25: ACM. DOI: 10.1145/2160158.2160163. [see page 59]

[Cor13] Graham Cormode. 'The continuous distributed monitoring model'. In: *ACM SIGMOD Record* 42.1 (2013), pages 5–14. DOI: 10.1145/2481528.2481530. [see page 11]

[Cou05] National Research Council. *Getting Up to Speed: The Future of Supercomputing*. Edited by Susan L. Graham, Marc Snir and Cynthia A. Patterson. Washington, DC: The National Academies Press, 2005. ISBN: 978-0-309-09502-0. DOI: 10.17226/11148. [see pages 6, 7]

[CR07] John Cieslewicz and Kenneth A. Ross. 'Adaptive Aggregation on Chip Multiprocessors'. In: *33rd International Conference on Very Large Data Bases (VLDB)*. 2007, pages 339–350. URL: https://www.vldb.org/conf/2007/papers/research/p339-cieslewicz.pdf, archived at https://web.archive.org/web/20170706094422/https://www.vldb.org/conf/2007/papers/research/p339-cieslewicz.pdf on 6th July 2017. [see page 25]

[CTW16] Yung-Yu Chung, Srikanta Tirthapura and David P. Woodruff. 'A Simple Message-Optimal Algorithm for Random Sampling from a Distributed Stream'. In: *IEEE Transactions on Knowledge and Data Engineering* 28.6 (2016), pages 1356–1368. DOI: 10.1109/TKDE.2016.2518679. [see page 59]

[Cul+93] David E. Culler, Richard M. Karp, David A. Patterson, Abhijit Sahay, Klaus E. Schauser, Eunice E. Santos, Ramesh Subramonian and Thorsten von Eicken. 'LogP: Towards a Realistic Model of Parallel Computation'. In: *4th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*. 1993. DOI: 10.1145/155332.155333. [see page 9]

[Cul+96] David E. Culler, Richard M. Karp, David A. Patterson, Abhijit Sahay, Eunice E. Santos, Klaus E. Schauser, Ramesh Subramonian and Thorsten von Eicken. 'LogP: A Practical Model of Parallel Computation'. In: *Communications of the ACM* 39.11 (1996), pages 78–85: ACM. DOI: 10.1145/240455.240477. [see page 9]

[CW04] Pei Cao and Zhe Wang. 'Efficient top-K query calculation in distributed networks'. In: *23rd ACM Symposium on Principles of Distributed Computing (PODC)*. 2004, pages 206–215. DOI: 10.1145/1011767.1011798. [see page 25]

[Dek+12] Ofer Dekel, Ran Gilad-Bachrach, Ohad Shamir and Lin Xiao. 'Optimal Distributed Online Prediction Using Mini-Batches'. In: *Journal of Machine Learning Research* 13 (2012), pages 165–202. URL: https://jmlr.org/papers/volume13/dekel12a/dekel12a.pdf, archived at https://web.archive.org/web/20200709124054/https://jmlr.org/papers/volume13/dekel12a/dekel12a.pdf on 9th July 2020. [see page 11]

[Dem19] James Demmel. *Communication-Avoiding Algorithms for Linear Algebra, Machine Learning and Beyond*. Talk given at the Argonne Training Program on Extreme-Scale Computing 2019. 2019. DOI: 10.1109/sc.companion.2012.351. URL: https://www.youtube.com/watch?v=iPCBCjgoAbk, archived at https://web.archive.org/web/20191129151750/https://www.youtube.com/watch?v=iPCBCjgoAbk on 29th Nov. 2019. [see pages 6, 7, 10]

[Dev86] Luc Devroye. *Non-Uniform Random Variate Generation*. Springer, 1986. ISBN: 978-1-4613-8645-2. DOI: 10.1007/978-1-4613-8643-8. [see pages 24, 55, 58, 59, 85]

[DG08] Jeffrey Dean and Sanjay Ghemawat. 'MapReduce: simplified data processing on large clusters'. In: *Communications of the ACM* 51.1 (2008), pages 107–113: ACM. DOI: 10.1145/1327452.1327492. [see pages 7, 15, 109]

[Dij74] Edsger W. Dijkstra. 'Self-stabilizing Systems in Spite of Distributed Control'. In: *Communications of the ACM* 17.11 (1974), pages 643–644: ACM. DOI: 10.1145/361179.361202. [see page 117]

[DP92] Narsingh Deo and Sushil K. Prasad. 'Parallel heap: An optimal parallel priority queue'. In: *The Journal of Supercomputing* 6.1 (1992), pages 87–98: Springer. DOI: 10.1007/BF00128644. [see pages 22, 25, 26]

[DT20] Mikhail Drobyshevskiy and Denis Turdakov. 'Random Graph Modeling: A Survey of the Concepts'. In: *ACM Computing Surveys* 52.6 (2020), 131:1–131:36: ACM. DOI: 10.1145/3369782. [see page 104]

[DY14] James Demmel and Kathy Yelick. 'Communication avoiding (CA) and other innovative algorithms'. In: *The Berkeley Par Lab: Progress in the Parallel Computing Landscape* (2014), pages 243–250. URL: https://web.archive.org/web/20160322131757/http://research.microsoft.com/en-us/um/redmond/about/collaboration/Par-lab-book/Chapter-9.pdf. [see page 10]

[Efr15] Pavlos S. Efraimidis. 'Weighted Random Sampling over Data Streams'. In: *Algorithms, Probability, Networks, and Games - Scientific Papers and Essays Dedicated to Paul G. Spirakis on the Occasion of His 60th Birthday*. Volume 9295. Lecture Notes in Computer Science (LNCS). Springer, 2015, pages 183–195. DOI: 10.1007/978-3-319-24024-4_12. [see pages 52, 56, 58, 60]

[Elg+20] Anis Elgabli, Jihong Park, Amrit S. Bedi, Mehdi Bennis and Vaneet Aggarwal. 'GADMM: Fast and Communication Efficient Framework for Distributed Machine Learning'. In: *Journal of Machine Learning Research* 21 (2020), 76:1–76:39. URL: http://jmlr.org/papers/v21/19-718.html, archived at https://web.archive.org/web/20200803142323/http://jmlr.org/papers/v21/19-718.html on 3rd Aug. 2020. [see page 4]

[Ell18] Jan Ellmers. „Dynamische Baumkompression mit Top Trees". In German. Bachelor's Thesis. Department of Computer Science, Karlsruhe Institute of Technology, Germany, Nov. 2018. [see page 154]

[ES06] Pavlos S. Efraimidis and Paul G. Spirakis. 'Weighted random sampling with a reservoir'. In: *Information Processing Letters* 97.5 (2006), pages 181–185: Elsevier. DOI: 10.1016/j.ipl.2005.11.003. [see pages 55, 56, 58, 60, 81, 84]

[ES99] Pavlos S Efraimidis and Paul G Spirakis. *Fast parallel weighted random sampling*. Technical report TR99.04.02. CTI Patras, 1999. URL: http://students.ceid.upatras.gr/~efraimid/tr990602.ps, archived at https://web.archive.org/web/20061013001949/http://students.ceid.upatras.gr/~efraimid/tr990602.ps on 13th Oct. 2006. [see pages 56, 58, 59]

[EW19] Stefan Edelkamp and Armin Weiß. 'BlockQuicksort: Avoiding Branch Mispredictions in Quicksort'. In: *ACM Journal of Experimental Algorithmics* 24.1 (2019), 1.4:1–1.4:22: ACM. DOI: 10.1145/3274660. [see page 4]

[Fan+00] Li Fan, Pei Cao, Jussara M. Almeida and Andrei Z. Broder. 'Summary cache: a scalable wide-area web cache sharing protocol'. In: *IEEE/ACM Transactions on Networking* 8.3 (2000), pages 281–293. DOI: 10.1109/90.851975. [see page 118]

[Fel+10] Jon Feldman, S. Muthukrishnan, Anastasios Sidiropoulos, Clifford Stein and Zoya Svitkina. 'On distributing symmetric streaming computations'. In: *ACM Transactions on Algorithms* 6.4 (2010), 66:1–66:19: ACM. DOI: 10.1145/1824777.1824786. [see page 12]

[Fia+12] David Fiala, Frank Mueller, Christian Engelmann, Rolf Riesen, Kurt B. Ferreira and Ron Brightwell. 'Detection and correction of silent data corruption for large-scale high-performance computing'. In: *2012 SC Conference on High Performance Computing Networking, Storage and Analysis (SC)*. 2012, page 78. DOI: 10.1109/SC.2012.49. [see page 118]

[FLN03] Ronald Fagin, Amnon Lotem and Moni Naor. 'Optimal aggregation algorithms for middleware'. In: *Journal of Computer and System Sciences* 66.4 (2003), pages 614–656: Elsevier. DOI: 10.1016/S0022-0000(03)00026-6. [see page 26]

[FMR62] CT Fan, Mervin E Muller and Ivan Rezucha. 'Development of sampling plans by using sequential (item by item) selection techniques and digital computers'. In: *Journal of the American Statistical Association* 57.298 (1962), pages 387–402: Taylor & Francis Group. DOI: 10.1080/01621459.1962.10480667. [see pages 56, 59]

[FR75] Robert W. Floyd and Ronald L. Rivest. 'Expected Time Bounds for Selection'. In: *Communications of the ACM* 18.3 (1975), pages 165–172: ACM. DOI: 10.1145/360680.360691. [see page 24]

[Fre77] Rusins Freivalds. 'Probabilistic Machines Can Use Less Running Time'. In: *7th IFIP Congress, Information Processing*. 1977, pages 839–842. [see page 116]

[Fri+12] Matteo Frigo, Charles E. Leiserson, Harald Prokop and Sridhar Ramachandran. 'Cache-Oblivious Algorithms'. In: *ACM Transactions on Algorithms* 8.1 (2012), 4:1–4:22: ACM. DOI: 10.1145/2071379.2071383. [see page 9]

[Fun+19] Daniel Funke, Sebastian Lamm, Ulrich Meyer, Manuel Penschuck, Peter Sanders, Christian Schulz, Darren Strash and Moritz von Looz. 'Communication-free massively distributed graph generation'. In: *Journal of Parallel and Distributed Computing* 131 (2019), pages 200–217: Elsevier. DOI: 10.1016/j.jpdc.2019.03.011. [see pages 104, 108]

[FW90] Michael L. Fredman and Dan E. Willard. 'BLASTING through the Information Theoretic Barrier with FUSION TREES'. In: *22nd ACM Symposium on Theory of Computing (STOC)*. May 1990. DOI: 10.1145/100216.100217. [see page 4]

[Gal+09] Mark Galassi, Jim Davies, James Theiler, Brian Gough, Gerard Jungmann, Patrick Alken, Michael Booth, Fabrice Rossi and Rhys Ulerich. *GNU scientific library: reference manual*. 3rd edition. Network Theory, 2009. ISBN: 0954612078. [see page 90]

[GGP10] Rosario Gennaro, Craig Gentry and Bryan Parno. 'Non-interactive Verifiable Computing: Outsourcing Computation to Untrusted Workers'. In: *30th Cryptology Conference (CRYPTO)*. Springer. 2010, pages 465–482. DOI: 10.1007/978-3-642-14623-7_25. [see page 116]

[GKR15] Shafi Goldwasser, Yael Tauman Kalai and Guy N. Rothblum. 'Delegating Computation: Interactive Proofs for Muggles'. In: *Journal of the ACM* 62.4 (2015), 27:1–27:64: ACM. DOI: 10.1145/2699436. [see page 116]

[GMR98] Phillip B. Gibbons, Yossi Matias and Vijaya Ramachandran. 'The Queue-Read Queue-Write Asynchronous PRAM Model'. In: *Theoretical Computer Science* 196.1-2 (1998), pages 3–29: Elsevier. DOI: 10.1016/S0304-3975(97)00193-X. [see page 6]

[Gol+10] Anna Goldenberg, Alice X. Zheng, Stephen E. Fienberg and Edoardo M. Airoldi. 'A Survey of Statistical Network Models'. In: *Foundations and Trends in Machine Learning* 2.2 (2010), pages 129–233: Now Publishers. DOI: 10.1561/2200000005. [see page 104]

[Gop+11] Vinodh Gopal, Jim Guilford, Erdinc Ozturk, Gil Wolrich, Wajdi Feghali, Jim Dixon and Deniz Karakoyunlu. *Fast CRC Computation for iSCSI Polynomial using CRC32 Instruction*. White Paper. Intel Corporation, 2011. [see page 130]

[Got+83] Allan Gottlieb, Ralph Grishman, Clyde P. Kruskal, Kevin P. McAuliffe, Larry Rudolph and Marc Snir. 'The NYU Ultracomputer–Designing an MIMD Shared Memory Parallel Computer'. In: *IEEE Transactions on Computers* 32.2 (1983), pages 175–189: IEEE. DOI: 10.1109/TC.1983.1676201. [see page 59]

[GSZ11] Michael T. Goodrich, Nodari Sitchinava and Qin Zhang. 'Sorting, Searching, and Simulation in the MapReduce Framework'. In: *Algorithms and Computation - 22nd International Symposium, ISAAC 2011*. Springer. 2011, pages 374–383. DOI: 10.1007/978-3-642-25591-5_39. [see page 8]

[GUW09] Hector Garcia-Molina, Jeffrey D. Ullman and Jennifer Widom. *Database systems - the complete book (2. ed.)* Pearson Education, 2009. ISBN: 978-0-13-187325-4. [see page 129]

[HA84] Kuang-Hua Huang and Jacob A. Abraham. 'Algorithm-Based Fault Tolerance for Matrix Operations'. In: *IEEE Transactions on Computers* 33.6 (1984), pages 518–528: IEEE. DOI: 10.1109/TC.1984.1676475. [see page 117]

[Háj64] Jaroslav Hájek. 'Asymptotic theory of rejective sampling with varying probabilities from a finite population'. In: *The Annals of Mathematical Statistics* (1964), pages 1491–1523: JSTOR. URL: http://www.jstor.org/stable/2238287. [see page 59]

[Har+20] Siva Kumar Sastry Hari, Paolo Rech, Timothy Tsai, Mark Stephenson, Arslan Zulfiqar, Michael B. Sullivan, Philip P. Shirvani, Paul Racunas, Joel S. Emer and Stephen W. Keckler. 'Estimating Silent Data Corruption Rates Using a Two-Level Model'. In: *Computing Research Repository (CoRR)* (Apr. 2020). arXiv: 2005.01445 [cs.DC]. [see page 111]

[HH43] Morris H Hansen and William N Hurwitz. 'On the theory of sampling from finite populations'. In: *The Annals of Mathematical Statistics* 14.4 (1943), pages 333–362: Institute of Mathematical Statistics. URL: http://www.jstor.org/stable/2235923. [see pages 58, 70]

[HMM93] Torben Hagerup, Kurt Mehlhorn and J. Ian Munro. 'Maintaining Discrete Probability Distributions Optimally'. In: *20th International Colloquium on Automata, Languages, and Programming (ICALP)*. Springer. 1993, pages 253–264. DOI: 10.1007/3-540-56939-1_77. [see page 58]

[Hoa61] C. A. R. Hoare. 'Algorithm 65: find'. In: *Communications of the ACM* 4.7 (1961), pages 321–322: ACM. DOI: 10.1145/366622.366647. [see pages 24, 29]

[Hoe63] Wassily Hoeffding. 'Probability Inequalities for Sums of Bounded Random Variables'. In: *Journal of the American Statistical Association* 58.301 (1963), pages 13–30: Taylor & Francis. DOI: 10.1080/01621459.1963.10500830. [see page 14]

[HP11] John L Hennessy and David A Patterson. *Computer architecture: a quantitative approach. Appendix F: Interconnection Networks*. 5th edition. Elsevier, 2011. ISBN: 9780123838728. URL: https://booksite.elsevier.com/9780123838728/references/appendix_f.pdf, archived at https://web.archive.org/web/20170828233131/https://booksite.elsevier.com/9780123838728/references/appendix_f.pdf on 28th Aug. 2017. [see page 2]

[HR15a] Lorenz Hübschle-Schneider and Rajeev Raman. 'Tree Compression with Top Trees Revisited'. In: *14th International Symposium on Experimental Algorithms (SEA)*. June 2015, pages 15–27. DOI: 10.1007/978-3-319-20086-6_2. [see page 153]

[HR15b] Lorenz Hübschle-Schneider and Rajeev Raman. 'Tree Compression with Top Trees Revisited'. In: *Computing Research Repository (CoRR)* (June 2015). arXiv: 1506.04499 [cs.DS]. [see page 154]

[HR62] Herman Otto Hartley and J.N.K. Rao. 'Sampling with unequal probabilities and without replacement'. In: *The Annals of Mathematical Statistics* 33.2 (1962), pages 350–374: Institute of Mathematical Statistics. DOI: 10.1214/aoms/1177704564. [see page 58]

[HS16] Lorenz Hübschle-Schneider and Peter Sanders. 'Communication Efficient Algorithms for Top-k Selection Problems'. In: *30th IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. May 2016, pages 659–668. DOI: 10.1109/IPDPS.2016.45. [see pages 21, 153]

[HS17] Lorenz Hübschle-Schneider and Peter Sanders. 'Communication Efficient Checking of Big Data Operations'. In: *Computing Research Repository (CoRR)* (Oct. 2017). arXiv: 1710.08255 [cs.DS]. [see pages 112, 154]

[HS18] Lorenz Hübschle-Schneider and Peter Sanders. 'Communication Efficient Checking of Big Data Operations'. In: *32nd IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. May 2018, pages 650–659. DOI: 10.1109/IPDPS.2018.00074. [see pages 112, 153]

[HS19a] Lorenz Hübschle-Schneider and Peter Sanders. 'Communication-Efficient (Weighted) Reservoir Sampling from Fully Distributed Data Streams'. In: *Computing Research Repository (CoRR)* (Oct. 2019). arXiv: 1910.11069 [cs.DS]. [see pages 51, 154]

[HS19b] Lorenz Hübschle-Schneider and Peter Sanders. 'Linear Work Generation of R-MAT Graphs'. In: *Computing Research Repository (CoRR)* (May 2019). arXiv: 1905.03525 [cs.DS]. [see pages 105, 154]

[HS19c] Lorenz Hübschle-Schneider and Peter Sanders. 'Parallel Weighted Random Sampling'. In: *Computing Research Repository (CoRR)* (Mar. 2019). arXiv: 1903.00227 [cs.DS]. [see pages 51, 154]

[HS19d] Lorenz Hübschle-Schneider and Peter Sanders. 'Parallel Weighted Random Sampling'. In: *27th European Symposium on Algorithms (ESA)*. 2019, 59:1–59:24. DOI: 10.4230/LIPIcs.ESA.2019.59. [see pages 51–53, 73, 90, 151, 153]

[HS20a] Lorenz Hübschle-Schneider and Peter Sanders. 'Brief Announcement: Communication-Efficient Weighted Reservoir Sampling from Fully Distributed Data Streams'. In: *32nd ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. 2020, pages 543–545. DOI: 10.1145/3350755.3400287. [see pages 51, 153]

[HS20b] Lorenz Hübschle-Schneider and Peter Sanders. 'Linear Work Generation of R-MAT Graphs'. In: *Network Science* 8.4 (2020), pages 543–550: Cambridge University Press. DOI: 10.1017/nws.2020.21. [see pages 51, 104, 153]

[HSM15] Lorenz Hübschle-Schneider, Peter Sanders and Ingo Müller. 'Communication Efficient Algorithms for Top-k Selection Problems'. In: *Computing Research Repository (CoRR)* (Feb. 2015). arXiv: 1502.03942 [cs.DS]. [see pages 21, 22, 68, 154]

[HSS12] Andy A. Hwang, Ioan A. Stefanovici and Bianca Schroeder. 'Cosmic rays don't strike twice: understanding the nature of DRAM errors and the implications for system design'. In: *ACM SIGPLAN Not.* 47.4 (2012), pages 111–122: ACM . [see page 118]

[HSZ08] Christian Henke, Carsten Schmoll and Tanja Zseby. 'Empirical evaluation of hash functions for multipoint measurements'. In: *Computer Communication Review* 38.3 (2008), pages 39–50: ACM SIGCOMM. DOI: 10.1145/1384609.1384614. [see page 136]

[HT52] D. G. Horvitz and D. J. Thompson. 'A Generalization of Sampling Without Replacement from a Finite Universe'. In: *Journal of the American Statistical Association* 47.260 (1952), pages 663–685: Taylor & Francis. DOI: 10.1080/01621459.1952.10483446. [see page 58]

[Hüb13] Lorenz Hübschle-Schneider. 'Speed-Consumption Trade-Off for Electric Vehicle Routing'. Bachelor's Thesis. Department of Computer Science, Karlsruhe Institute of Technology, Germany, June 2013. [see page 154]

[Hüb14] Lorenz Hübschle-Schneider. 'Compression Methods for Labelled Trees'. Master's Thesis. Department of Computer Science, University of Leicester, United Kingdom, Sept. 2014. [see page 154]

[HYZ19] Zengfeng Huang, Ke Yi and Qin Zhang. 'Randomized Algorithms for Tracking Distributed Count, Frequencies, and Ranks'. In: *Algorithmica* 81.6 (2019), pages 2222–2243: Springer. DOI: 10.1007/s00453-018-00531-y. [see pages 23, 25, 26]

[Int19] Intel. *Intel Math Kernel Library 2019*. Intel. 2019. URL: https://software.intel.com/en-us/mkl-reference-manual-for-c. [see page 89]

[JáJ11] Joseph F. JáJá. 'PRAM (Parallel Random Access Machines)'. In: *Encyclopedia of Parallel Computing*. Springer US, 2011, pages 1608–1615. DOI: 10.1007/978-0-387-09766-4_23. [see pages 4, 5]

[JáJ92] Joseph JáJá. *An Introduction to Parallel Algorithms*. Addison-Wesley, 1992. ISBN: 0-201-54856-9. [see page 5]

[Jay+19] Rajesh Jayaram, Gokarna Sharma, Srikanta Tirthapura and David P. Woodruff. 'Weighted Reservoir Sampling from Distributed Streams'. In: *38th ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems (PODS)*. 2019, pages 218–235. DOI: 10.1145/3294052.3319696. [see pages 60, 86]

[JW94] Ben H. H. Juurlink and Harry A. G. Wijshoff. 'The Parallel Hierarchical Memory Model'. In: *4th Scandinavian Workshop on Algorithm Theory (SWAT)*. Springer. 1994, pages 240–251. DOI: 10.1007/3-540-58218-5_22. [see page 10]

[Kar+18] Jeyhun Karimov, Tilmann Rabl, Asterios Katsifodimos, Roman Samarev, Henri Heiskanen and Volker Markl. 'Benchmarking Distributed Stream Data Processing Systems'. In: *34th IEEE International Conference on Data Engineering (ICDE)*. 2018, pages 1507–1518. DOI: 10.1109/ICDE.2018.00169. [see page 15]

[Kar+93] Richard M. Karp, Abhijit Sahay, Eunice E. Santos and Klaus E. Schauser. 'Optimal Broadcast and Summation in the LogP Model'. In: *5th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. 1993, pages 142–153. DOI: 10.1145/165231.165250. [see page 9]

[Kat88] Howard P. Katseff. 'Incomplete Hypercubes'. In: *IEEE Transactions on Computers* 37.5 (1988), pages 604–608: IEEE. DOI: 10.1109/12.4611. [see page 28]

[Kim+08] John Kim, William J. Dally, Steve Scott and Dennis Abts. 'Technology-Driven, Highly-Scalable Dragonfly Topology'. In: *35th International Symposium on Computer Architecture (ISCA)*. June 2008, pages 77–88. DOI: 10.1109/ISCA.2008.19. [see page 2]

[Kim+14] Yoongu Kim, Ross Daly, Jeremie S. Kim, Chris Fallin, Ji-Hye Lee, Donghyuk Lee, Chris Wilkerson, Konrad Lai and Onur Mutlu. 'Flipping bits in memory without accessing them: An experimental study of DRAM disturbance errors'. In: *41st International Symposium on Computer Architecture (ISCA)*. 2014, pages 361–372. DOI: 10.1109/ISCA.2014.6853210. [see page 111]

[KIT20] KIT HPC wiki. *ForHLR II Hardware and Architecture*. 2020. URL: https://wiki.scc.kit.edu/hpc/index.php/ForHLR_-_Hardware_and_Architecture, archived at https://web.archive.org/web/20200505151256/https://wiki.scc.kit.edu/hpc/index.php/ForHLR_-_Hardware_and_Architecture on 5th May 2020. [see page 87]

[KK06] Amos Korman and Shay Kutten. 'On Distributed Verification'. In: *8th International Conference on Distributed Computing and Networking (ICDCN)*. Springer. 2006, pages 100–114. DOI: 10.1007/11947950_12. [see page 117]

[KKP10] Amos Korman, Shay Kutten and David Peleg. 'Proof labeling schemes'. In: *Distributed Computing* 22.4 (2010), pages 215–233: Springer. DOI: 10.1007/s00446-010-0095-3. [see page 117]

[KM16] Rob Kitchin and Gavin McArdle. 'What makes Big Data, Big Data? Exploring the ontological characteristics of 26 datasets'. In: *Big Data & Society* 3.1 (2016). SAGE Publications. DOI: 10.1177/2053951716631130. [see page 15]

[Kri+10] Dmitri Krioukov, Fragkiskos Papadopoulos, Maksim Kitsak, Amin Vahdat and Marián Boguñá. 'Hyperbolic geometry of complex networks'. In: *Physical Review E* 82 (3 Sept. 2010), page 036106: American Physical Society. DOI: 10.1103/PhysRevE.82.036106. [see page 104]

[KS06] Kanela Kaligosi and Peter Sanders. 'How Branch Mispredictions Affect Quicksort'. In: *14th European Symposium on Algorithms (ESA)*. Springer. Sept. 2006, pages 780–791. DOI: 10.1007/11841036_69. [see page 4]

[KSV10] Howard J. Karloff, Siddharth Suri and Sergei Vassilvitskii. 'A Model of Computation for MapReduce'. In: *21st ACM-SIAM Symposium on Discrete Algorithms (SODA)*. 2010, pages 938–948. DOI: 10.1137/1.9781611973075.76. [see page 8]

[Kum+94] Vipin Kumar, Ananth Grama, Anshul Gupta and George Karypis. *Introduction to Parallel Computing. Design and Analysis of Algorithms*. Benjamin/Cummings, 1994. [see pages 13, 74]

[KZ93] Richard M. Karp and Yanjun Zhang. 'Randomized Parallel Algorithms for Backtrack Search and Branch-and-Bound Computation'. In: *Journal of the ACM* 40.3 (1993), pages 765–789: ACM. DOI: 10.1145/174130.174145. [see pages 21, 23, 25, 35, 36]

[Lan14] Kevin J. Lang. 'Practical Algorithms for Generating a Random Ordering of the Elements of a Weighted Set'. In: *Theory of Computing Systems* 54.4 (2014), pages 659–688: Springer. DOI: 10.1007/s00224-013-9496-6. [see page 59]

[Lei85] Charles E. Leiserson. 'Fat-Trees: Universal Networks for Hardware-Efficient Supercomputing'. In: *IEEE Transactions on Computers* 34.10 (1985), pages 892–901: IEEE. DOI: 10.1109/TC.1985.6312192. [see page 2]

[Lei92] F. Thomson Leighton. *Introduction to Parallel Algorithms and Architectures*. Morgan Kaufmann, 1992, pages 389–783. ISBN: 978-1-4832-0772-8. DOI: 10.1016/C2013-0-08299-0. [see page 13]

[Les+10] Jure Leskovec, Deepayan Chakrabarti, Jon M. Kleinberg, Christos Faloutsos and Zoubin Ghahramani. 'Kronecker Graphs: An Approach to Modeling Networks'. In: *Journal of Machine Learning Research* 11 (2010), pages 985–1042. URL: https://jmlr.org/papers/volume11/leskovec10a/leskovec10a.pdf, archived at https://web.archive.org/web/20200709124144/https://jmlr.org/papers/volume11/leskovec10a/leskovec10a.pdf on 9th July 2020. [see page 104]

[Li+07] Feifei Li, Ke Yi, Marios Hadjieleftheriou and George Kollios. 'Proof-Infused Streams: Enabling Authentication of Sliding Window Queries On Streams'. In: *33rd International Conference on Very Large Data Bases (VLDB)*. 2007, pages 147–158. URL: http://www.vldb.org/conf/2007/papers/research/p147-li.pdf, archived at https://web.archive.org/web/20170925162307/http://www.vldb.org/conf/2007/papers/research/p147-li.pdf on 25th Sept. 2017. [see page 116]

[Li+13] Yinan Li, Ippokratis Pandis, René Müller, Vijayshankar Raman and Guy M. Lohman. 'NUMA-aware algorithms: the case of data shuffling'. In: *6th Biennial Conference on Innovative Data Systems Research (CIDR)*. 2013. URL: http://cidrdb.org/cidr2013/Papers/CIDR13_Paper121.pdf, archived at https://web.archive.org/web/20170829113623/http://cidrdb.org/cidr2013/Papers/CIDR13_Paper121.pdf on 29th Aug. 2017. [see page 25]

[Li94] Kim-Hung Li. 'Reservoir-Sampling Algorithms of Time Complexity $\mathcal{O}(n(1 + \log(N/n)))$'. In: *ACM Transactions on Mathematical Software (TOMS)* 20.4 (Dec. 1994), pages 481–493: ACM. DOI: 10.1145/198429.198435. [see page 59]

[Lip89] Richard J. Lipton. 'New Directions In Testing'. In: *Workshop: Distributed Computing And Cryptography*. 1989, pages 191–202. DOI: 10.1090/dimacs/002/13. URL: https://books.google.com/books?id=LuI4RNSg1ioC&pg=PA191. [see page 124]

[LL99] Anthony LaMarca and Richard E. Ladner. 'The Influence of Caches on the Performance of Sorting'. In: *Journal of Algorithms* 31.1 (1999), pages 66–104. DOI: 10.1006/jagm.1998.0985. [see page 5]

[Lor20] Collin Lorbeer. „Kommunikationseffizientes Branch-and-Bound für das Rucksackproblem". In German. Bachelor's Thesis. Department of Computer Science, Karlsruhe Institute of Technology, Germany, Apr. 2020. [see page 154]

[Man+05] Amit Manjhi, Vladislav Shkapenyuk, Kedar Dhamdhere and Christopher Olston. 'Finding (Recently) Frequent Items in Distributed Data Streams'. In: *21st IEEE International Conference on Data Engineering (ICDE)*. 2005, pages 767–778. DOI: 10.1109/ICDE.2005.68. [see page 25]

[McC+11] Ross M. McConnell, Kurt Mehlhorn, Stefan Näher and Pascal Schweitzer. 'Certifying algorithms'. In: *Computer Science Review* 5.2 (2011), pages 119–161: Elsevier. DOI: 10.1016/j.cosrev.2010.09.009. [see page 116]

[McK17] Paul E. McKenney. 'Is Parallel Programming Hard, And, If So, What Can You Do About It? (v2017.01.02a)'. In: *Computing Research Repository (CoRR)* (Jan. 2017). arXiv: 1701.00854 [cs.DC]. [see page 1]

[MHS12] Milo M. K. Martin, Mark D. Hill and Daniel J. Sorin. 'Why on-chip cache coherence is here to stay'. In: *Communications of the ACM* 55.7 (2012), pages 78–89: ACM. DOI: 10.1145/2209249.2209269. [see page 2]

[MN98] Makoto Matsumoto and Takuji Nishimura. 'Mersenne Twister: A 623-Dimensionally Equidistributed Uniform Pseudo-Random Number Generator'. In: *ACM Transactions on Modeling and Computer Simulation (TOMACS)* 8.1 (1998), pages 3–30. DOI: 10.1145/272991.272995. [see pages 89, 130]

[Moo65] Gordon E. Moore. 'Cramming more components onto integrated circuits'. In: *Electronics* 38.8 (Apr. 1965), pages 114–117. Reprinted in *IEEE Solid-State Circuits Society Newsletter* 11(3) (2006). DOI: 10.1109/N-SSC.2006.4785860. [see page 1]

[Moo75] Gordon E. Moore. 'Progress in digital integrated electronics'. In: *International Electron Devices Meeting*. 1975, pages 11–13. Reprinted in *IEEE Solid-State Circuits Society Newsletter* 11(3) (2006). DOI: 10.1109/N-SSC.2006.4804410. [see page 1]

[MR95] Rajeev Motwani and Prabhakar Raghavan. *Randomized Algorithms*. Cambridge University Press, 1995. ISBN: 0-521-47465-5. DOI: 10.1017/cbo9780511814075. [see page 51]

[MTW04] George Marsaglia, Wai Wan Tsang and Jingbo Wang. 'Fast generation of discrete random variables'. In: *Journal of Statistical Software* 11.3 (2004). Foundation for Open Access Statistics. DOI: 10.18637/jss.v011.i03. [see page 58]

[MTW05] Sebastian Michel, Peter Triantafillou and Gerhard Weikum. 'KLEE: A Framework for Distributed Top-k Query Algorithms'. In: *31st International Conference on Very Large Data Bases (VLDB)*. 2005, pages 637–648. URL: https://www.vldb.org/archives/website/2005/program/paper/thu/p637-michel.pdf, archived at https://web.archive.org/web/20200428150723/https://www.vldb.org/archives/website/2005/program/paper/thu/p637-michel.pdf on 28th Apr. 2020. [see page 25]

[MU05] Michael Mitzenmacher and Eli Upfal. *Probability and Computing: Randomized Algorithms and Probabilistic Analysis*. Cambridge University Press, 2005. ISBN: 978-0-521-83540-4. DOI: 10.1017/CBO9780511813603. [see page 14]

[MU17] Michael Mitzenmacher and Eli Upfal. *Probability and Computing: Randomization and Probabilistic Techniques in Algorithms and Data Analysis*. 2nd. Cambridge University Press, 2017. ISBN: 978-1-107-15488-9. [see page 28]

[Mül+15] Ingo Müller, Peter Sanders, Arnaud Lacurie, Wolfgang Lehner and Franz Färber. 'Cache-Efficient Aggregation: Hashing Is Sorting'. In: *ACM SIGMOD International Conference on Management of Data*. 2015, pages 1123–1136. DOI: 10.1145/2723372.2747644. [see page 25]

[Mül16] Kirill Müller. 'Accelerating weighted random sampling without replacement'. In: *Arbeitsberichte Verkehrs-und Raumplanung* 1141 (2016). IVT, ETH Zürich . [see page 59]

[Mur+10] Richard C Murphy, Kyle B Wheeler, Brian W Barrett and James A Ang. 'Introducing the Graph 500'. In: *Cray User's Group* 19 (2010), pages 45–74 . [see page 104]

[Mut05] S. Muthukrishnan. 'Data Streams: Algorithms and Applications'. In: *Foundations and Trends in Theoretical Computer Science* 1.2 (2005), pages 117–236. DOI: 10.1561/0400000002. [see page 10]

[MVN03] Yossi Matias, Jeffrey Scott Vitter and Wen-Chun Ni. 'Dynamic Generation of Discrete Random Variates'. In: *Theory of Computing Systems* 36.4 (2003), pages 329–358: Springer. DOI: 10.1007/s00224-003-1078-6. [see pages 58, 59]

[MX11] Mohammad Mahdian and Ying Xu. 'Stochastic kronecker graphs'. In: *Random Structures & Algorithms* 38.4 (2011), pages 453–466. DOI: 10.1002/rsa.20335. [see page 104]

[Mye04] Glenford J. Myers. *The art of software testing (2. ed.)* Wiley, 2004. ISBN: 978-0-471-46912-4. [see page 116]

[NS95] Moni Naor and Larry J. Stockmeyer. 'What Can be Computed Locally?' In: *SIAM Journal on Computing* 24.6 (1995), pages 1259–1277: SIAM. DOI: 10.1137/S0097539793254571. [see page 117]

[NV13] Suman Nath and Ramarathnam Venkatesan. 'Publicly verifiable grouped aggregation queries on outsourced data streams'. In: *29th IEEE International Conference on Data Engineering (ICDE)*. 2013, pages 517–528. DOI: 10.1109/ICDE.2013.6544852. [see page 116]

[OR95] Frank Olken and Doron Rotem. 'Random sampling from databases: a survey'. In: *Statistics and Computing* 5.1 (1995), pages 25–42: Springer. DOI: 10.1007/BF00140664. [see pages 51, 55, 58]

[Pap+13] Stavros Papadopoulos, Graham Cormode, Antonios Deligiannakis and Minos N. Garofalakis. 'Lightweight authentication of linear algebraic queries on data streams'. In: *ACM SIGMOD International Conference on Management of Data*. 2013, pages 881–892. DOI: 10.1145/2463676.2465281. [see page 116]

[Pap+14] Stavros Papadopoulos, Graham Cormode, Antonios Deligiannakis and Minos N. Garofalakis. 'Lightweight Query Authentication on Streams'. In: *ACM Transactions on Database Systems (TODS)* 39.4 (2014), 30:1–30:45. DOI: 10.1145/2656336. [see page 116]

[Par+16] Bryan Parno, Jon Howell, Craig Gentry and Mariana Raykova. 'Pinocchio: nearly practical verifiable computation'. In: *Communications of the ACM* 59.2 (2016), pages 103–112: ACM. DOI: 10.1145/2856449. [see page 116]

[Pel00] David Peleg. *Distributed computing: a locality-sensitive approach*. SIAM, 2000. ISBN: 978-0-89871-464-7. DOI: 10.1137/1.9780898719772. [see page 9]

[Pen+20] Manuel Penschuck, Ulrik Brandes, Michael Hamann, Sebastian Lamm, Ulrich Meyer, Ilya Safro, Peter Sanders and Christian Schulz. 'Recent Advances in Scalable Network Generation'. In: *Computing Research Repository (CoRR)* (Mar. 2020). arXiv: 2003.00736 [cs.DS]. [see page 104]

[PGM13] James S. Plank, Kevin M. Greenan and Ethan L. Miller. 'Screaming fast Galois field arithmetic using intel SIMD instructions'. In: *11th USENIX conference on File and Storage Technologies (FAST)*. 2013, pages 299–306. [see page 124]

[Pla89] C. Greg Plaxton. 'On the Network Complexity of Selection'. In: *30th Symposium on Foundations of Computer Science (FOCS)*. 1989, pages 396–401. DOI: 10.1109/SFCS.1989.63509. [see page 24]

[PT12] Mihai Patrascu and Mikkel Thorup. 'The Power of Simple Tabulation Hashing'. In: *Journal of the ACM* 59.3 (2012), 14:1–14:50: ACM. DOI: 10.1145/2220357.2220361. [see page 130]

[R C20] R Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing. Vienna, Austria, 2020. URL: https://www.R-project.org. [see page 90]

[Raj90] Sanguthevar Rajasekaran. 'Randomized Parallel Selection'. In: *10th Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS)*. Springer. 1990, pages 215–224. DOI: 10.1007/3-540-53487-3_46. [see page 24]

[Ran91] Abhiram G. Ranade. 'How to Emulate Shared Memory'. In: *Journal of Computer and System Sciences* 42.3 (1991), pages 307–326: Elsevier. DOI: 10.1016/0022-0000(91)90005-P. [see page 67]

[Rei85] Rüdiger Reischuk. 'Probabilistic Parallel Algorithms for Sorting and Selection'. In: *SIAM Journal on Computing* 14.2 (1985), pages 396–409: SIAM. DOI: 10.1137/0214030. [see page 24]

[RHC62] J. N. K. Rao, Herman Otto Hartley and W. G. Cochran. 'On a Simple Procedure of Unequal Probability Sampling Without Replacement'. In: *Journal of the Royal Statistical Society: Series B (Methodological)* 24.2 (1962), pages 482–491. DOI: 10.1111/j.2517-6161.1962.tb00475.x. [see page 58]

[RR89] Sanguthevar Rajasekaran and John H. Reif. 'Optimal and Sublogarithmic Time Randomized Parallel Sorting Algorithms'. In: *SIAM Journal on Computing* 18.3 (1989), pages 594–607: SIAM. DOI: 10.1137/0218041. [see pages 55, 80]

[RS98] Martin Raab and Angelika Steger. '"Balls into Bins" - A Simple and Tight Analysis'. In: *2nd International Workshop on Randomization and Approximation Techniques in Computer Science (RANDOM)*. Springer. 1998, pages 159–170. DOI: 10.1007/3-540-49543-6_13. [see pages 69, 74]

[Rub81] Reuven Y. Rubinstein. *Simulation and the Monte Carlo method*. Wiley series in probability and mathematical statistics. Wiley, 1981. ISBN: 0471089176. [see page 58]

[San+16] Peter Sanders, Sebastian Lamm, Lorenz Hübschle-Schneider, Emanuel Schrade and Carsten Dachsbacher. 'Efficient Random Sampling – Parallel, Vectorized, Cache-Efficient, and Online'. In: *Computing Research Repository (CoRR)* (Oct. 2016). arXiv: 1610.05141 [cs.DS]. [see page 154]

[San+18] Peter Sanders, Sebastian Lamm, Lorenz Hübschle-Schneider, Emanuel Schrade and Carsten Dachsbacher. 'Efficient Parallel Random Sampling – Vectorized, Cache-Efficient, and Online'. In: *ACM Transactions on Mathematical Software (TOMS)* 44.3 (Apr. 2018), 29:1–29:14: ACM. DOI: 10.1145/3157734. [see pages 57, 59, 70, 73, 75, 153]

[San+19] Peter Sanders, Kurt Mehlhorn, Martin Dietzfelbinger and Roman Dementiev. *Sequential and Parallel Algorithms and Data Structures - The Basic Toolbox*. Springer, 2019. ISBN: 978-3-030-25208-3. DOI: 10.1007/978-3-030-25209-0. [see pages 6, 13, 14, 29, 68, 69, 124, 128]

[San08] Peter Sanders. *Course on Parallel Algorithms, Lecture slides, 126–128*. 2008. URL: https://algo2.iti.kit.edu/sanders/courses/paralg08/vorlesung1.pdf, archived at https://web.archive.org/web/20190819145046/https://algo2.iti.kit.edu/sanders/courses/paralg08/vorlesung1.pdf on 19th Aug. 2019. [see page 29]

[San09] Peter Sanders. 'Algorithm Engineering – An Attempt at a Definition'. In: *Efficient Algorithms, Essays Dedicated to Kurt Mehlhorn on the Occasion of His 60th Birthday*. Springer. 2009, pages 321–340. DOI: 10.1007/978-3-642-03456-5_22. [see page 16]

[San10] Peter Sanders. 'Algorithm Engineering - An Attempt at a Definition Using Sorting as an Example'. In: *12th Workshop on Algorithm Engineering and Experiments (ALENEX)*. 2010, pages 55–61. DOI: 10.1137/1.9781611972900.6. [see page 16]

[San20] Peter Sanders. 'Connecting MapReduce Computations to Realistic Machine Models'. In: *Computing Research Repository (CoRR)* (2020). arXiv: 2002.07553 [cs.DS]. [see page 9]

[San96] Peter Sanders. 'On the Competitive Analysis of Randomized Static Load Balancing'. In: *First Workshop on Randomized Parallel Algorithms*. 1996. URL: https://algo2.iti.kit.edu/sanders/papers/rand96.pdf, archived at https://web.archive.org/web/20200505142152/https://algo2.iti.kit.edu/sanders/papers/rand96.pdf on 5th May 2020. [see pages 69, 81]

[San98] Peter Sanders. 'Randomized Priority Queues for Fast Parallel Access'. In: *Journal of Parallel and Distributed Computing, Special Issue on Parallel and Distributed Data Structures* 49.1 (1998), pages 86–97: Elsevier. Extended version: https://algo2.iti.kit.edu/sanders/papers/1997-7.ps.gz, archived at https://web.archive.org/web/20120714063036/https://algo2.iti.kit.edu/sanders/papers/1997-7.ps.gz on 14th July 2012. DOI: 10.1006/jpdc.1998.1429. [see pages 21–28, 35, 36]

[Sch93] Marco Schneider. 'Self-Stabilization'. In: *ACM Computing Surveys* 25.1 (1993), pages 45–67. DOI: 10.1145/151254.151256. [see page 117]

[Set+12] Srinath T. V. Setty, Victor Vu, Nikhil Panpalia, Benjamin Braun, Andrew J. Blumberg and Michael Walfish. 'Taking Proof-Based Verified Computation a Few Steps Closer to Practicality'. In: *21th USENIX Security Symposium*. 2012, pages 253–268. URL: https://www.usenix.org/conference/usenixsecurity12/technical-sessions/presentation/setty, archived at https://web.archive.org/web/20200904143824/https://www.usenix.org/conference/usenixsecurity12/technical-sessions/presentation/setty on 4th Sept. 2020. [see page 116]

[Shu17] Julian Shun. 'Improved Parallel Construction of Wavelet Trees and Rank/Select Structures'. In: *Data Compression Conference (DCC)*. 2017, pages 92–101. DOI: 10.1109/DCC.2017.85. [see page 67]

[SM13] Peterand Sebastian Schlag Sanders and Ingo Müller. 'Communication efficient algorithms for fundamental big data problems'. In: *IEEE International Conference on Big Data*. 2013, pages 15–23. DOI: 10.1109/BigData.2013.6691549. [see pages 37, 43]

[SPW11] Bianca Schroeder, Eduardo Pinheiro and Wolf-Dietrich Weber. 'DRAM errors in the wild: a large-scale field study'. In: *Communications of the ACM* 54.2 (2011), pages 100–107: ACM. DOI: 10.1145/1897816.1897844. [see page 111]

[SS16] Peter Sanders and Christian Schulz. 'Scalable generation of scale-free graphs'. In: *Information Processing Letters* 116.7 (2016), pages 489–491: Elsevier. DOI: 10.1016/j.ipl.2016.02.004. [see page 104]

[SSM16] Christian L. Staudt, Aleksejs Sazonovs and Henning Meyerhenke. 'NetworKit: A tool suite for large-scale complex network analysis'. In: *Network Science* 4.4 (2016), pages 508–530: Cambridge University Press. Code archived at https://web.archive.org/web/20200925083008/https://codeload.github.com/networkit/networkit/tar.gz/7.1 on 25th Sept. 2020. DOI: 10.1017/nws.2016.20. [see page 106]

[SSP07] Johannes Singler, Peter Sanders and Felix Putze. 'MCSTL: The Multi-core Standard Template Library'. In: *13th International Euro-Par Conference*. Springer. 2007, pages 682–694. DOI: 10.1007/978-3-540-74466-5_72. [see page 29]

[SST09] Peter Sanders, Jochen Speck and Jesper Larsson Träff. 'Two-tree algorithms for full bandwidth broadcast, reduction and scan'. In: *Parallel Computing* 35.12 (2009), pages 581–594: Elsevier. DOI: 10.1016/j.parco.2009.09.001. [see page 13]

[Sun77] Alan B. Sunter. 'List sequential sampling with equal or unequal probabilities without replacement'. In: *Journal of the Royal Statistical Society: Series C (Applied Statistics)* 26.3 (1977), pages 261–268: Wiley Online Library. DOI: 10.2307/2346966. [see page 56]

[Sur+17] Ananda Theertha Suresh, Felix X. Yu, Sanjiv Kumar and H. Brendan McMahan. 'Distributed Mean Estimation with Limited Communication'. In: *34th International Conference on Machine Learning (ICML)*. 2017, pages 3329–3337. URL: http://proceedings.mlr.press/v70/suresh17a/suresh17a.pdf, archived at https://web.archive.org/web/20200731163518/http://proceedings.mlr.press/v70/suresh17a/suresh17a.pdf on 31st July 2020. [see page 4]

[Sut05] Herb Sutter. 'The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software'. In: *Dr. Dobb's Journal* 30 (3 Mar. 2005), pages 202–210. URL: http://www.gotw.ca/publications/concurrency-ddj.htm, archived at https://web.archive.org/web/20191019043343/http://www.gotw.ca/publications/concurrency-ddj.htm on 19th Oct. 2019. [see pages 1, 2]

[Til06] Yves Tillé. *Sampling algorithms*. Springer Series in Statistics. Springer, 2006. DOI: 10.1007/0-387-34240-0. [see pages 58, 59]

[Tos+14] Ankit Toshniwal, Siddarth Taneja, Amit Shukla, Karthikeyan Ramasamy, Jignesh M. Patel, Sanjeev Kulkarni, Jason Jackson, Krishna Gade, Maosong Fu, Jake Donham, Nikunj Bhagat, Sailesh Mittal and Dmitriy V. Ryaboy. 'Storm@twitter'. In: *ACM SIGMOD International Conference on Management of Data*. 2014, pages 147–156. DOI: 10.1145/2588555.2595641. [see page 15]

[Tsa+10] Meng-Tsung Tsai, Da-Wei Wang, Churn-Jung Liau and Tsan-sheng Hsu. 'Heterogeneous Subset Sampling'. In: *16th International Conference on Computing and Combinatorics (COCOON)*. Springer. 2010, pages 500–509. DOI: 10.1007/978-3-642-14031-0_53. [see page 59]

[TT19] Kanat Tangwongsan and Srikanta Tirthapura. 'Parallel Streaming Random Sampling'. In: *25th International Euro-Par Conference*. Springer. 2019, pages 451–465. DOI: 10.1007/978-3-030-29400-7_32. [see pages 10, 59]

[TW11] Srikanta Tirthapura and David P. Woodruff. 'Optimal Random Sampling from Distributed Streams Revisited'. In: *25th International Symposium on Distributed Computing (DISC)*. Springer. 2011, pages 283–297. DOI: 10.1007/978-3-642-24100-0_27. [see page 59]

[Ull12] Jeffrey D. Ullman. 'Designing good MapReduce algorithms'. In: *ACM Crossroads* 19.1 (2012), pages 30–34. DOI: 10.1145/2331042.2331053. [see page 8]

[US 20] U.S. Census Bureau. *Annual Survey of Manufactures Methodology*. 2020. URL: https://www.census.gov/programs-surveys/asm/technical-documentation/methodology.html, archived at https://web.archive.org/web/20200920161743/https://www.census.gov/programs-surveys/asm/technical-documentation/methodology.html on 20th Sept. 2020. [see page 59]

[Vah15] Amin Vahdat. *A Look Inside Google's Data Center Network*. Keynote presentation at Open Networking Summit 2015, Santa Clara. 2015. URL: https://youtube.com/watch?v=FaAZAII2x0w, archived at https://web.archive.org/web/20190214081757/https://youtube.com/watch?v=FaAZAII2x0w on 14th Feb. 2019. [see page 3]

[Val90] Leslie G. Valiant. 'A Bridging Model for Parallel Computation'. In: *Communications of the ACM* 33.8 (1990), pages 103–111: ACM. DOI: 10.1145/79173.79181. [see page 7]

[Var+91] Peter J. Varman, Scott D. Scheufler, Balakrishna R. Iyer and Gary R. Ricard. 'Merging Multiple Lists on Hierarchical-Memory Multiprocessors'. In: *Journal of Parallel and Distributed Computing* 12.2 (1991), pages 171–177: Elsevier. DOI: 10.1016/0743-7315(91)90022-2. [see pages 23, 26, 29, 50]

[Vit85] Jeffrey Scott Vitter. 'Random Sampling with a Reservoir'. In: *ACM Transactions on Mathematical Software (TOMS)* 11.1 (1985), pages 37–57: ACM. DOI: 10.1145/3147.3165. [see page 59]

[vNeu45] John von Neumann. *The First Draft Report on the EDVAC (Electronic Discrete Variable Automatic Calculator)*. Technical report. Contract no. W-670-ORD-4926 between the United States Army Ordinance Department and the University of Pennsylvania. Moore School of Electrical Engineering, University of Pennsylvania, June 1945. [see page 4]

[Vos91] Michael D. Vose. 'A Linear Algorithm For Generating Random Numbers With a Given Distribution'. In: *IEEE Transactions on Software Engineering (TSE)* 17.9 (1991), pages 972–975. DOI: 10.1109/32.92917. [see pages 17, 55, 56, 58, 60, 105]

[Wal77] Alastair J. Walker. 'An Efficient Method for Generating Discrete Random Variables with General Distributions'. In: *ACM Transactions on Mathematical Software (TOMS)* 3.3 (1977), pages 253–256: ACM. DOI: 10.1145/355744.355749. [see pages 55, 58]

[WB15] Michael Walfish and Andrew J. Blumberg. 'Verifying computations without reexecuting them'. In: *Communications of the ACM* 58.2 (2015), pages 74–84: ACM. DOI: 10.1145/2641562. [see page 116]

[WC81] Mark N. Wegman and Larry Carter. 'New Hash Functions and Their Use in Authentication and Set Equality'. In: *Journal of Computer and System Sciences* 22.3 (1981), pages 265–279: Elsevier. DOI: 10.1016/0022-0000(81)90033-7. [see pages 123, 130]

[WE80] Chak-Kuen Wong and Malcolm C. Easton. 'An Efficient Method for Weighted Sampling Without Replacement'. In: *SIAM Journal on Computing* 9.1 (1980), pages 111–113: SIAM. DOI: 10.1137/0209009. [see page 58]

[Wika] Wikichip.org. *Infinity Fabric (IF) - AMD*. URL: https://en.wikichip.org/wiki/amd/infinity_fabric, archived at https://web.archive.org/web/20190807082137/https://en.wikichip.org/wiki/amd/infinity_fabric on 7th Aug. 2019. [see page 2]

[Wikb] Wikichip.org. *Mesh Interconnect Architecture - Intel*. URL: https://en.wikichip.org/wiki/intel/mesh_interconnect_architecture, archived at https://web.archive.org/web/20191114152915/https://en.wikichip.org/wiki/intel/mesh_interconnect_architecture on 14th Nov. 2019. [see page 2]

[Wim+15a] Martin Wimmer, Jakob Gruber, Jesper Larsson Träff and Philippas Tsigas. 'The Lock-free *k*-LSM Relaxed Priority Queue'. In: *Computing Research Repository (CoRR)* (2015). arXiv: 1503.05698 [cs.DS]. [see page 25]

[Wim+15b] Martin Wimmer, Jakob Gruber, Jesper Larsson Träff and Philippas Tsigas. 'The lock-free *k*-LSM relaxed priority queue'. In: *20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*. 2015, pages 277–278. DOI: 10.1145/2688500.2688547. [see page 25]

[Yi+09] Ke Yi, Feifei Li, Graham Cormode, Marios Hadjieleftheriou, George Kollios and Divesh Srivastava. 'Small synopses for group-by query verification on outsourced data streams'. In: *ACM Transactions on Database Systems (TODS)* 34.3 (2009), 15:1–15:42. DOI: 10.1145/1567274.1567277. [see pages 116, 123, 134]

[YZ13] Ke Yi and Qin Zhang. 'Optimal Tracking of Distributed Heavy Hitters and Quantiles'. In: *Algorithmica* 65.1 (2013), pages 206–223: Springer. DOI: 10.1007/s00453-011-9584-4. [see page 25]

[Zah+10] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker and Ion Stoica. 'Spark: Cluster Computing with Working Sets'. In: *2nd USENIX Workshop on Hot Topics in Cloud Computing (HotCloud)*. 2010. URL: https://www.usenix.org/conference/hotcloud-10/spark-cluster-computing-working-sets, archived at https://web.archive.org/web/20200415112422/https://www.usenix.org/conference/hotcloud-10/spark-cluster-computing-working-sets on 15th Apr. 2020. [see pages 15, 109, 111, 118]

[Zah+13] Matei Zaharia, Tathagata Das, Haoyuan Li, Timothy Hunter, Scott Shenker and Ion Stoica. 'Discretized streams: fault-tolerant streaming computation at scale'. In: *24th ACM Symposium on Operating Systems Principles (SOSP)*. 2013, pages 423–438. DOI: 10.1145/2517349.2522737. [see pages 10, 15]

[Zha+20] Kai Zhao, Sheng Di, Sihuan Li, Xin Liang, Yujia Zhai, Jieyang Chen, Kaiming Ouyang, Franck Cappello and Zizhong Chen. 'Algorithm-Based Fault Tolerance for Convolutional Neural Networks'. In: *Computing Research Repository (CoRR)* (Mar. 2020). arXiv: 2003.12203 [cs.DC]. [see page 117]

[Zip65] George Kingsley Zipf. *The psycho-biology of language*. Originally printed by Houghton Mifflin Co., 1935. The MIT Press, 1965. DOI: 10.4324/9781315009421. [see page 24]

[ZL79] J. F. Ziegler and W. A. Lanford. 'Effect of Cosmic Rays on Computer Memories'. In: *Science* 206.4420 (1979), pages 776–788: AAAS. DOI: 10.1126/science.206.4420.776. [see page 111]