The Processing-in-Memory Model

Hongbo Kang
khb20@mails.tsinghua.edu.cn
Tsinghua University
China

Phillip B. Gibbons
gibbons@cs.cmu.edu
Carnegie Mellon University
United States

Guy E. Blelloch
guyb@cs.cmu.edu
Carnegie Mellon University
United States

Laxman Dhulipala
laxman@mit.edu
MIT CSAIL
United States

Yan Gu
ygu@cs.ucr.edu
UC Riverside
United States

Charles McGuffey
cmcguffe@cs.cmu.edu
Carnegie Mellon University
United States

ABSTRACT

As computational resources become more efficient and data sizes grow, *data movement* is fast becoming the dominant cost in computing. *Processing-in-Memory* is emerging as a key technique for reducing costly data movement, by enabling computation to be executed on compute resources embedded in the memory modules themselves.

This paper presents the *Processing-in-Memory (PIM)* model, for the design and analysis of parallel algorithms on systems providing processing-in-memory modules. The PIM model focuses on keys aspects of such systems, while abstracting the rest. Namely, the model combines (i) a CPU-side consisting of parallel cores with fast access to a small shared memory of size M words (as in traditional parallel computing), (ii) a PIM-side consisting of P PIM modules, each with a core and a local memory of size $\Theta(n/P)$ words for an input of size n (as in traditional distributed computing), and (iii) a network between the two sides. The model combines standard parallel complexity metrics for both shared memory (work and depth) and distributed memory (local work, communication time) computing. A key algorithmic challenge is to achieve load balance among the PIM modules in both their communication and their local work, while minimizing the communication time. We demonstrate how to overcome this challenge for an ordered search structure, presenting a parallel PIM-skiplist data structure that efficiently supports a wide range of batch-parallel queries and updates.

CCS CONCEPTS

• **Theory of computation** → **Parallel computing models**; **Distributed computing models**; *Sorting and searching*; *Predecessor queries*; • **Hardware** → *Emerging architectures.*

KEYWORDS

processing-in-memory; models of parallel computation; skip list; batch-parallel data structures

This work is licensed under a Creative Commons Attribution International 4.0 License.

SPAA '21, July 6–8, 2021, Virtual Event, USA.
© 2021 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-8070-6/21/07.
https://doi.org/10.1145/3409964.3461816

ACM Reference Format:
Hongbo Kang, Phillip B. Gibbons, Guy E. Blelloch, Laxman Dhulipala, Yan Gu, and Charles McGuffey. 2021. The Processing-in-Memory Model. In *Proceedings of the 33rd ACM Symposium on Parallelism in Algorithms and Architectures (SPAA '21), July 6–8, 2021, Virtual Event, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3409964.3461816

1 INTRODUCTION

As computational resources become more efficient and data sizes grow, *data movement* is fast becoming the dominant cost in computing. *Processing-in-memory* [21] (a.k.a., *near-data-processing*) is emerging as a key technique for reducing costly data movement, by enabling computation to be executed on CPU resources embedded in the memory modules themselves. Instead of a traditional memory hierarchy where data must be moved to the CPU cores to be computed on, processing-in-memory enables compute to be pushed to the memory, thereby saving data movement.

Although various forms of processing-in-memory have been proposed and studied going back to at least 1970 [29], it is only now gaining widespread attention as an emerging key technology (see [21] for a survey with hundreds of recent references). Although there are many technologies proposed for enabling processing-in-memory, whether it be via design, integration, packaging, and combinations thereof, a particularly promising approach is the use of 3D die-stacked memory cubes. In such emerging memory cubes, memory layers are stacked on top of a processing layer in one tightly-integrated package, enabling a simple compute core to have low-latency, high-bandwidth memory access to the entire cube of memory.

While there has been considerable work on the systems/architecture/technology side of processing-in-memory [21], there has been very little work on the theory side. Fundamental open problems include: What is a good model of computation for processing-in-memory? How is algorithm design different? What are the fundamental limitations of using processing-in-memory? Can we provide theoretical justification for why processing-in-memory is a good idea?

In this paper, we take a first step towards answering some of these questions. We define the first model of computation for emerging processing-in-memory systems, which seeks to capture both the advantages and limitations of such systems. (See §2.2 for related work.) Our *Processing-in-Memory (PIM)* model (Fig. 1) combines (i) a *CPU side* consisting of parallel cores with fast access to a small shared memory of size M words (as in traditional shared-memory parallel computing), (ii) a *PIM side* consisting of P PIM modules, each

with a core and a local memory of size $\Theta(n/P)$ words for an input of size n (as in traditional distributed-memory computing), and (iii) a *network* between the two sides. The model combines standard parallel complexity metrics for both shared memory (work and depth) and distributed memory (local work and communication time) computing.

Although many models for shared memory and many models for distributed memory have been proposed, it is the combination of the two in a single model that makes the PIM model novel and interesting for algorithm design. Effective use of the PIM side requires "PIM-balance", namely, balanced work and communication across all the PIM modules, even though each compute task is being offloaded to the local memory holding the needed data. One could apply standard approaches of randomly mapping shared memory to distributed memory (e.g., [32]) and then treat the entire model as a shared-memory model, but then all memory accesses would be non-local, defeating the goal of using processing-in-memory in order to minimize data movement. Conversely, one could ignore the shared memory and treat the entire model as a distributed-memory model, but our results show benefits from using the shared memory (e.g., for sorting up to M numbers without incurring any network communication, or for avoiding PIM load imbalance in balls-in-bins settings with small balls-to-bins ratios).

As a case study of parallel algorithm design on such a model, we consider the challenge of efficiently maintaining a skip list [23, 25] under adversary-controlled batch-parallel queries and updates.

The two main contributions of the paper are:

- We define the PIM model, the first model of parallel computation capturing the salient aspects of emerging processing-in-memory systems.
- We design and analyze a PIM-friendly skip-list data structure, which efficiently supports a wide range of batch-parallel updates, point queries, and range queries, under adversary-controlled batches. A key feature of our algorithms is that (nearly) all the performance metrics, including network communication, are independent of the number of keys n in the data structure and also independent of any query/update skew (that arises, e.g., when a large number of distinct successor queries all target the same successor node).

2 MODEL

2.1 The Processing-in-Memory Model

Our goal is to define a model of parallel computation that focuses on the key aspects of systems with processing-in-memory modules, while abstracting away the rest. In particular, our model has only two parameters, M and P, and instead, we use separate metrics for aspects that have system-dependent costs (e.g., computation on traditional CPU cores vs. computation on PIM modules vs. communication). A key aspect we model is that processing-in-memory is fast because there is processing (a core) embedded with each memory module, and each memory module is small enough that the latency for the core to access its local memory is low and the throughput is high (significantly lower/higher than accessing the same memory from one of the traditional CPU cores—this is the savings driving the push to processing-in-memory). As such, PIM divides the system's "main memory" into a collection of "core + local memory" modules, akin to traditional distributed memory models. Unlike distributed memory models, however, there is also the "traditional" part of the system comprised of multiple cores sharing a fast last-level cache. Thus, our PIM model combines a shared-memory model (for a small amount of shared memory) with a distributed-memory model. And indeed, it is the combination of both that makes the PIM model both novel and interesting for algorithm design.

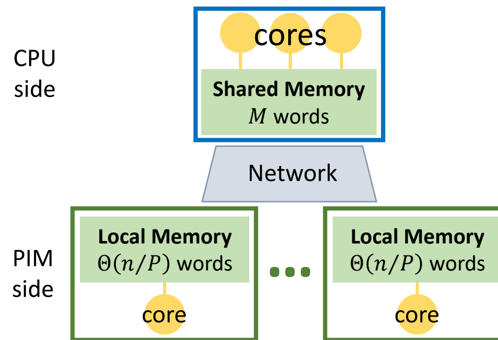

Figure 1: The PIM Model

As illustrated in Fig. 1, the ***Processing-in-Memory (PIM)*** model combines a ***CPU side*** consisting of parallel cores (called ***CPU cores***) with fast access to a small ***shared memory*** of size M words (as in traditional shared-memory parallel computing), a ***PIM side*** consisting of P ***PIM modules***, each with a core (called a ***PIM core***) and a ***local memory*** of size $\Theta(n/P)$ words (as in traditional distributed-memory computing), and a network between the two sides. Each CPU or PIM core executes unit-work instructions from a random access machine (RAM) instruction set augmented, on the CPU side, with an atomic test-and-set primitive (for synchronization among the asynchronous CPU cores). In general, specific algorithms may use stronger synchronization primitives (e.g., compare-and-swap), where noted (but none of the algorithms in this paper require more than a test-and-set).

A CPU core offloads work to a PIM core using a **TaskSend** instruction, which specifies a PIM-core ID and a task (function ID and arguments) to execute on that PIM core. The network is used to route such tasks to the designated PIM core, where the task is queued up and the PIM core is awakened if it is asleep. Each PIM core repeatedly invokes an iterator that removes a task from its queue and then executes the task. If the queue is empty, the PIM core goes to sleep. Tasks specify a shared-memory address to write back the task's return value on completion of the task. All tasks must return either a value or an acknowledgment of completion. We will sometimes say a PIM module A offloads a task to another PIM module B—this is done by A returning a value to the shared memory, which in turn causes the offload from the CPU side to B.

The PIM model combines standard complexity metrics for both shared (work and depth) and distributed (local work, communication time) computing. On the CPU side, we account for the ***CPU work*** (total work summed over all the CPU cores) and ***CPU depth*** (sum of the work on the critical path on the CPU side), a.k.a. *CPU span*. On the PIM side, we account for the ***PIM time***, which is the maximum local work on any one PIM core. Communication costs are measured in terms of *IO time*, as follows. The network operates

in bulk-synchronous rounds [32], where a set of parallel messages, each with a constant number of words, is sent between the CPU side and the PIM side, followed by a barrier synchronization. As in the BSP model [32], we define the notion of an h-relation, but in the PIM model, h is the maximum number of messages to/from any one PIM module (i.e., ignoring the CPU side). Let $h_1, \ldots, h_r$ be the h's for a computation with r bulk-synchronous rounds. Then the ***IO time*** is defined to be $\sum_{i=1}^{r} h_i$. The cost of the r synchronization barriers is $r \log P$.[1]

We assume that $M = O(n/P)$ and $M = \Omega(P \cdot \text{polylog}(P))$. The algorithms we present in this paper have the further restriction that M is independent of n and at most $\Theta(P \log^2 P)$. M is assumed to be small because, as discussed above, our shared memory is intended to model the fast last-level cache on the CPU side, while the PIM modules comprise the "main memory" of the system.

Discussion. To simplify the model, the number of cores on the CPU side is not specified. Because we analyze the CPU side using work-depth analysis and we assume a work-stealing scheduler [10], there is no need to pre-specify the number of cores on that side. For any specified number of CPU cores P', the time on the CPU side for an algorithm with W CPU work and D CPU depth would be $O(W/P' + D)$ expected time [3, 10]. There is no penalty for the stealer to execute a task instead of a stealee, because both have the same fast access to the CPU side shared memory.[2] As a result of not pre-specifying, the PIM model can be used to model systems with more or fewer CPU cores than PIM cores, and with CPU cores that are more powerful than PIM cores.

Communicating an h-relation over the network is charged as h IO time, but it is *not* charged as CPU work/depth or PIM time. This is for simplicity, and one could always determine what that cost would be, if desired, by simply adding $h \cdot P$ to the CPU work and h to the PIM time. For the algorithms we present in this paper, doing so would *not* asymptotically increase the CPU work or PIM time.

We allow CPU cores to perform concurrent reads and writes to the shared memory. A variant of the model could account for write-contention to shared memory locations, by assuming k cores writing to a memory location incurs time k—the so-called "queue-write" model [15]. We leave such analysis to future work.

In summary, to analyze an algorithm for the PIM model, one needs to give the CPU work and depth, the PIM time, and the IO time. Other metrics of interest are the number of bulk-synchronous rounds (for algorithms where the synchronization cost dominates the IO time) and the minimum CPU memory size needed for the algorithm.

Algorithm design. The model targets "in-memory, not in-cache" algorithms, in which the space needed for the entire computation is at most $O(n)$ words but much larger than M words. The input starts evenly divided among the PIM modules. Outputs of size $> M$ are stored in the PIM modules as well. For algorithms that involve maintaining a data structure under queries and updates, the queries/updates are assumed to arrive on the CPU side and query results are returned to the CPU side. As is common for parallel algorithm design, we assume the *batch-parallel* setting [1, 8, 13, 24, 30, 31, 33], in which queries or updates arrive as a set that can be executed in parallel, called a ***batch***. We consider adversary-controlled queries/updates, where the adversary determines the sequence of batches and the queries or updates within each batch, subject to the following constraints: (i) all the queries/updates within a batch are the same operation type (e.g., all inserts), (ii) there is a minimum batch size, typically $P \cdot \text{polylog}(P)$, and (iii) the queries/updates cannot depend on the outcome of random choices made by the algorithm (and hence, e.g., cannot depend on the outcome of a randomized mapping of data structure components to PIM modules).

Many of our bounds hold with high probability (*whp*) in P: A bound is $O(f(P))$ ***whp*** if, for all $\alpha \geq 1$, the bound is $O(\alpha \cdot f(P))$ with probability at least $1 - 1/P^{\alpha}$.

PIM-balance. A key algorithmic challenge is to achieve load balance among the PIM modules in both their communication and their local work, while minimizing the IO time. As noted above, the CPU side can achieve load balance via work-stealing because the stealer has the same fast access to the shared memory as the stealee. But on the PIM side, work-stealing is impractical because tasks are tied to a specific local memory (the whole point of processing-in-memory), and hence fast local accesses by the stealee would be replaced by slow non-local accesses by the stealer.

We say an algorithm is ***PIM-balanced*** if it takes $O(W/P)$ PIM time and $O(I/P)$ IO time, where W is the sum of the work by the PIM cores and I is the total number of messages sent between the CPU side and the PIM side. The challenge in achieving PIM-balance is that both PIM time and IO time (specifically the h for each h-relation) are based on the *maximum* across the PIM modules (not the average).

Thus, special care is needed to achieve PIM-balance. Algorithms must avoid offloading to PIM modules in an unbalanced way. Because computation is moved to where data reside, data access must be balanced across PIM modules. But as noted in §1, while randomly mapping data addresses to PIM modules would help balance data access, it would defeat the purpose of processing-in-memory because all data accesses would be non-local. Thus, more selective randomization is needed. Note as well that offloading P tasks to P PIM modules randomly would not be PIM-balanced, because some PIM module would receive $\Theta(\log P / \log\log P)$ tasks *whp* [6], implying $\Theta(\log P / \log\log P)$ IO time *whp*. We will use the following two balls-in-bins lemmas.

Lemma 2.1 ([26]). *Placing $T = \Omega(P \log P)$ balls into P bins randomly yields $\Theta(T/P)$ balls in each bin* whp.

Lemma 2.2 ([22, 27]). *Placing weighted balls with total weight $W = \sum w_i$ and weight limit $W/(P \log P)$ into P bins randomly yields $O(W/P)$ weight in each bin* whp.

The cited references for Lemma 2.2 provide a proof for $O(W/P)$ weight in expectation. A proof for *whp* appears in the Appendix.

2.2 Related Work

One approach to designing algorithms for PIM systems would be to use known hashing-based emulations of shared memory (e.g., a PRAM) on a distributed memory model (e.g., the BSP model).

[1] In this paper, we do not explicitly discuss synchronization cost, except in the one case (Theorem 5.1) where it dominates the IO time.

[2] For models that provide a private cache for each CPU core, there is a slight cost to warm up the stealer's private cache [2].

Valiant [32] showed that each step of an exclusive-read exclusive-write (EREW) PRAM with $p \log p$ virtual processors can be emulated on a p-processor BSP in $O(\log p)$ steps *whp* (assuming the BSP's gap parameter g is a constant and its L parameter is $O(\log p)$), and that $O(\log p)$ steps of a BSP can be emulated on a p-node hypercube in $O(\log p)$ steps *whp*. Integer semi-sorting can be used to extend the emulation to the concurrent-read concurrent-write (CRCW) PRAM within the same bounds, although the constants are much higher [32]. However, these emulations are impractical because all accessed memory incurs *maximal* data movement (i.e., across the network between the CPU cores and the PIM memory), which is the exact opposite of the goal of having processing-in-memory in order to minimize data movement. The algorithms in this paper will make *selective* use of both hashing and sorting.

Choe et al. [11] studied concurrent data structures on processing-in-memory systems. They provided an empirical evaluation of PIM-aware algorithms for linked lists, FIFO queues, and skip lists, and showed that lightweight modifications to PIM hardware can significantly improve performance. They did not define a model of computation. Also unlike our paper, they studied skip lists for a workload of uniformly-random keys, and provide a skip list algorithm that partitions keys by disjoint key ranges. Under uniformly-random keys, this works well (because in expectation, each PIM module processes the same number of queries/updates), but it would serialize (i.e., no parallelism) in the more general case we consider of adversary-controlled query/update keys, whenever all keys fall within the range hosted by a single PIM-module.

An earlier paper by Liu et al. [19] presented a performance model for a processing-in-memory system that specified parameters for (i) the latency of a memory access by a CPU core, (ii) the latency of a local memory access by a PIM core, and (iii) the latency of a last-level cache access by a CPU core. The performance model was used to analyze different algorithms for concurrent data structures, including a skip list. However, the skip list algorithm partitioned keys across the PIM modules using disjoint key ranges, as above.

Das et al. [12] proposed a model for a system with both traditional main memory (DRAM) and high-bandwidth memory (HBM). In their model, each of P CPU cores is connected by its own channel to a shared HBM of limited size and there is a single channel between the HBM and an infinite main memory. Das et al. showed how to automatically manage which data to keep in the HBM and when to transfer data from HBM to main memory. There was no distributed memory in their model and only one type of cores. An earlier paper by Bender et al. [5] proposed and studied an external-memory model with both a small but high-bandwidth memory and a large but low-bandwidth memory. The bandwidth difference was modeled as retrieving a much larger block in a single external-memory read or write.

Ziegler et al. [34] analyzed different ways to distribute a tree-based structure over multiple computers. Besides coarse-grained partitioning by key ranges, they study coarse-grained partitioning by hash and fine-grained partitioning that randomly distributes all nodes. Both partitioning schemes improve performance on skewed workloads, but partitioning by hash decreases range query performance, and fine-grained partitioning decreases performance on uniform workloads.

3 PIM-BALANCED SKIP LIST

In the remainder of the paper, we present our efficient, PIM-balanced algorithms for maintaining a skip list [23, 25] under adversary-controlled batch-parallel queries and updates. Our skip list supports seven types of point operations and range operations: Get (key), Update (key, value), Predecessor (key), Successor (key), Upsert (key, value)[3], Delete (key), and RangeOperation (LKey, RKey, Function). Recall from §2.1 that all operations in a batch are of the same type, there is a minimum batch size, and the adversary's choice of keys cannot depend on random choices made by the algorithm.

This section describes the high-level design of our algorithms.

3.1 Overview

Some prior work on ordered data structures in a PIM-equipped or distributed system used range partitioning [11, 19, 34]. Although these algorithms may save IO between CPU and PIM modules under uniform workloads, their structure, even with dynamic data migration, suffers from PIM-imbalance for skewed or adversarial workloads that force many operations to occur on a small number of range partitions. Some other methods are invented to relieve this, but they bring new problems. Coarse-grain partitioning by hash has low range queries performance because range queries must be broadcasted. Fine-grained partitioning causes too much IO because every key search would access nodes in many different PIM modules.

Our approach. The key idea of our approach is to combine (i) a uniform load-balancing scheme for a subset of the keys with (ii) a scheme that redundantly stores in every PIM node the commonly-used nodes in the upper levels of the skip list. More formally, we divide the skip list horizontally into a lower part and an upper part, and refer to the nodes in the two parts as ***upper-part*** nodes and ***lower-part*** nodes. This is shown in Fig. 2, where upper-part nodes are in white, and the lower part nodes have different color and texture according to their PIM module. The upper part is replicated in *all PIM modules*, and the lower part is *distributed randomly* to PIM modules by a hash function on the (keys, level) pairs. (In this paper, levels are counted bottom up with leaves at level 0). For an upper-part node, its replicas are stored across all PIM modules at the *same* local memory address on each PIM. We refer to the nodes in the last level of the upper part (second level in Fig. 2) as ***upper-part leaves***.

To be specific, for a skip list of $n = \Omega(P \log P)$ key-value pairs[4], if the height of the lower part is set to $h_{low} = \log P$, the size of the upper part will be only $O(n/P)$ *whp*. On the other hand, for a search path in this skip list, $O(\log n)$ nodes will fall into the upper part and only $O(\log P)$ nodes will fall into the lower part *whp* [17].

3.2 Pointer Structure

Next, we present a more detailed description of our data structure. As in traditional skip lists, each node in our skip list has four pointers: *left*, *right*, *up* and *down*. *up* pointers to upper-part nodes point

[3] Upsert is an operation that inserts the key-value pair if the key does not already exist, or updates the value if it does.

[4] Throughout this paper, we consider a skip list where a level i node also appears in level $i+1$ with probability $1/2$ and all logarithms are base 2.

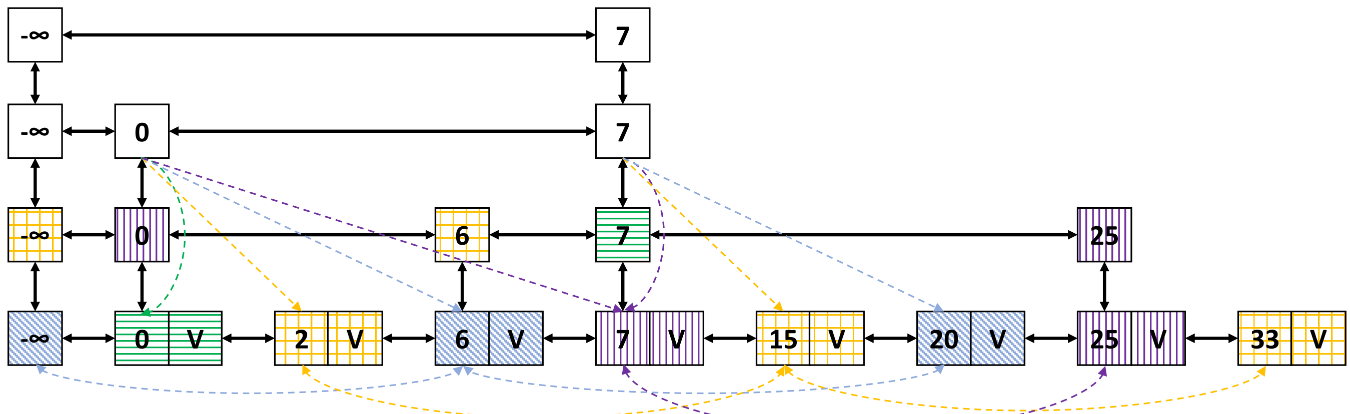

Figure 2: The pointer structure of a skip list on a 4-PIM-module system. Nodes on different PIM modules are of different color, and are in white when they are replicated among all PIM modules. Solid pointers are used in point operations, and are stored in the PIM module of their FROM node. Dashed pointers are used in range operations, and are stored in the PIM module of their TO node (as indicated by their color). Dashed pointers for $-\infty$ are omitted.

to the PIM-local copy of the upper part, and *up* or *down* pointers to lower-part nodes may point to nodes in different PIM modules. When we need to access remote addresses, we use the **TaskSend** instruction, as follows. A **RemoteWrite** is performed by sending a write task, and a **RemoteRead** is performed by sending the current task to the remote PIM module to proceed.

To support range queries, we add several additional pointers. For leaves, we add two additional pointers: *local-left* and *local-right*. These pointers point to the previous and next leaf in the *same* PIM module to form a ***local leaf list*** for each PIM module. For each upper-part leaf on each PIM module, we maintain an additional pointer called **next-leaf**, which points from an upper-part leaf to its successor in the local leaf list. We illustrate these three types of pointers in Fig. 2 (dashed pointers).

Theorem 3.1. *The skip list takes $O(n)$ words in total, and $O(n/P)$ words* whp *in each PIM module.*

Proof. By Lemma 2.1, for $n = \Omega(P \log P)$, $O(n/P)$ lower-part nodes are sent to each PIM module *whp*. The upper part has $O(n/P)$ nodes *whp*. As the result, each PIM module holds $O(n/P)$ nodes of constant size *whp*. □

We introduce three terms for the sake of clarity when discussing our algorithms. We call nodes on the search path from the root to a specific node its ***ancestors***. Note that since the upper part is replicated, there are multiple physical search paths to a node. These paths share the same lower-part nodes, and their upper nodes are replicas of the same nodes on different PIM modules. We conceptually think of these identical search paths as a single (non-redundant) path, which yields a tree, and we define ***parent*** and ***child*** based on this tree.

3.3 Challenge of Imbalanced Node Access

Although we randomly distribute the lower-part nodes among PIM processors, simply batching operations does not guarantee PIM-balance. For example, multiple Get (or Update) operations with the same key can cause contention on the PIM module holding the key. Deduplication of queries can solve imbalance for these two operations, but it cannot solve imbalance caused by other important operations. For example, as noted earlier and analyzed in §4.2, if the adversary requests the Successor of multiple different keys with the *same successor*, then the path to the result will be accessed multiple times, causing contention. In the extreme, this can serialize the entire batch of Successor operations.

Note that in these examples, PIM-imbalance is a direct result of imbalanced node access. Therefore, our approach to PIM-balance is to avoid imbalanced node access. In the next two sections, we present detailed algorithms for the individual operations.

4 POINT OPERATIONS

This section presents our algorithms for the six point operations supported by our skip list. Table 1 summarizes the bounds we obtain.

4.1 Get(k) and Update(k, v)

Execution of a single operation. Because the lower-part nodes are distributed to PIM modules by a random hash function, the Get (Update can be solved similarly) operation can use this function as a shortcut to find the PIM module that the target node must be stored on. By storing an additional hash table locally on each PIM module to map keys to leaf nodes directly, we can efficiently process Get (Update) queries.

Specifically, within a PIM module, we use a de-amortized hash table supporting $O(1)$ *whp* work operations [16]. The table supports the $O(n/P)$ keys that are stored in this PIM node in $O(1)$ *whp* PIM work per Get, Update, Delete, and Insert operation.

To execute each operation, we send it directly to the PIM module for the key according to the hash value, and then query for the key within the PIM's local hash table, ignoring non-existent keys. This takes $O(1)$ messages and $O(1)$ *whp* PIM work. Note that this approach works because Get (Update) operations neither use nor modify the pointer structure.

PIM-balanced batch execution. The batched Get operation is executed in batches of size $P \log P$. It first goes through a parallel semisort [9, 18] on the CPU side to remove duplicate operations.

Operation	Batch Size	IO time	PIM time	CPU work/op	CPU depth	Minimal M needed
GET / UPDATE	$P \log P$	$O(\log P)^*$	$O(\log P)^*$	$O(1)^\dagger$	$O(\log P)^*$	$\Theta(P \log P)$
PREDECESSOR / SUCCESSOR	$P \log^2 P$	$O(\log^3 P)^*$	$O(\log^2 P \cdot \log n)^*$	$O(\log P)^\dagger$	$O(\log^2 P)^*$	$\Theta(P \log^2 P)^*$
UPSERT	$P \log^2 P$	$O(\log^3 P)^*$	$O(\log^2 P \cdot \log n)^*$	$O(\log P)^\dagger$	$O(\log^2 P)^*$	$\Theta(P \log^2 P)^*$
DELETE	$P \log^2 P$	$O(\log^2 P)^*$	$O(\log^2 P)^*$	$O(1)^\dagger$	$O(\log^2 P)^*$	$\Theta(P \log^2 P)^*$

Table 1: Complexity of our batch-parallel point operations on a skip list of n keys. *CPU work/op* is the total CPU work for the batch divided by the batch size. $*$: with high probability (*whp*) in P. $\dagger$: in expectation. Bounds without superscripts are worst-case bounds.

Then, it sends each query to the target PIM module and finishes the rest of the computation locally on the PIM module.

THEOREM 4.1. *Batched* GET (UPDATE) *operations using a batch size of $P \log P$ can be executed* whp *in $O(\log P)$ IO time, $O(\log P)$ PIM time, and $O(\log P)$ CPU depth. The execution performs $O(P \log P)$ expected CPU work.*

PROOF. Semisorting the batch takes $O(P \log P)$ expected CPU work with $O(\log P)$ *whp* depth [9]. By Lemma 2.1, sending up to $P \log P$ GET operations with different keys to random PIM modules sends $O(\log P)$ operations to each PIM module *whp*, and hence takes $O(\log P)$ *whp* IO time. Because each operation takes $O(1)$ *whp* PIM work, the batch takes $O(\log P)$ *whp* PIM time. □

We note that this algorithm is PIM-balanced because the PIM time is $O(\log P) = O(W/P)$ for $W = O(P \log P)$ PIM work and the IO time is $O(\log P) = O(I/P)$ for $I = O(P \log P)$ messages, as required. Importantly, PIM-balance holds irrespective of the distribution of these queries.

4.2 PREDECESSOR(k) and SUCCESSOR(k)

A PREDECESSOR(k) (SUCCESSOR(k)) query returns a pointer to the largest (smallest) key in the skip list less than (greater than) or equal to k. For keys that are in the skip list, we can apply a similar idea as in our GET algorithm to shortcut directly to the PIM node containing the key. More generally, though, the requested key k is not in the skip list, and the predecessor/successor must be found using the pointers in the skip list. A naïve idea to handle this case would be to broadcast the query to all PIM modules to execute it, but this would take $\Theta(P)$ IO messages and $\Theta(P \log n)$ *whp* PIM work for each query. In what follows, we focus on SUCCESSOR queries; PREDECESSOR queries are symmetric.

Execution of a single operation. For a single operation, we follow the same search approach of an ordinary skip list. First, we send the SUCCESSOR operation to a random PIM module, and traverse from the root to the corresponding upper-part leaf. Because the lower part is distributed on different PIM modules, the current PIM module may need to send the SUCCESSOR operation to a lower-part node by a TaskSend. After repeatedly querying for lower-part nodes and reaching a leaf, the operation stops, and the leaf is sent back to the shared memory. Note that accessing each node in the lower part takes $O(1)$ IO messages. Because each search path has $O(\log n)$ *whp* nodes in the upper part and $O(\log P)$ *whp* nodes in the lower part [17], each SUCCESSOR operation takes $O(\log n)$ *whp* PIM work and $O(\log P)$ *whp* IO messages.

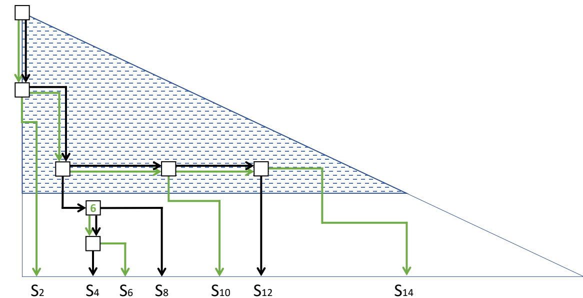

Figure 3: The pivot search paths in stage 1 of the SUCCESSOR algorithm. The upper part is shaded blue. Black paths are from phase 1 (to S_8) and phase 2 (to S_4 and S_{12}), and green paths are from phase 3. Note that S_6 starts at LCA(S_4, S_8), while S_{10} starts at the root—this is because LCA(S_4, S_8) is in the lower part, while LCA(S_8, S_{12}) is not.

PIM-imbalanced batch execution. We first show that just naïvely distributing the queries in a batch would lead to an imbalanced workload on different PIM modules. In the *naïve search algorithm*, we execute all operations in parallel. In step 0, we send the operations to random PIM modules and traverse the upper part of the skip list. Then in each subsequent step, we push each query one node further in its search path on the lower part. It takes $O(\log P)$ *whp* steps to finish every query.

However, the adversary can request a batch of $P \log^2 P$ *different* keys all with the *same* successor, causing lower-part nodes to become contention points. In such cases, executing one step can take up to $P \log^2 P$ IO time, and the whole process can take up to $P \log^3 P$ IO time, completely eliminating parallelism.

PIM-balanced batch execution. Instead, to achieve good load balance, we take special care to avoid contention on nodes. This is done in two stages: In stage 1, the algorithm picks pivots, computes the SUCCESSOR of each pivot, and also stores the lower-part search paths. Crucially, we ensure that performing the searches for the pivots can be provably done without contention. Then in stage 2, we execute all of the sets of operations between pivots, using the saved lower-part search paths to accelerate these searches.

To be specific, the batched SUCCESSOR operation is executed in batches of size $P \log^2 P$. The keys in the batch are first sorted on the CPU side. We pick $P \log P$ pivots to divide the batch into segments of $\log P$ operations, and we also pick the operation with the smallest key and the largest key in the batch as pivots. We then compute the SUCCESSOR for each pivot in a parallel divide-and-conquer style (see Fig. 3 as an example):

(1) In phase 0, a CPU core packs the smallest-key and largest-key pivots into a mini-batch of size two. It then runs the naïve

search algorithm on the mini-batch. During the execution, PIM modules send lower-part nodes on the search path for each operation back to the shared memory.
(2) In phase 1, we execute the operation that is the median pivot using a *start node hint*: If two of the paths recorded in phase 0 (paths to the result of the smallest-key and the largest-key) share no lower-part node, start at the root. If they share a leaf, directly take the leaf as the result. Otherwise, start from the lowest common lower-part node of the two paths.
(3) After phase i, unexecuted pivots are divided into 2^i segments. We pack the median pivot of each segment into a mini-batch. In phase $i+1$, use the path to the two ends of each segment to generate the start node hint (as in phase 1), execute the naïve search algorithm using the hint, and record the path for the subsequent phases.

After stage 1, we have the lower-part node path for each pivot operation on the CPU side. In stage 2, we use the recorded search path for the pivots to generate start node hints (as done in each phase of stage 1). We then execute all operations using the naïve search algorithm from the start nodes.

Lemma 4.2. *No node will be accessed more than* 3 *times in each phase in stage 1.*

Proof. We prove the lemma by contradiction. As mentioned in §3.2, joining all possible search paths gives a directed tree. Suppose in the i'th phase, one lower-part node is accessed by 4 pivot operations with key $i_1 \leq i_2 \leq i_3 \leq i_4$. Then there must be three operations with keys $j_1 < j_2 < j_3$ where $i_1 \leq j_1 \leq i_2 \leq j_2 \leq i_3 \leq j_3 \leq i_4$ executed before the i'th phase. These 3 search paths cut the tree into 4 non-empty pieces, and each search chain falls into one piece. The pieces are non-empty because if any piece is empty, we can directly get the result without accessing any nodes. Note that the search-path tree is a binary tree, so it's impossible for 4 non-empty pieces to share a node, contradicting the assumption that 4 operations access the same node. □

Theorem 4.3. *Batched* Successor *(*Predecessor*) operations using a batch size of* $P\log^2 P$ *can be executed* whp *in* $O(\log^3 P)$ *IO time,* $O(\log^2 P\cdot\log n)$ *PIM time, and* $O(\log^2 P)$ *CPU depth. The execution performs* $O(P\log^3 P)$ *expected CPU work, and uses* $\Theta(P\log^2 P)$ whp *shared memory.*

Proof. Stage 1 is executed in $O(\log P)$ phases. In each phase, up to $P\log P$ pivot operations are batched and executed by the naïve search algorithm. Recall that the naïve search algorithm is executed in $O(\log P)$ *whp* steps, because in each step (after step 0) we push all $O(P\log P)$ operations one step forward. There are $O(P\log P)$ IO messages in each step. We proved constant contention in these IOs (Lemma 4.2), so each step takes $O(\log P)$ *whp* IO time.

As for PIM time, each execution of the naïve search algorithm takes $O(\log P\cdot\log n)$ *whp* PIM time, including $O(\log P\cdot\log n)$ in its step 0, and $O(\log P)$ PIM time in each following step. In total it takes $O(\log^2 P\cdot\log n)$ *whp* PIM time for $O(\log P)$ phases.

On the CPU side, sorting takes $O(P\log^3 P)$ expected CPU work, and $O(\log P)$ *whp* CPU depth [9]. In each of the following $O(\log P)$ phases, the CPU stores and processes paths of length $O(\log P)$ *whp*. Thus, storing $P\log P$ paths take $\Theta(P\log^2 P)$ memory *whp*. In each phase, finding the LCA over $O(P\log P)$ paths takes $O(P\log^2 P)$ CPU work with $O(\log P)$ depth *whp*.

In stage 2, the naïve search algorithm takes $O(\log P)$ *whp* steps. In each step, $O(P\log^2 P)$ nodes are visited with $O(\log P)$ contention, so by Lemma 2.2, it takes $O(\log^2 P)$ IO time *whp*. Similarly to stage 1 but applying Lemma 2.2 to its one and only phase, stage 2 takes $O(\log^2 P\cdot\log n)$ *whp* PIM time. Finally, on the CPU side, stage 2 only starts the naïve search algorithm using the hints from stage 1 and then collects the results at the end.

Summing the bounds for stages 1 and 2 completes the proof. □

4.3 Upsert(k, v)

The Upsert operation is a combination of Insert and Update. Namely, if k is already in the skip list, we update the value to v; otherwise, we insert (k, v) to the skip list. Our Upsert algorithm first tries to perform Update (§4.1), falling back to Insert if Update does not find the key k. To insert a key, we need to read and modify nodes on the path to its predecessor and successor. Note that it is sufficient to find the predecessor because we can get the successor using the *right* pointer in the skip list.

Execution of a single operation. A single Insert operation works as follows:

(1) Decide which levels the inserted nodes will appear (say up to l_i) based on random coin tossing.
(2) Generate and insert new upper-part nodes in the upper part for all PIM modules.
(3) Distribute the new lower-part nodes randomly to the PIM modules. Insert these nodes into the local leaf linked list and the local hash table for the PIM modules that store them.
(4) Generate the *up* and *down* pointers for each node. The only pointer from the highest lower-part node to an upper-part leaf (replicated among the PIM modules if it exists) will point to the local copy of the upper-part leaf.
(5) In each new leaf, record addresses of all lower-part new nodes in its *up* chain, and the existence of an upper-part node as a boolean flag. These are used in Delete.
(6) Run a Predecessor operation to return to the CPU side the last l_i nodes on the predecessor path. With this information, a CPU core can use RemoteWrite to modify *left* and *right* pointers to insert the new node into the horizontal linked list immediately after the predecessor.

As such, inserting a key takes $O(\log P)$ *whp* IO messages and $O(\log n)$ *whp* PIM work because of the Predecessor operation. In expectation, each Insert generates $O(1)$ lower-part nodes and $O(1/P)$ upper-part nodes, so it generates $O(1)$ expected IO messages. However, we note that the worst case and high probability bound can be much worse, because we may need to add up to $O(\log n)$ upper-part nodes to each of the P PIM modules. Next, we will show that batching helps, enabling the bounds in Table 1.

PIM-balanced batch execution. The main challenge in batch insertions is that an inserted node's neighbor can be another inserted node that is processed by another PIM module. Hence, our algorithm needs to identify this case and correctly set up these pointers (see Fig. 4). This process is an additional step shown in Algorithm 1. The complete algorithm has the following stages:

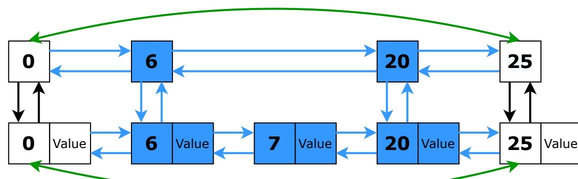

Figure 4: For batch INSERT, we insert blue nodes into a skip list starting with only white nodes, and the key challenge is to build blue pointers. For batch DELETE, we delete blue nodes from a skip list with blue and white nodes, and the key challenge is to build green pointers from the blue pointers.

(1) In parallel run steps 1–5 of the single INSERT operation for each key in the batch.
(2) Run the parallel batched PREDECESSOR operation to fetch the last l_i nodes on each predecessor path.
(3) Construct the new horizontal pointers using Algorithm 1.

The input of Algorithm 1 is an array A. $A[i][j]$ holds the address of the j'th new node (numbered from 0) of height i (i'th level bottom-up, numbered from 0) as *cur*, and pointers to its left node and right node in the same level as *pred* and *succ*, respectively. In Fig. 4, let $addr_{k,i}$ denote the address of the node with key k in level i. Then, e.g., we have $A[0][1] = (addr_{7,0}, addr_{0,0}, addr_{25,0})$, and $A[1][1] = (addr_{20,1}, addr_{0,1}, addr_{25,1})$.

Once $A[i][j]$ is generated, we can find the new nodes with matching *pred* and *succ* (the blue nodes in Fig. 4, considering each level separately), and link them accordingly. Details on how to chain them together are shown in Algorithm 1.

THEOREM 4.4. *Batched UPSERT operations using a batch size of $P \log^2 P$ can be executed* whp *in $O(\log^3 P)$ IO time, $O(\log^2 P \cdot \log n)$ PIM time, and $O(\log^2 P)$ CPU depth. The execution performs $O(P \log^3 P)$ expected CPU work, and uses $\Theta(P \log^2 P)$* whp *shared memory.*

PROOF. We first show that stage 1 of UPSERT takes $O(\log^2 P)$ IO time and $O(\log^2 P)$ PIM time *whp*. For the upper part, for every $\log^2 P$ INSERTS reaching this part, the probability distribution of the number of new upper-part nodes is a negative binomial distribution $NB(\log^2 P, 1/2)$, which is $O(\log^2 P)$ *whp*. Since $O(\log^2 P)$ INSERT *whp* reach the upper part, each batch generates $O(\log^2 P)$ new upper-part nodes *whp*, which takes *whp* $O(\log^2 P)$ IO time and $O(\log^2 P)$ PIM time. Lower part nodes are balanced so it takes $O(\log^2 P)$ IO time and $O(\log^2 P)$ PIM time *whp*. For storing the *up* pointers, the total IO is $O(P \log^2 P)$ *whp*, and the maximum IO on a single node is $\log P$, so it also finishes in $O(\log^2 P)$ IO time *whp*.

Now we analyze the cost of Algorithm 1. The PIM side takes $O(\log^2 P)$ IO time because each pointer is modified exactly once, and there is no internal dependency among them. The work on the CPU side is asymptotically bounded by the cost of PREDECESSOR, which is $O(P \log^3 P)$ expected. With high probability the heights of new nodes are $O(\log P)$, so generating nodes is of $O(\log P)$ CPU depth. Other additional operations are of $O(1)$ height. □

4.4 DELETE(k)

Note that we can perform a DELETE operation by performing INSERT in reverse. However, since a deleted key must exist in the data structure, we can obtain an $O(\log P)$ speed up by taking shortcuts to directly access leaves, as in GET and UPDATE operations.

Algorithm 1: Constructing horizontal pointers

```
Input: An 2D array A[i][j], and each element is a triple
       (cur, pred, succ) indicating the j'th newly generated
       node in the i'th level, and the predecessor and
       successor of this node.
1  parallel for i ← 0 to log P − 1 do
2    parallel for j ← 0 to A[i].size − 1 do
3      if j = A[i].size − 1 or
        A[i][j].succ ≠ A[i][j + 1].succ then
          /* the right end in a segment        */
4         RemoteWrite(A[i][j].cur.right, A[i][j].succ)
5         RemoteWrite(A[i][j].succ.left, A[i][j].cur)
6      else
          /* not the right end in a segment    */
7         RemoteWrite(A[i][j].cur.right, A[i][j + 1].cur)
8         RemoteWrite(A[i][j + 1].cur.left, A[i][j].cur)
9      if i = 0 or A[i][j].pred ≠ A[i][j − 1].pred then
          /* the left end in a segment         */
10        RemoteWrite(A[i][j].pred.right, A[i][j].cur)
11        RemoteWrite(A[i][j].cur.left, A[i][j].pred)
```

Execution of a single operation. A single DELETE operation works in four steps:

(1) The operation is sent to the PIM module holding the key, and the PIM module uses the local hash table to find the node.
(2) The PIM module marks the leaf and its *up* chain as "deleted", using the *up* chain addresses saved in that leaf. It also removes this node from the local leaf linked list.
(3) PIM modules with marked upper-part nodes will broadcast the addresses to all PIM nodes to delete all of the replicas.
(4) All marked lower-part nodes are spliced out of the horizontal linked list by linking *left* and *right* nodes.

Because DELETE removes nodes from the upper part with probability $O(1/P)$, an upper-part deletion takes $O(1)$ IO messages and $O(1)$ PIM work in expectation, and thus deleting a key takes $O(1)$ IO messages and $O(1)$ PIM work in expectation.

PIM-balanced batch execution. Similarly to batched INSERT, the main challenge in batched DELETE is to remove nodes from the horizontal linked lists in parallel because these nodes can be linked together (see Fig. 4 for an illustration of the main algorithmic goal). Our algorithm has two stages:

(1) In parallel run steps 1–3 of the single DELETE operation for each key in the batch.
(2) Use the list-contraction-based algorithm described next to remove deleted nodes from horizontal linked lists.

In stage 2, we look to splice out the marked nodes from the horizontal linked lists. Because up to $P \log^2 P$ consecutive nodes may need to be deleted from a horizontal linked list, splicing out all nodes independently in parallel will lead to conflicts. Instead, we handle this problem using list contraction. However, we cannot simply apply list contraction algorithms on remote nodes, because we need $\Omega(P \log P)$ RemoteWrites to random positions for each

bulk-synchronous round in order to guarantee PIM-balance, which is not possible for some algorithms (e.g., [20]). Also, prior BSP algorithms [14] assume that the size of the list is $\Omega(P^2 \log P)$, much larger than our target batch size. Our solution is to create a local copy of all marked nodes in the shared memory, and apply list contraction on the CPU side using an efficient parallel list contraction algorithm [9, 28], and then directly splice out all marked nodes remotely in parallel. The steps are as follows:

(1) The CPU side constructs a hash table mapping remote pointers to local (shared-memory) pointers. For all marked nodes received by the CPU side, pointers are stored in the hash table for the local versions. We instantiate a new local node if the node does not exist in the hash table, and otherwise link existing local nodes. Note that for consecutive marked nodes in a linked list, the first *unmarked node* on the left and the right ends will also be copied and stored locally.
(2) Perform parallel randomized list contraction on the CPU side to splice out marked nodes in local linked lists.
(3) Remotely link the *unmarked nodes* to splice out marked nodes.

Theorem 4.5. *Batched* Delete *operations using a batch size of* $P \log^2 P$ *can be executed* whp *in* $O(\log^2 P)$ *IO time,* $O(\log^2 P)$ *PIM time, and* $O(\log P)$ *CPU depth. The execution performs* $O(P \log^2 P)$ *expected CPU work, and uses* $\Theta(P \log^2 P)$ whp *shared memory.*

Proof. Stage 1 takes $O(\log^2 P)$ IO time and $O(\log^2 P)$ PIM time *whp* for the exact same reason as stage 1 of PIM-balanced batch Insert. Stage 2 is dominated by the $O(P \log^2 P)$ expected CPU work and $O(\log P)$ *whp* depth for list contraction [9]. ◻

5 RANGE OPERATIONS

In this section, we turn our attention from point operations to range operations. We consider a general form of range operations, RangeOperation(*LKey*, *RKey*, Func), in which for every key-value pair (k, v) such that $LKey \le k \le RKey$, and Func is applied to the value v. If Func returns a value, that value is returned to the CPU side. For example, a read(v) function executes a simple range query and a fetch-and-add(v, Δ) function increments a value by Δ and returns the old value. Let K denote the number of key-value pairs in a query's range.

For simplicity in the analysis, we assume that the RangeOperation is a constant-size task (in words) and that Func takes constant PIM time. More complicated operations can be split into a range query returning the values, a function applied on the CPU side, and a range update that writes back the results. Also, note that we can extend function to allow for associative and commutative reduction functions such as counting the number of key-value pairs in the range.

We present two types of execution for range operations: (i) broadcasting the operation to all the PIM modules or (ii) following the skip list structure to access all nodes within the specified range.

5.1 Range Operations by Broadcasting

In this first type, the range operation task is broadcast to all the PIM modules, who execute the range operation locally. Note that the normal skip-list pointers (the solid pointers in Fig. 2) do not enable a purely local execution of a range operation. Instead, we use the *local leaf list* and *next-leaf* pointers introduced in Section 3.2 (the dashed pointers in Fig. 2).

Specifically, each PIM module determines the local successor of *LKey* and then follows its *local leaf list* (of increasing keys) to access all local key-value pairs in the range (i.e., all pairs with key $\le$ *RKey*), and applying Func to each. To find the local successor, each PIM module takes three steps:

(1) Search the upper part until reaching the rightmost upper-part leaf with $k \le LKey$.
(2) Take the *next-leaf* pointer to the local leaf list.
(3) Search through local leaves until find the successor.

The following theorem shows that this algorithm is PIM-balanced.

Theorem 5.1. *For* $K = \Omega(P \log P)$*, broadcasting-based range operations can be executed in* $O(1)$ *IO time and* $O(K/P + \log n)$ whp *PIM time. For range operations that return values, the values can be returned in* $O(K/P)$ whp *IO time. The algorithm uses* $O(1)$ *bulk-synchronous rounds.*

Proof. Broadcasting the task is an h-relation with $h = 1$. For step 1, each search path has $O(\log n)$ *whp* nodes in the upper part [17]. step 2 is constant PIM time. For step 3, a skip list leaf is an upper-part leaf with probability $(1/2)^{\log p} = 1/p$, so *whp* there are $O(P \log P)$ skip list leaves (i.e., key-value pairs) between every pair of adjacent upper-part leaves. Thus, by Lemma 2.1, the segment of a local leaf list between consecutive *next-leaf* pointers is $O(\log P)$ *whp* size, and hence the local successor can be found in $O(\log P)$ *whp* PIM time.

Also by Lemma 2.1, *whp* all PIM modules will have $\Theta(K/P)$ key-value pairs within the range. Applying Function takes constant PIM time per pair, for a total of $O(K/P)$ *whp* time. Likewise, a total of $O(K/P)$ *whp* IO time suffices to send any return values back to the CPU side. ◻

5.2 Range Operations by Tree Structure

The above type of range operation is wasteful for small ranges, as it involves all the PIM modules even when only a few contain any keys in the range. This motivates the second type, which uses the skip list tree structure instead.

Execution of a single operation. To execute a range operation by tree structure, we visit all nodes that may have a child in the range. Similar to the search path for a point operation, these nodes form a *search area*. Note that there are $O(K + \log n)$ *whp* ancestors for a range with K key-value pairs, so the search area is of size $O(K + \log n)$ *whp*, of which $O(K + \log P)$ *whp* nodes are in the lower part.

We call this algorithm the *naïve range search algorithm*. It marks all the leaves in the range, and if desired, labels them with their index within the range:

(1) Randomly send the operation to one PIM module, then search from root to leaf (crossing PIM module boundaries as needed) to mark all nodes in the *search area*.
(2) Compute a prefix sum of the marked nodes, via a leaf-to-root traversal calculating the number of marked leaves in each node's subtree, followed by a root-to-leaf traversal in which each node tells its children the number of marked nodes

before them. Return the total number of marked nodes to the CPU side.

The function can now be applied to each marked leaf by the respective PIM modules, or (value, index) pairs can be sent to the CPU side and the function applied there. We can use the index to sort elements to save CPU time if needed.

For a single range operation, the naïve range search algorithm requires $\Theta(K + \log P)$ IO messages *whp* to/from random PIM modules. It also requires $\Theta(K + \log n)$ total PIM work *whp*, of which $\Theta(K/P+\log n)$ is at a selected random PIM module and $\Theta(K+\log P)$ is at random PIM modules.

PIM-balanced batch execution. To obtain PIM-balance, we batch $P \log^2 P$ range operations together. The minimum shared memory size M needed for this algorithm is $\Theta(Plog^2P)$.

Contention caused by batching range operations is similar to that in batched Successor operations, so we apply a similar technique:

(1) Split the batch ranges into at most $2P \log^2 P$ *disjoint* ascending subrange operations on the CPU side.
(2) Similarly to stage 1 in batched Successor operations: Select $P \log P$ pivot subranges, and do a divide-and-conquer on pivot operations to get the leftmost and rightmost search paths (in the lower part) for each pivot subrange.
(3) Similarly to stage 2 in batched Successor operations: Execute the naïve range search algorithm with the start point hint. After this stage, each leaf knows its index in the subrange, and the CPU side knows the size of each subrange.
(4) We compute the prefix sum of the subrange sizes in ascending order, and partition the subranges into groups such that the total subrange size for a group is $\Theta(P \log^2 P)$ and hence fits in the shared memory. This may entail splitting up a single large subrange into parts in order to fit. (Alternatively, we could apply the algorithm from §5.1 to all large ranges.) We execute the groups in ascending order, fetching in parallel all the key-value pairs (and indexes, if needed) in a group to the CPU side, and then executing in parallel each range operation that overlaps the group. Note that a single range operation may overlap multiple groups. RemoteWrite is used to write back any updated values.

The following shows that this algorithm is PIM-balanced.

Theorem 5.2. *Tree-structure-based range operations with batch size $P \log^2 P$ covering a total of $\kappa = \Omega(P \log P)$ key-value pairs can be executed in $O(\kappa/P + \log^3 P)$ IO time and $O((\kappa/P + \log^2 P) \log n)$ PIM time, both* whp.

Proof. The analysis follows from the analysis for Successor, with the added complexity of actually performing the range part. For $P \log^2 P$ operations covering κ key-value pairs, $O(\kappa/P)$ *whp* covered keys will reach the upper part. Thus, upper-part search takes $O(\kappa \log n/P)$ *whp* PIM time. As noted above, each lower-part search using the naïve range search algorithm over a subrange of size K involves $\Theta(K + \log P)$ work at random PIM modules. Thus, summed over all $2P \log^2 P$ subranges, we have $O(\kappa+P \log^3 P)$ work at random PIM modules, i.e., $O(\kappa/P + \log^3 P)$ *whp* PIM time by Lemma 2.1. Moreover, each group is $\Theta(P \log^2 P)$ keys at random PIM modules, so it is $\Theta(\log^2 P)$ *whp* PIM time per group, for a total of $O(\kappa/P)$ *whp* PIM time summed across all $\Theta(\kappa/(P \log^2 P))$ groups.

In terms of IO time, the naïve range search algorithm over a subrange of size K involves $\Theta(K + \log P)$ *whp* IO messages to/from random PIM modules. Summed over all subranges, we have $O(\kappa + P \log^3 P)$ IO messages to/from random PIM modules, which is $O(\kappa/P + \log^3 P)$ *whp* IO time. Moreover, processing each group is $\Theta(\log^2 P)$ *whp* IO time per group, for a total of $O(\kappa/P)$ *whp* IO time.

Adding the Successor PIM time and IO time to these bounds for the range part yields the theorem. □

6 CONCLUSION

This paper presented the *Processing-in-Memory (PIM)* model, for the design and analysis of parallel algorithms on systems providing processing-in-memory modules. The model combines aspects of both shared memory and distributed memory computing, making it an interesting model for algorithm design. We presented an efficient skip list for the model, supporting a wide range of batch-parallel operations. The data structure guarantees PIM-balance for any adversary-controlled batch of operations. Future work includes designing other algorithms for the PIM model and measuring the performance of our algorithms on emerging PIM systems.

Acknowledgments

This research was supported by NSF grants CCF-1910030, CCF-1919223, CCF-2028949, and CCF-2103483, VMware University Research Fund Award, the Parallel Data Lab (PDL) Consortium (Alibaba, Amazon, Datrium, Facebook, Google, Hewlett-Packard Enterprise, Hitachi, IBM, Intel, Microsoft, NetApp, Oracle, Salesforce, Samsung, Seagate, and TwoSigma), and National Key Research & Development Program of China (2020YFC1522702).

A APPENDIX

Proof of Lemma 2.2. Consider a weighted balls-in-bins game where we throw weighted balls into P bins. The total weight is W, and the maximum weight of any ball is $W/(P \log P)$. Let $S = W/P$. We show that the total weight of any bin is $O(S)$ *whp*. Our proof is similar to that of Lemma 2.1 from [4].

We first fix a bin b and show that the total weight of this bin is $O(S)$ *whp*. For each ball i, let X_i be the random variable that is w_i, the weight of ball i, if i is in bin b and 0 otherwise. Let $X = \sum_i X_i$, $S = E[X]$, $R = W/(P \log P)$, and $\sigma^2 = \sum_i E[X_i^2]$. By Bernstein's inequality [7], for any constant $c \geq 1$, we have:

$$\mathbb{P}(|X - E[X]| \geq cE[X]) = \exp\left(-\frac{c^2 E[X]^2/2}{\sigma^2 + RcE[X]/3}\right) = \exp\left(-\frac{c^2 S^2/2}{\sigma^2 + cRS/3}\right)$$

Now $\sigma^2 = \sum_i \frac{1}{P}\left(1 - \frac{1}{P}\right) w_i^2 \leq \frac{1}{P} \max\{w_i\} \sum_i w_i = RS$. Therefore:

$$\mathbb{P}(|X - E[X]| \geq c \cdot (2E[X])) \leq \exp\left(-\frac{(2c)^2 S^2/2}{RS(2c/3 + 1)}\right) \leq \exp(-c \log P) \leq (1/P^c)$$

as desired. Applying a union bound over all P bins completes the proof. □

REFERENCES

[1] Umut A. Acar, Daniel Anderson, Guy E. Blelloch, and Laxman Dhulipala. 2019. Parallel Batch-Dynamic Graph Connectivity. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. 381–392.

[2] Umut A. Acar, Guy E. Blelloch, and Robert D. Blumofe. 2002. The Data Locality of Work Stealing. *Theoretical Computer Science* 35, 3 (2002), 321–347.

[3] Nimar S. Arora, Robert D. Blumofe, and C. Greg Plaxton. 2001. Thread Scheduling for Multiprogrammed Multiprocessors. *Theory of Computing Systems* 34, 2 (2001), 115–144.

[4] Soheil Behnezhad, Laxman Dhulipala, Hossein Esfandiari, Jakub Łącki, Vahab Mirrokni, and Warren Schudy. 2019. Massively Parallel Computation via Remote Memory Access. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. 59–68.

[5] Michael A. Bender, Jonathan W. Berry, Simon D. Hammond, K. Scott Hemmert, Samuel McCauley, Branden Moore, Benjamin Moseley, Cynthia A. Phillips, David S. Resnick, and Arun Rodrigues. 2017. Two-level Main Memory Co-design: Multi-threaded Algorithmic Primitives, Analysis, and Simulation. *J. Parallel Distributed Comput.* 102 (2017), 213–228.

[6] Petra Berenbrink, Tom Friedetzky, Zengjian Hu, and Russell Martin. 2008. On Weighted Balls-into-bins Games. *Theoretical Computer Science* 409, 3 (2008), 511–520.

[7] Sergei Bernstein. 1924. On a modification of Chebyshev's inequality and of the error formula of Laplace. *Ann. Sci. Inst. Sav. Ukraine, Sect. Math* 1, 4 (1924), 38–49.

[8] Guy E. Blelloch, Daniel Ferizovic, and Yihan Sun. 2016. Just Join for Parallel Ordered Sets. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. 253–264.

[9] Guy E. Blelloch, Jeremy T. Fineman, Yan Gu, and Yihan Sun. 2020. Optimal Parallel Algorithms in the Binary-Forking Model. *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)* (2020), 89–102.

[10] Robert D. Blumofe and Charles E. Leiserson. 1999. Scheduling Multithreaded Computations by Work Stealing. *J. ACM* 46, 5 (1999).

[11] Jiwon Choe, Amy Huang, Tali Moreshet, Maurice Herlihy, and R. Iris Bahar. 2019. Concurrent Data Structures with Near-Data-Processing: an Architecture-Aware Implementation. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. 297–308.

[12] Rathish Das, Kunal Agrawal, Michael A. Bender, Jonathan W. Berry, Benjamin Moseley, and Cynthia A. Phillips. 2020. How to Manage High-Bandwidth Memory Automatically. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. 187–199.

[13] Xiaojun Dong, Yan Gu, Yihan Sun, and Yunming Zhang. 2021. Efficient Stepping Algorithms and Implementations for Parallel Shortest Paths. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*.

[14] Alexandros V. Gerbessiotis, Constantinos J. Siniolakis, and Alexandre Tiskin. 1997. Parallel Priority Queue and List Contraction: The BSP Approach. In *European Conference on Parallel Processing (Euro-Par)*. Springer, 409–416.

[15] Phillip B. Gibbons, Yossi Matias, and Vijaya Ramachandran. 1999. The Queue-Read Queue-Write PRAM Model: Accounting for Contention in Parallel Algorithms. *SIAM J. on Computing* 28, 2 (1999), 3–29.

[16] Michael T. Goodrich, Daniel S. Hirschberg, Michael Mitzenmacher, and Justin Thaler. 2011. Fully De-amortized Cuckoo Hashing for Cache-oblivious Dictionaries and Multimaps. *arXiv preprint arXiv:1107.4378* (2011).

[17] Michael T. Goodrich and Roberto Tamassia. 2015. *Algorithm Design and Applications*. Wiley Hoboken.

[18] Yan Gu, Julian Shun, Yihan Sun, and Guy E. Blelloch. 2015. A Top-Down Parallel Semisort. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. 24–34.

[19] Zhiyu Liu, Irina Calciu, Maurice Herlihy, and Onur Mutlu. 2017. Concurrent Data Structures for Near-memory Computing. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. 235–245.

[20] Gary L. Miller and John H. Reif. 1985. Parallel Tree Contraction and Its Application. In *IEEE Symposium on Foundations of Computer Science (FOCS)*. 478–489.

[21] Onur Mutlu, Saugata Ghose, Juan Gómez-Luna, and Rachata Ausavarungnirun. 2020. A Modern Primer on Processing in Memory. *CoRR* abs/2012.03112 (2020).

[22] Jürg Nievergelt and Edward M. Reingold. 1973. Binary Search Trees of Bounded Balance. *SIAM J. Comput.* 2, 1 (1973), 33–43.

[23] Thomas Papadakis. 1993. *Skip Lists and Probabilistic Analysis of Algorithms*. University of Waterloo Ph. D. Dissertation.

[24] Wolfgang J. Paul, Uzi Vishkin, and Hubert Wagener. 1983. Parallel Dictionaries in 2-3 Trees. In *Intl. Colloq. on Automata, Languages and Programming (ICALP)*. 597–609.

[25] William Pugh. 1990. Skip Lists: A Probabilistic Alternative to Balanced Trees. *Commun. ACM* 33, 6 (1990), 668–676.

[26] Martin Raab and Angelika Steger. 1998. "Balls into bins"–A simple and tight analysis. In *International Workshop on Randomization and Approximation Techniques in Computer Science*. Springer, 159–170.

[27] Peter Sanders. 1996. On the Competitive Analysis of Randomized Static Load Balancing. In *Workshop on Randomized Parallel Algorithms (RANDOM)*.

[28] Julian Shun, Yan Gu, Guy E. Blelloch, Jeremy T. Fineman, and Phillip B. Gibbons. 2015. Sequential Random Permutation, List Contraction and Tree Contraction are Highly Parallel. In *ACM-SIAM Symposium on Discrete Algorithms (SODA)*. 431–448.

[29] Harold S. Stone. 1970. A Logic-in-Memory Computer. *IEEE Trans. Comput.* C-19, 1 (1970), 73–78.

[30] Yihan Sun, Daniel Ferizovic, and Guy E. Blelloch. 2018. PAM: Parallel Augmented Maps. In *ACM Symposium on Principles and Practice of Parallel Programming (PPOPP)*. 290–304.

[31] Thomas Tseng, Laxman Dhulipala, and Guy E. Blelloch. 2019. Batch-parallel Euler Tour trees. In *SIAM Meeting on Algorithm Engineering and Experiments (ALENEX)*. 92–106.

[32] Leslie G. Valiant. 1990. A Bridging Model for Parallel Computation. *Commun. ACM* 33, 8 (1990), 103–111.

[33] Yiqiu Wang, Shangdi Yu, Yan Gu, and Julian Shun. 2021. A Parallel Batch-Dynamic Data Structure for the Closest Pair Problem. In *ACM Symposium on Computational Geometry (SoCG)*.

[34] Tobias Ziegler, Sumukha Tumkur Vani, Carsten Binnig, Rodrigo Fonseca, and Tim Kraska. 2019. Designing Distributed Tree-based Index Structures for Fast RDMA-capable Networks. In *ACM SIGMOD International Conference on Management of Data*. 741–758.