Non-Clairvoyant Scheduling for Processing-in-Memory

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

Yiwei Zhao
Carnegie Mellon University
Pittsburgh, PA, USA
yiweiz3@andrew.cmu.edu

Kunal Agrawal
Washington U. in St. Louis
St. Louis, MO, USA
kunal@wustl.edu

Yongwei Wu
Tsinghua University
Beijing, China
wuyw@tsinghua.edu.cn

Phillip B. Gibbons
Carnegie Mellon University
Pittsburgh, PA, USA
gibbons@cs.cmu.edu

Abstract

Processing-in-memory (PIM) is a promising architectural approach to mitigate the high cost of off-chip memory access by enabling (i) low-latency, on-memory-module data access and (ii) aggregate memory bandwidth that scales with the number of modules.

To fully exploit the potential of PIM systems, we formulate and study the *PIM Scheduling* problem, which captures the trade-offs between computation, data movement, and load balancing across a host CPU and multiple PIM modules. We establish fundamental lower bounds on the execution time of any schedule. We then design a non-preemptive clairvoyant algorithm that achieves a constant-factor approximation to the optimal schedule. More importantly, we develop a non-clairvoyant scheduling algorithm that does not know task work in advance, yet loses only a small additive term relative to the clairvoyant lower bound. Besides scheduling based on fixed data placement, we also prove a performance upper bound under uniform random data placement.

We evaluate our scheduling algorithms on both an analytical PIM simulator and a real-world 2048-module UPMEM-PIM machine. Both algorithms outperform baselines across various workload settings and hardware regimes, with up to 9.8× speedup in simulation and up to 1.8× speedup on real PIM hardware.

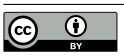
CCS Concepts

• **Theory of computation → Scheduling algorithms**; **Online algorithms**; • **Computer systems organization → Heterogeneous (hybrid) systems**.

Keywords

processing-in-memory, near-data-processing, scheduling

ACM Reference Format:
Hongbo Kang, Yiwei Zhao, Kunal Agrawal, Yongwei Wu, and Phillip B. Gibbons. 2026. Non-Clairvoyant Scheduling for Processing-in-Memory. In *38th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA '26), July 6–10, 2026, London, United Kingdom.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3816782.3819200



SPAA '26, London, United Kingdom

ACM ISBN 979-8-4007-2761-0/26/07
https://doi.org/10.1145/3816782.3819200

1 Introduction

The growing disparity between CPU performance and memory speed (a.k.a. the *Memory Wall*) has made off-chip data movement the dominant cost in data-intensive applications. Such data movement incurs substantial energy overhead, and system performance is limited by high memory latency and limited memory bandwidth.

Processing-in-memory (PIM), a.k.a. near-data-processing (NDP), has recently reemerged as a promising architectural solution to the memory wall problem. Typically, a PIM system is comprised of a powerful host CPU and a collection of PIM modules, where lightweight compute units (PIM cores) are directly integrated into memory modules. In traditional von Neumann architectures, computation requires data to be transferred between memory and CPU over off-chip channels, incurring significant data movement overhead. In contrast, PIM systems allow computation to be offloaded to PIM cores, which are closer to where data reside. This approach improves both performance and energy efficiency by exploiting low-latency, low-energy memory accesses within each PIM module, as well as aggregate memory bandwidth and compute capacity that scale with the number of modules. By offloading computation closer to data, PIM substantially reduces off-chip data movement, leading to lower energy consumption.

To fully exploit the potential of PIM systems, we introduce and study a new scheduling problem arising from PIM systems, which we call the ***PIM Scheduling*** problem. Consider a set of n data objects stored on a collection of P servers (PIM modules) plus a designated master server (host CPU). A batch of m tasks arrive at the master, where each task requests access to an object. Unlike prior scheduling work in which the master server merely routes (***Pushes***) each task to a server holding the task's requested object, in the PIM Scheduling problem, the master can alternatively ***Pull*** the object back to the master and execute the task at the master. Compared to prior scheduling problems, this provides an alternative means to balance the load across the servers, especially given the key (realistic) assumption in PIM Scheduling that the master server (the host CPU) has significantly higher computing power than other servers (the PIM cores).

However, the Pull is not free. Because the motivation for PIM systems is to reduce data movement, algorithms designed for PIM account for not just the work performed by each server but also its data movement costs. Thus, a key aspect of the load on a server is the time spent sending and receiving data, such as when fetching an object as part of a Pull. The overall goal is to minimize the makespan

of the schedule, accounting for the load on the servers (the time for executing tasks and exchanging data with the master) and the time for executing tasks on the master.

Our recent work on PIM algorithms for ordered [20, 22], string [21], and spatial [38, 39] indexes shows that judicious use of both Push and Pull in PIM systems enables both load balance and asymptotic reductions in data movement, leading to better throughput and energy efficiency, both in theory and practice. In this paper, we consider a more general PIM Scheduling problem beyond the index-specific techniques and particular problem settings studied by these works.

Our primary setting studies the case where the storage of objects on servers is fixed in advance (although we can copy objects between the master server and the other servers, as part of our schedule). Each task is characterized by the size of its task descriptor, the size of its reply, its target object, and the work needed to execute the task on the object. Each object is characterized by its size and the host server where it is stored. Together with the bandwidths and processing rates of the system, these quantities define the cost of Push vs. Pull, and by extension, the makespan of a given schedule. Note as well that an object that is pulled to the master can be used by *all* tasks with that target object, so scheduling decisions should be made based on the entire batch of tasks.

We first study a ***clairvoyant*** setting, in which all task and object properties are known to the scheduler. We identify unavoidable bottlenecks that lower bound the makespan of any schedule, including the optimal clairvoyant schedule: the maximum per-PIM communication/execution load, the total system load, and the time needed by the *hardest* individual task (i.e., the task with highest min(Push cost, Pull cost)). These lower-bound observations are simple; the main technical challenge is to design schedules that satisfy these bottlenecks simultaneously despite asymmetric CPU/PIM compute power, bounded CPU-PIM channels, and, later, unknown task work. We present a non-preemptive scheduling algorithm that achieves a constant-approximation to the *optimal* clairvoyant schedule. For settings in which the master server's *processing speed* (i.e., work per unit time) is $\Omega(P)$ higher than any other server (called the *Strong-CPU* setting), Push and Pull suffice to achieve the constant-approximation. On the other hand, when the master server is asymptotically weaker than all other servers combined (the *Weak-CPU* setting), we add a third option ***Redirect*** that migrates an object from its host server to another server (via the master server), in order to obtain constant-approximation. In clairvoyant scheduling, the main challenge is the trade-off between load balance and communication, when the scheduler decides between Pull and Redirect for the Weak-CPU setting. To address this challenge, we formulate the decision process as an asymmetric parallel machine scheduling problem, and resolve the asymmetry using a greedy bin-packing algorithm.

Second, we study a ***non-clairvoyant*** setting, in which each task's work is unknown to the scheduler. We present a non-clairvoyant scheduling algorithm that achieves a constant-factor approximation to the optimal *clairvoyant* schedule under the Strong-CPU setting, again using only Push and Pull. For the Weak-CPU setting, our non-clairvoyant scheduling algorithm (using Redirect) differs asymptotically from the clairvoyant lower bound by just an additive term reflecting the highest Push cost for a *single* task in the batch; this is true even in the extreme case where the master and each non-master server have the *same* processing speed. A key challenge in the non-clairvoyant setting is that we only learn a task's work once the task has completed, making Push vs. Pull vs. Redirect decisions difficult. To deal with this challenge, our non-clairvoyant algorithm can preempt tasks, but we make the pessimistic (for our algorithm) assumption that a preempted task must completely start over if it is restarted on a different server. Two key techniques are introduced to solve the unknown-work problem: optimistic execution and exponentially-growing Redirect rounds. When the task work is needed to choose between two options, optimistic execution initially chooses the option with smaller known cost, and switches to the other option after only limited wasted work. Exponentially-growing Redirect rounds are designed for the execution of high-work tasks. They ensure that such tasks, despite being unforeseeable in the non-clairvoyant setting, do not break the load balance guarantees.

Beyond the scheduling problem, we also study the object placement problem. Here, we consider a uniform random mapping of objects to servers, and show an upper bound based on a new balls-into-bins lemma. We also give an adversarial argument showing that, among batch-oblivious placements, random placement is worst-case optimal up to the same asymptotic bound.

We implement and evaluate both our clairvoyant and non-clairvoyant scheduling algorithms on two complementary platforms: an analytical simulator we build for general PIM systems across different hardware regimes, and a real-world 2048-module UPMEM-PIM machine [29]. We compare both algorithms against baselines that either use all-push or all-pull strategies, or apply a minimize-total-load heuristic described in §3.4. Across various workload settings and hardware regimes, both algorithms consistently outperform all baselines. Overall, our clairvoyant algorithm achieves up to 9.8× speedup in simulation and up to 1.8× speedup on real PIM hardware, while our non-clairvoyant algorithm achieves up to 9.7× and 1.4× speedup, respectively.

In summary, the main contributions of this paper are:

- We formalize the *PIM Scheduling problem*, which captures the asymmetric compute power, CPU-mediated communication, and the Push/Pull/Redirect execution model in PIM systems.
- We derive *lower bounds* on execution time that capture per-PIM load balance, total system load, and the hardest individual task.
- We design a non-preemptive *clairvoyant* algorithm achieving a constant-factor competitive ratio to the optimal schedule.
- We design a *non-clairvoyant* algorithm whose execution time matches the lower bound up to an additive term corresponding to the highest single-task Push cost.
- We present an upper bound on execution time under random initial *object placement*, and provide an adversarial strategy for other object placement algorithms.
- We evaluate our scheduling algorithms on an analytical PIM-system simulator and on real-world UPMEM-PIM hardware [29], showing up to 9.8× and 1.8× speedups, respectively, over the best prior scheduler baselines.

Our code is at https://github.com/cmuparlay/PIM-Scheduling.

2 Background and Related Work

2.1 The PIM Model

In this paper, we use the Processing-in-Memory (PIM) Model [19] to analyze our scheduling algorithms. This model has proven effective for theoretical analysis [21, 38] while also faithfully capturing the behavior of real-world PIM systems [20, 22, 39].

Hardware Abstraction. A PIM system, as depicted in Fig. 1, consists of a host CPU and a PIM side with P PIM modules. The CPU follows a conventional multicore architecture with a shared on-chip cache of size M words. Each *PIM module* contains a small on-chip *local memory* of $O(N/P)$ words (where N denotes the problem size or total storage footprint), collectively forming the *PIM memory*, along with a general-purpose but relatively lightweight compute unit referred to as the *PIM core*. The host CPU can load programs onto PIM modules, trigger their execution, and detect their completion.

The host CPU can access both its on-chip cache and the off-chip memories of all PIM modules, whereas each PIM core is limited to its own local memory. Communication between the CPU and PIM modules occurs via CPU-initiated read and write operations on the memories of the PIM modules. Direct PIM-to-PIM communication is not supported; instead, data exchange across different PIM modules must be manipulated by CPU-managed memory operations.

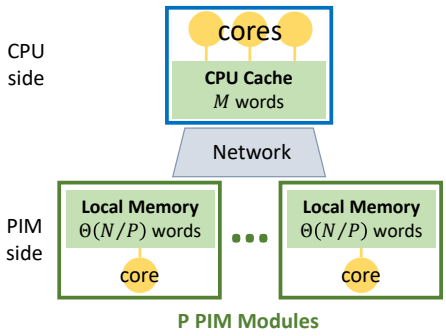


Figure 1: The Processing-in-Memory (PIM) Model [19].

Execution and Cost Metrics. We assume an execution model similar to the Bulk-Synchronous Parallel (BSP) paradigm [34], where computation proceeds in synchronous rounds that alternate between CPU execution, PIM execution (all PIM cores in parallel), and CPU-PIM data transfers (scatter-gather communication between the CPU and all the PIM modules). Each round incurs a distinct latency, and the overall execution time is the sum of these round costs. Formally, the total execution time T is decomposed into three components: the CPU execution time T_{CPU}, the PIM execution time T_{PIM}, and the communication time T_{comm}. This is an algorithmic abstraction rather than a dependence on any specific hardware generation, although current PIM systems also expose a compatible bulk host interface in which the CPU launches kernels on many PIM modules and later collects their results [20, 29].

2.2 Scheduling in PIM Systems

While there has been extensive work on scheduling for PIM systems, most prior work focuses on application-driven design, such as AI/ML [14, 26], nearest neighbor search [13, 37], graph processing [4, 15], and databases [20, 24]. Among the few efforts that target scheduling for general-purpose workloads (e.g., [12, 18, 23, 25, 30, 32, 33]), existing approaches are primarily ***empirical***. This gap motivates our development of a formal analysis of PIM Scheduling with complexity-theory and/or competitive-ratio guarantees.

As noted in §1, our recent work on PIM-optimized indexes [20–22, 38–40] shows that judicious use of both Push and Pull on PIM enables both load balance and asymptotic reductions in data movement (with high probability). However, that work differs from this paper by (*i*) focusing only on indexes, (*ii*) considering only the case where the CPU has at least P times the compute power of a PIM core, (*iii*) assuming all tasks in a batch are the same type of operation (e.g., all index lookups) and all objects are with a same fixed size, (*iv*) not providing competitive analysis, and (*v*) relying on algorithm-controlled data placement and replication.

2.3 Distributed Scheduling

The PIM Scheduling problem studied in this paper is similar to some problems studied in the context of distributed key-value stores, distributed file systems and distributed content placement (e.g., [2, 5, 6, 16, 17, 36]). In these systems, objects are distributed across many servers, often under the coordination of a master server. This architecture naturally maps to the PIM setting: servers correspond to PIM modules, while the master server corresponds to the host CPU. When a request to access an object arrives, the master routes the request to a server that stores the requested object. Most of these works are empirical in nature, but they show the importance of handling online requests that may be skewed toward certain keys/objects. Because these systems do not have the ability, as PIM does, to pull the object to the master and execute the task at the master, load balancing generally involves either moving objects from heavily-loaded servers to less-loaded servers, placing the object *a priori* based on some assumptions about the pattern of object arrival, or replicating the objects across servers.

Some recent theoretical work in this area considers randomized placement of objects for better load balancing [3, 35]. In particular, Wang et al. [35] uses techniques similar to our object placement results (§6) in that it applies random object placement and moves objects to improve load balance; however, due to the difference in models, the algorithm details and the analytical results are significantly different. Analysis of these randomized object placement schemes often use techniques developed in the context of the balls-into-bins problem [7]—here n balls are placed uniformly at random into m bins and the weight of the heaviest bin is analyzed. In PIM scheduling (and distributed data stores) with randomized placement, the order is reversed—the objects are first placed randomly across PIM modules (servers) and then the requests are routed to the appropriate server. However, similar analytical techniques apply.

One technique often used in balls-into-bins problems is *power of two choices* [27] where balls can choose two locations independently at random and go to the less loaded location. When all balls have the same weight, the power of two choices significantly improves load balance. In the context of PIM scheduling, this would mean that we copy each object to two locations *a priori* and when tasks arrive, they are routed to the less loaded PIM module. However,

Table 1: Notation used throughout the paper

System Parameters	
P	Number of PIM modules in the system, each with a CPU-PIM channel
ϕ	CPU processing speed relative to one PIM module
Task Properties	
m	Number of tasks in a batch
t_i	Target object of task i
q_i	Transfer time of task i's descriptor over a CPU-PIM channel
w_i	PIM execution time of task i (Note: CPU execution time = w_i/ϕ)
r_i	Transfer time of task i's reply over a CPU-PIM channel
Object Properties	
n	Number of objects stored in the system
h_k	Host PIM module ID of object k
s_k	Transfer time of object k over a CPU-PIM channel
Q_k	Total descriptor transfer time of tasks targeting object k in a batch
W_k	Total PIM execution time of tasks targeting object k in a batch
R_k	Total reply transfer time of tasks targeting object k in a batch

the weighted case (where balls may have weights, corresponding to the task sizes or communication costs) makes the analysis of this algorithm significantly more challenging. Prior work has only considered such analysis under stochastic assumptions and/or when the weights of balls are known in advance [8, 9, 31]. Therefore, these techniques do not directly apply in our context.

3 Problem Setup

Overview. In this paper, we study task scheduling in the PIM Model with P PIM modules and a host CPU. Our algorithm takes a task batch as input that arrives at the CPU cache, and aims at finishing the batch in minimum total execution time. Within a batch, each task i needs to perform some computation on a target object t_i. Each object is initially stored on a host PIM module. To run the task, the task descriptor and its target object must co-locate at some processing unit, either the CPU or any PIM module. Therefore, our algorithm must decide the best way to co-locate them, which is either (*i*) ***Push*** the task to the host PIM module of the object, (*ii*) ***Pull*** the object to the CPU, or (*iii*) ***Redirect***: pull the object, then send both the task and the object to a non-host PIM. We will also consider (in §6) a setting where we control the initial placement of objects across PIM modules, before batches arrive.

In this section, we start by formally defining the PIM Scheduling problem, by defining the parameters and operations. Then in §4 and §5, we will introduce our clairvoyant and non-clairvoyant scheduling algorithms, assuming no control over object placement. In §6, we will introduce and analyze our object placement strategy.

3.1 Problem Definition

Objects. There are n objects stored in the system. Object k is stored on exactly one PIM module, called its ***host PIM module***, with ID h_k. Moving object k between a PIM module and the CPU costs s_k ***object transfer time***.[1] Note that the CPU can do P transfers in parallel, one per PIM module. For each object, both h_k and s_k are known to both its host PIM module and the CPU. In §4 and §5, we assume the placement function h_k is fixed. In §6, the algorithm decides the placement function before any batches arrive.

Tasks. Tasks arrive at the CPU in batches of size m. We optimize the makespan of one batch at a time, and do not consider fully streaming arrivals where new tasks keep arriving while the current batch is still being scheduled. This batch-parallel setting matches the bulk execution interface abstracted in §2.1; dynamic arrivals can be handled by forming successive batches, but online admission across batches is out of the scope of this paper.

Task i operates on one ***target object*** t_i with ***PIM execution time*** w_i. The target object has host PIM module h_{t_i}, which we also call the ***host PIM module*** of the task. The descriptor of task i has transfer time q_i, and its reply has transfer time r_i. Under our non-clairvoyant setting, we make the realistic assumption that q_i, r_i, and t_i are known in advance by the scheduler, but the PIM execution time w_i is known only *after* the completion of task execution. We assume tasks are read-only, and they can be flexibly aborted and restarted. We also assume tasks can be paused and resumed, but only on the same processing unit (i.e., the same PIM module or the host CPU).

Note that a batch may include multiple tasks with the same target object. We denote the total PIM execution time of tasks targeting object k in a batch as W_k. Similarly, we denote the total descriptor (reply) transfer times as Q_k (R_k, respectively). Any object that is targeted by at least one task is called an ***active*** object for the batch.

Table 1 summarizes the notation used in this paper.

3.2 System Assumption

We assume that the system follows the PIM Model described in §2.1. While the original PIM model has separate metrics for CPU computation, PIM computation, and CPU-PIM communication, we need to combine these metrics into one for the total makespan.

We model the relative processing power of the CPU over each PIM core by a parameter $\phi \geq 1$. Thus a task i that would take w_i time if executed on a PIM module takes only w_i/ϕ time on the CPU. When the CPU is asymptotically at least as powerful as all PIM cores combined (i.e., $\phi = \Omega(P)$), we call this setting the ***Strong-CPU*** setting; otherwise, we call it the ***Weak-CPU*** setting. As for relative bandwidth, the CPU can perform as many as P transfers in parallel, one per PIM module. This models a balanced system (such as the UPMEM-PIM discussed in §7) where the bandwidth out/in the CPU side matches the bandwidth in/out the PIM side.

Load Balance. All P PIM modules and all P CPU-PIM communication channels run in parallel in PIM compute and CPU-PIM communication rounds, respectively. The execution time of each synchronous round is decided by the straggler. For both CPU-PIM communication rounds and PIM execution rounds, the execution time is defined as the execution time of the PIM module with maximum load. If a future PIM system allows CPU and PIM computation, or computation and communication, to overlap within a round, the

[1] To simplify notation, instead of defining object sizes in bytes and then dividing by the CPU-PIM bandwidth to get transfer times, we directly define (normalized) transfer times. Similarly, instead of defining task work and then dividing by processing speeds to get execution times, we will directly define (normalized) execution times.

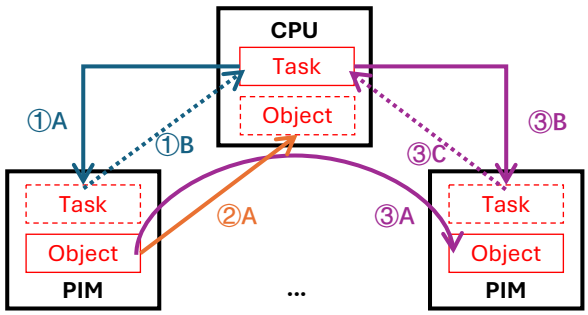


Figure 2: Single-task scheduling diagram showing data flows for ① Push, ② Pull and ③ Redirect. Solid (dashed) boxes show task and object locations prior to (during, respectively) scheduling.

round time changes from a sum of these components to their maximum, which affects our bounds only by a constant factor. For CPU execution, there is no load balance issue.[2]

Pause & Re-schedule. We assume that each PIM module can always pause the execution of its currently running task and switch to another task on it. However, we make the pessimistic (for our algorithm) assumption that the paused tasks can only be resumed at the exact same PIM module. Trying to execute the task at a different PIM module requires sending related data (the task descriptor and the object) and completely restarting the task execution. Allowing migration to preserve partial progress would define a stronger model, because the scheduler would be able to move only intermediate state and then execute only the *remaining* work on the new processing unit. Our lower bounds do not rely on this pessimistic restart-on-migration restriction. Exploiting partial progress would require different scheduling algorithms and is left for future work.

3.3 Execution Methods

Push. The most straightforward way to execute a task i is to Push the task to the object (① in Fig. 2). A typical use case for Push is a lightweight task (with short execution time, a small descriptor, and a small reply) with a huge target object (with large object transfer time). Sending the task descriptor (①A), executing the task, and receiving the reply (①B), takes q_i, w_i, and r_i time, respectively.

Pull. Another way to execute a task i is to Pull the object t_i, then run on the CPU. A typical use case is a heavyweight task (long execution time and/or a large descriptor/reply), or a collection of tasks with high aggregate cost, with a small target object (small object transfer time). In such case, fetching the object (②A) and running the task on the CPU is faster, and avoids incurring this high load on the object's host PIM module. Pulling the object and executing the task takes s_{t_i} and w_i/ϕ time, respectively. We do not need to send/receive the task descriptor/reply in this case.

Redirect. A natural extension to the Push and Pull studied in prior PIM papers (§2.2) is Redirect. It pulls an object to the CPU (③A), sends the object (③A) and the descriptor (③B) to another PIM to execute, and receives the reply (③C). In §5, we will show that we can achieve our bounds without Redirect, but only in the Strong-CPU setting ($\phi = \Omega(P)$). In the Weak-CPU setting, there is asymptotically greater compute power on the PIM side, and we should make use of this power rather than leaving the PIM cores underutilized. A Redirect for task i from its host PIM module h_{t_i} to a different PIM module h' incurs $2s_{t_i} + q_i$ time to co-locate task and object on h', w_i time to execute on h', and r_i time to send the reply back to the host CPU.

Merging Rounds. All three execution methods involve multiple rounds. For example, Pull needs a CPU-PIM communication round to fetch the object, and a CPU execution round to execute the task on the object. Based on §3.2, the time is the sum of separate maxes for each round. We simplify our analysis by analyzing the "max of sums" instead of the "sum of maxes", at the cost of a constant factor (since the number of rounds being so merged is constant). We define the ***load*** for a PIM module to be the combined costs in time. For example, a Push for task i contributes q_i load for sending the descriptor to its host PIM module, w_i load for execution on that PIM module, and r_i load for sending the reply back to the CPU, adding up to $q_i + w_i + r_i$ total load for its host PIM module h_{t_i}.

3.4 Main Challenge

Since the PIM model is a distributed model with a CPU and P PIM modules, we need both low total load and good load balance to achieve low total execution time. The main challenge is to achieve both simultaneously, especially under the non-clairvoyant setting.

If we consider only the total load and ignore load balance, then there is a trivial solution, which we call *minimize-total-load* (MTL): Execute each task exactly once, never redirect, and choose between push and pull by comparing Q_k, R_k, W_k and s_k. This minimizes the total load as it simultaneously minimizes execution and transfer costs. However, it can cause severe load imbalance—it may accumulate all tasks on the host CPU, leaving the PIM side underutilized, which is unacceptably inefficient in the Weak-CPU setting.

3.5 Lower Bound

In this section, we present a per-instance (i.e., batch-specific) lower bound on the execution time of any schedule for a given batch of tasks. Note that the proof for this lower bound uses the "merging rounds" technique (§3.3). The three terms below are elementary bottlenecks when viewed separately: one captures the heaviest load forced onto a single host PIM, one captures the total work that must be absorbed by the CPU/PIM system even with perfect load balance, and one captures the time needed by the hardest individual task even if it is routed optimally. Their role in the paper is as a target for the algorithms in §4–§5: the algorithms must simultaneously respect these bottlenecks while deciding when to Push, Pull, or Redirect.

Lower Bound per PIM. Every task needs to access one object on exactly one PIM. Therefore, for every PIM module j, it must deal with (execute or move) all tasks i with host PIM module j (i.e., $h_{t_i} = j$). This lower bound enumerates all active objects with tasks on the PIM. For each object k, either all its tasks are executed locally on the PIM (push), or the object itself is pulled to the CPU side (pull or redirect). Also, there must be sufficient time for the

[2] As argued in [19], work-stealing [1, 10, 11] can be used across host CPU cores to obtain load balance since data are only moved locally within the CPU's on-chip cache, whereas work-stealing *across* PIM modules incurs costly between-chip data movement.

last PIM module to finish. Recall that Q_k, W_k, R_k denote aggregate task costs for object k. We define:

$$T_{PerPIM} = \max_j \left\{ \sum_{h_k=j} \min(Q_k + W_k + R_k, s_k) \right\}$$

Lower Bound on Total Load. Every task must be executed somewhere in the system. In this lower bound, we assume perfect load balance throughout the system. It can either be executed on a single PIM, causing $q_i + w_i + r_i$ load for the PIM, which is $(q_i + w_i + r_i)/P$ in total for the system with P PIMs; or executed on the CPU, causing w_i/ϕ load. We define:

$$T_{Load} = \sum_i \min\{(q_i + w_i + r_i)/P, w_i/\phi\}$$

Lower Bound on the Hardest Single Task. Finally, each individual task must complete on either the CPU or a PIM module. If task i is executed on a PIM module, even ignoring any object movement needed for Redirect, it needs at least $q_i + w_i + r_i$ time for descriptor transfer, execution, and reply transfer. If it is executed on the CPU, its target object must be pulled to the CPU before the task can run there, taking at least $s_{t_i} + w_i/\phi$ time. Therefore, we define the single-task lower bound:

$$T_{Single} = \max_i \min\left\{q_i + w_i + r_i, s_{t_i} + w_i/\phi\right\}$$

Theorem 3.1. *$\Omega(T_{PerPIM} + T_{Load} + T_{Single})$ is a lower bound on the execution time, regardless of the schedule.*

4 Clairvoyant Scheduling

In this section, we introduce our clairvoyant scheduling algorithm. The algorithm assumes that when the task batch arrives at the CPU, the scheduler knows in advance the target object t_i, the task descriptor transfer time q_i, the PIM execution time w_i, and the reply transfer time r_i for each task i in the batch.

Theorem 4.1. *Our clairvoyant scheduling algorithm achieves constant-approximation on makespan compared to the optimal clairvoyant scheduling algorithm.*

The clairvoyant scheduler possesses complete knowledge of all tasks and objects, allowing it to internally simulate the entire execution process before the actual execution. Note that to finish all tasks within a batch, each task must be fully executed once, which requires the co-location of the task descriptor and the object. Our scheduler executes each task exactly once in the system, incurring no restarts or migrations.

As discussed in §3.3, each task is executed on the CPU side or on one PIM module, through Push, Pull, or Redirect (Fig. 2). The key ***scheduling decision*** is the physical execution location of each task, and the core functionality of the clairvoyant scheduler is to make these decisions. For task i, the scheduling decision is defined by where it was executed, denoted as $x_i \in \{CPU, PIM_j\}$. Before the execution begins, the CPU internally calculates all scheduling decisions, then the execution follows the outcome decisions throughout the execution process.

4.1 Execution Process

The execution has five rounds, as shown in Figure 3. First, the CPU computes the scheduling decision x_i for each task. Second, in a communication round, all task descriptors and objects are transferred according to scheduling decisions—those not already at the scheduled location are moved to enable execution. Third, in the execution round, all PIM modules execute all tasks assigned to them, which is guaranteed to be executable after the communication round. Fourth, the CPU executes all tasks assigned to itself. Fifth, in the last communication round, task replies are sent to the CPU.

4.2 Cost Analysis

The whole execution process, including all data transfers and object movements, is decided by the scheduling decisions x_i's. An optimal scheduling algorithm leads to decisions that minimize the total execution time. In this subsection, we formalize the execution time according to the scheduling decision. The scheduling algorithm that makes the decisions will be introduced in the subsections that follow.

Merging Rounds. Recall that the total execution time is the combination of the execution time of each round, which is decided by the straggler for PIM-involved rounds. Therefore, the total execution time is a combination of the preparation communication time (Round 2), the PIM execution time (Round 3), the CPU execution time (Round 4), and the reply communication time (Round 5), ignoring the cost of the scheduling algorithm.

$$T = T^{(2)}_{comm} + T_{PIMexec} + T_{CPUexec} + T^{(5)}_{comm}$$

Note that three rounds out of five are bounded by PIM stragglers. According to the merging rounds method mentioned in §3.3, we can ignore the synchronization barriers to simplify the analysis, without violating asymptotic bounds, and define *load* as the combined costs in time. Instead of directly minimizing T, we minimize T', which is decided by the maximum load among PIMs:

$$T' = \max_i \left\{T^{(2)}_{comm,i} + T_{PIMexec,i} + T^{(5)}_{comm,i}\right\} + T_{CPUexec} = \Theta(T)$$

Push. Task i is executed by Push to its host PIM module when its scheduling decision is $x_i = PIM_{h_{t_i}}$. It introduces $q_i + w_i + r_i$ load to the PIM module $PIM_{h_{t_i}}$.

Pull. Task i is executed by Pull when its scheduling decision is $x_i = CPU$. It introduces s_{t_i} load to its host module $PIM_{h_{t_i}}$, and w_i/ϕ load to CPU execution.

Redirect. For task i, when $x_i = PIM_j \neq PIM_{h_{t_i}}$, the task is executed on PIM_j through Redirect. It introduces s_{t_i} load to its host $PIM_{h_{t_i}}$, and $s_{t_i} + q_i + w_i + r_i$ load to its execution PIM module $x_i = PIM_j$.

4.3 Scheduling Decisions: Push

We now introduce the scheduling algorithm that decides x_i for all tasks, and prove that its makespan is constant competitive compared with the optimal clairvoyant scheduler with decision set x^*_i.

Push tasks. The first stage is to figure out tasks that must be executed on the host PIM module through Push with $x_i = PIM_{h_{t_i}}$.

To achieve this, the CPU reads the task batch, and analyzes all *active* objects (i.e., objects with one or more tasks targeting it). For

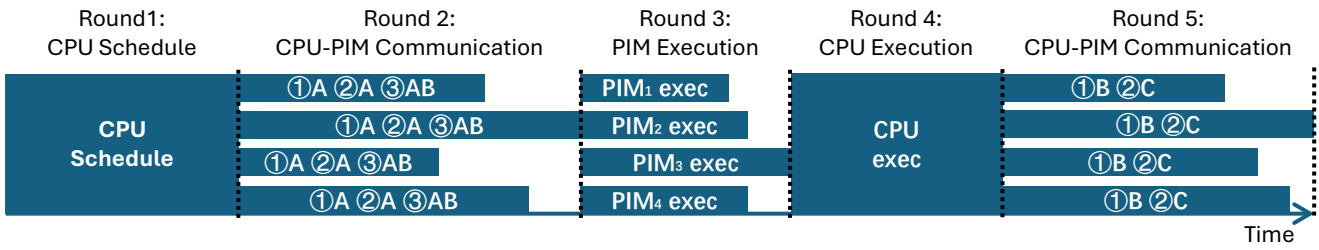


Figure 3: Overview of Clairvoyant Optimal Scheduling. The circled numbers and the letters refer to Fig. 2.

each active object k, we sum the total descriptor transfer time Q_k, the total PIM execution time W_k, and the total reply transfer time R_k, and compare these costs to the object transfer time s_k. The CPU can compute all these values in the clairvoyant setting.

There are two choices for each active object k: we either solve all tasks on it by Push, or we must pull the object back to the CPU for the other two types of executions (Pull and Redirect). Solving by Push takes $Q_k + W_k + R_k$ time in total, while pulling the object takes s_k time at least. Therefore, for any object in which Push is less expensive than simply pulling the object, using Push to solve all tasks on the active object is the optimal solution.

To be more precise, for any object k with

$$Q_k + W_k + R_k \le s_k$$

any task i targeting this object ($t_i = k$) will be executed by Push ($x_i = PIM_{h_{t_i}}$). This decision is always optimal. For any task $x_i = PIM_{h_{t_i}}$, either we have $x_i^* = PIM_{h_{t_i}}$, or we can set $x_i^* = PIM_{h_{t_i}}$ without worsening the makespan of the optimal schedule x_i^*. However, the other side does not hold.

Pull objects. After the first stage, all unfinished tasks will be executed through Pull or Redirect by the scheduler. Therefore, in the second stage, all objects that are still active are pulled to the host CPU, incurring s_k load for object k's host PIM module h_k. After this stage, all active task descriptors and objects are on the host CPU.

Note that there is a corner case—some object may have all its tasks executed by Push in the optimal schedule, but falls into this category (Pull or Redirect) in our scheduling. This mismatch is possible, and can lead to additional object movement compared with the optimal schedule. However, the cost of moving the object is less than the cost of executing the Pushs; thus, this mismatch does not break the constant competitiveness.

4.4 Scheduling Decisions: Pull and Redirect

In the third stage, the scheduler needs to decide the actual execution location x_i for each task i, choosing between the CPU or PIMs.

Strong-CPU vs. Weak-CPU. This is where the Strong-CPU and Weak-CPU settings differ. When $\phi = \Omega(P)$, Redirect is not asymptotically necessary: since the CPU is asymptotically as powerful as all PIM modules combined, a would-be redirected task can instead be run on the CPU after its object has been pulled, while preserving the constant-factor bound. In the Weak-CPU regime, the CPU cannot absorb all such work, so Redirect is needed to use the aggregate compute power of the PIM modules. Rather than handling the two regimes with separate algorithms, we use one assignment formulation that permits both Pull and Redirect.

After the Push stage, the remaining decision can be viewed as an extended version of a Parallel Machine Scheduling Problem: there are up to m tasks within the system, and each task must be executed fully on one machine (the host CPU or one of the P PIM modules). If task i is executed on the CPU, it incurs w_i/ϕ CPU load. If it is executed on PIM_j, it incurs $q_i + w_i + r_i$ load on PIM_j, and an additional s_{t_i} load if it is the first task accessing object t_i on PIM_j. We aim at acquiring a scheduling that minimizes the sum of the maximum load among PIM modules and the host CPU load.

Problem Simplification. To transform the problem into a more standard version of Parallel Machine Scheduling (PMS), we apply two simplifications without violating the constant competitiveness.

First, we aim at minimizing the maximum load among the CPU and all PIMs, rather than minimizing the sum of the CPU load and the maximum of PIM load. According to the "merge round" result in §3.3, this simplification preserves constant competitiveness.

Second, we aim at eliminating the impact of object movement loads s_{t_i} in the asymptotic analysis of our algorithm. Note that if task i has $q_i + w_i + r_i \ge s_{t_i}$, we can ignore the communication cost, because the task parameters dominate the load. However, the Push stage (§4.3) only guarantees that the *total load* on each object satisfies such constraint: $Q_k + W_k + R_k \ge s_k$. The total load can be a summation over many small tasks, each having an execution load $q_i + w_i + r_i < s_{t_i}$. If these small tasks are scheduled separately to different PIMs, object movement load might dominate the cost.

To eliminate this cost, we merge small tasks into larger ***meta-tasks***. Small tasks with $q_i + w_i + r_i < s_{t_i}$ are completely eliminated after this merging process. A potential issue of this merging technique is in load balance, because tasks that were able to be scheduled separately are forced to be scheduled together after merging, leading to larger task granularity. However, we set limits to ensure that it cannot break the constant competitiveness. When we merge multiple tasks together, we can view this as merging all but the largest task (the one with maximum $q_i + w_i + r_i$) into that largest task. We ensure that the total load of the small tasks $\sum q_i' + w_i' + r_i' \le 2 \cdot s_{t_i}$, so the additional load generated through this merging process will not change the makespan by more than a constant.

After the simplifications, the problem is very similar to an asymmetric parallel machine scheduling. There are P slow machines (PIM modules) and one fast machine (host CPU) in the system, and we need to schedule up to m tasks on these machines, aiming at the minimum makespan. Task i runs faster on the fast machine (w_i/ϕ), and slower on the slow machines ($q_i + w_i + r_i$).

Scheduling Algorithm. The scheduling algorithm starts by a binary search on the makespan result T, and tries to find a feasible

scheduling under this constraint. The minimum feasible T is taken as our makespan result.

To verify the feasibility of T and extract the corresponding schedule, we start with scheduling tasks on the CPU. We first schedule on the CPU ($x_i \leftarrow CPU$) all tasks that cannot be scheduled on PIM ($q_i + w_i + r_i > T$). Then we try to schedule tasks that save the most to the CPU, by sorting all tasks according to their savings ratio $\frac{q_i+w_i+r_i}{w_i/\phi}$, and greedily assigning tasks with maximum saving ratio to the CPU, until no task can fit into the T time budget for the CPU.

Finally we assign tasks to PIMs. We sort tasks according to their load ($q_i + w_i + r_i$), then schedule tasks from largest load to smallest to PIMs. Tasks first go to the first PIM module, and switch to the second PIM when the next task does not fit into the first PIM without exceeding T. If all tasks are successfully scheduled, T is feasible and we have a valid schedule; otherwise we mark T as infeasible.

Proof. We prove that if the optimal solution incurs T^* execution time for the simplified PMS problem, our scheduling algorithm always succeeds with $T = 2T^*$.

We prove by contradiction. Suppose our algorithm marks $2T^*$ as infeasible, then there must be tasks left after filling up the last PIM module. Therefore, every PIM module must have more than $T/2 = T^*$ load. This means that compared with the optimal schedule with time T^*, our schedule with $T = 2T^*$ digests less load on the CPU side. Note that our CPU scheduling algorithm can be viewed as a greedy scheduling algorithm prioritizing the gain ratio. Therefore, we found a contradiction with the following knapsack lemma.

Lemma 4.2 ([28]). *Consider a 0-1 knapsack problem with capacity C. Let G be the solution obtained by sorting items in non-increasing value-to-size ratio and greedily packing items with capacity $2C$. Then the value of G is at least the optimal value for capacity C.*

5 Non-Clairvoyant Scheduling

In this section, we introduce our non-clairvoyant scheduling algorithm. The algorithm assumes that when the task batch arrives at the CPU, the scheduler knows in advance the target object t_i, the task descriptor transfer time q_i, and the reply transfer time r_i for each task i in the batch, but the PIM execution time w_i is unknown until the task finishes. Not knowing the task execution times fundamentally changes the algorithm design—the CPU can no longer simulate the execution process in advance and use methods like binary search.

Our non-clairvoyant algorithm runs in four stages, and has the following execution time upper bound:

$$O\left(T_{PerPIM} + T_{Load} + T_{Heaviest}\right)$$

where $T_{Heaviest} = \max_i\{q_i + w_i + r_i\}$ is the highest PIM-side load (i.e., Push cost) of a single task in the batch. The lower bound in Theorem 3.1 includes a single-task term T_{Single}, which takes the cheaper side between PIM and CPU execution for each task (i.e., min(Push cost, Pull cost)). Thus, $T_{Heaviest}$ should be viewed as the non-clairvoyant additive overhead in our analysis, not as a standalone lower bound. In this section, we will introduce each stage in order. During this process, we prove that each step satisfies the upper bound.

5.1 Push Stage

This stage is the only stage that we execute tasks by Push—sending task descriptors to their host PIM module for local execution. This stage has a similar target to the Push stage in the clairvoyant setting—aiming at finding objects k with

$$Q_k + W_k + R_k \le s_k$$

The main challenge is that W_k is unknown in the non-clairvoyant setting; we address it by executing each task for a short trial period.

In this stage, the CPU reads the task batch, and calculates the sum of the total task descriptor transfer time and the total reply transfer time ($Q_k + R_k$) on each object k, and compares the sum against the object transfer time s_k. For every object k, if $s_k \ge Q_k + R_k$, task descriptors are sent to their host PIM module h_k, then the PIM module runs these tasks for at most $s_k - Q_k - R_k$ time. Once all tasks on an object are finished, the PIM module starts running the next object immediately. In such case, the object is deactivated and the reply(s) will be returned to the CPU. Otherwise, the tasks are aborted and all results are deleted in order to avoid the communication cost.

After this stage, any object that remains active has $W_k \ge s_k - Q_k - R_k$. This stage finishes in no more than T_{PerPIM} time, because objects that remain active spend at most s_k time each, and deactivated objects spend $Q_k + W_k + R_k$ time each.

5.2 Pull Object Stage

In this stage, the CPU pulls all active objects from the PIM modules. These tasks are either executed by the CPU (Pull) or by a remote PIM (Redirect) after this stage.

After this stage, all task descriptors and active objects are on the CPU side, and we decide whether or not to Redirect in later stages. This stage finishes in no more than T_{PerPIM} time, because each object takes exactly s_k time, and each active object has $W_k \ge s_k - Q_k - R_k$ after the Push stage.

Similar to our clairvoyant scheduling, additional objects might be pulled to the CPU, compared with the optimal scheduling. However, these elements do not asymptotically change the bound.

5.3 CPU Execution Stage

This stage is the only stage that we execute tasks by Pull on the CPU. Recall from §3.5 that each task contributes at least $(q_i + r_i + w_i)/P$ system load if executed on the PIM side and w_i/ϕ CPU load if executed by Pull. In this stage, we finish all ***CPU-friendly*** tasks whose Pull execution cost is no more than the Redirect cost:

$$\frac{q_i + w_i + r_i}{P} \ge \frac{w_i}{\phi}$$

The stage then follows the same Strong-CPU vs. Weak-CPU comparison as the clairvoyant Pull/Redirect stage in §4.4.

Strong-CPU. In the Strong-CPU setting, the CPU has more compute power than all PIMs combined: $\phi = \Omega(P)$. Therefore, the cost of Pull is always asymptotically no larger than Redirect, and we do not need Redirect in this case. Under this assumption, the scheduling algorithm ends here with a better upper bound $O(T_{PerPIM} + T_{Load})$ that matches the lower bound up to constant competitiveness. This bound also covers the single-task term T_{Single}: for any task, the

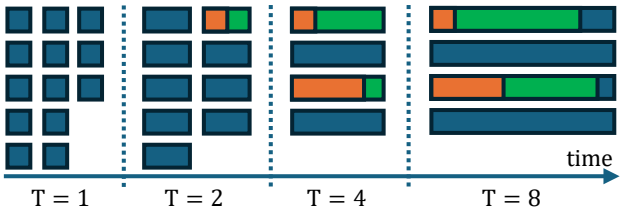


Figure 4: The execution scheme of Redirect.

object-transfer part is charged to T_{PerPIM} and the remaining per-task descriptor/execution/reply load is charged to either T_{PerPIM} or T_{Load}, depending on whether the task execution is cheaper on the PIM side or on the CPU side.

Weak-CPU. In the Weak-CPU setting, we have $\phi \geq 1$ and $\phi = o(P)$. We cannot simply ignore the computation power of all PIMs and run all tasks on the CPU as in the Strong-CPU setting. Instead, we run each task for time:

$$\frac{q_i + r_i}{P - \phi}$$

This guarantees that all CPU-friendly tasks finish. Note that this time is positive in the Weak-CPU setting since $\phi = o(P) < P$.

This stage finishes in no more than T_{Load} time on the CPU side, for both Strong-CPU and Weak-CPU settings.

5.4 Redirect Stage

Finally in this stage, we finish all remaining tasks by Redirect, which requires sending objects to PIMs. Therefore, similar to the clairvoyant scheduling step in §4.4, we need to merge tiny tasks into large meta-tasks. The only difference is that the PIM execution time w_i is unknown in the non-clairvoyant setting, therefore we merge small tasks with $q_i + r_i < s_{t_i}$, removing the w_i factor. After merging, every (meta-)task's load falls into the range $[q_i + r_i, \infty)$, whose exact value depends on the PIM execution time w_i.

The Redirect stage contains multiple rounds, as shown in Fig. 4. In this example, each rectangle is a task, there are five PIM modules (one per row) and four rounds (separated by dotted lines). During round x, the algorithm tries to Redirect all tasks onto PIM modules and spend $T = 2^{x-1}$ time on each task. Recall that we know the minimum load of each task $(q_i + r_i)$, so a task will not be considered if it definitely cannot be executed. Each execution slot is represented as a blue rectangle. The minimum load is marked in orange in this figure, while the unknown PIM execution time w_i is in green. A task may re-run multiple times before completion, but the total load remains linear to its actual load, since the lengths of time grow exponentially across rounds.

The execution time is bounded by $O(T_{Load} + T_{Heaviest})$. As shown in Fig. 4, there are full time slots (with five tasks) and partially-filled time slots. The total time for full time slots is bounded by $O(T_{Load})$ because of perfect load balance. The total time for partially-filled time slots is dominated by $O(T_{Heaviest})$, because the redirect stage finishes as soon as the heaviest task ends.

Theorem 5.1. *Our non-clairvoyant scheduling algorithm executes a task batch in $O(T_{PerPIM} + T_{Load} + T_{Heaviest})$ time.*

6 Object Placement

In prior sections, we studied task scheduling and data movement during execution, assuming that both the batch and the initial object placement are adversary controlled. As a result, the T_{PerPIM} term in the lower bound depends on the initial object placement:

$$T_{PerPIM} = \max_j \left\{ \sum_{h_k = j} \min(W_k + Q_k + R_k, s_k) \right\}$$

In this section, we study the initial object placement algorithm under a batch-oblivious setting: the placement is chosen before the task batch is known, and the scheduler then runs after the batch arrives. To be more specific, we study the uniform random placement strategy that stores every object at a random PIM module. This strategy is natural because it spreads the per-object terms in T_{PerPIM} across PIM modules without requiring any prediction of future access patterns. Our positive result is the high-probability bound stated in Theorem 6.1, obtained from Lemma 6.2. The adversary strategy at the end of the section gives the matching intuition: for any batch-oblivious placement, an adversarial batch can force the placement to behave no better than random in the worst case. For object k, we denote its placement-relevant object load by

$$L_k = \min(Q_k + W_k + R_k, s_k)$$

and let $L_{max} = \max_k L_k$.

Theorem 6.1 (Random object placement). *Assume that (i) each object is stored independently and uniformly at random on one of the P PIM modules and (ii) the task batch is generated obliviously to the random placement. Then, with probability at least $1 - 1/P$, the T_{PerPIM} term in the scheduling guarantees from §4 and §5 can be replaced by the placement-independent bound*

$$T'_{PerPIM} = O\left(\sum_k L_k/P + \sqrt{(\sum_k L_k^2) \log P/P} + L_{max} \log P \right)$$

Proof. We use the same scheduling algorithm after introducing the random object placement. Recall that the scheduling algorithm first decides which active objects can be finished by Push, then pulls the remaining active objects to the CPU, and only afterward runs the remaining tasks on the CPU (Pull) or on a remote PIM module (Redirect). The initial object placement affects only the first two stages: the PIM-side load incurred by Push executions, and the load for pulling active objects from their host PIM modules to the CPU. Once task descriptors and active objects have been pulled to CPU, the initial object placement no longer matters.

Every active object k contributes the object load L_k defined above to the first two stages. The "min" comes from the decision between Push and Pull. Under random object placement, each active object is stored uniformly randomly at one PIM module with probability $1/P$. Thus the placement-induced load in these two stages is exactly a balls-into-bins process with one ball for each active object, ball weight L_k, and one bin for each PIM module. Applying Lemma 6.2 gives the stated high-probability upper bound on the maximum PIM load, which is T'_{PerPIM}. □

Lemma 6.2. *We have b balls such that ball i has weight $a_i > 0$, and let $a_{\max} = \max_i a_i$. Each ball is placed into one of P bins independently uniformly at random. The maximum load of any bin is*

$$O\Big(\frac{\sum_{i=1}^{b} a_i}{P} + \sqrt{\frac{\sum_{i=1}^{b} a_i^2}{P}\log P} + a_{\max}\log P\Big)$$

with probability at least $1 - 1/P$.

Proof. Say Y_j is the load of bin j. Let

$$\mu = \mathbb{E}[Y_j] = \sum_{i=1}^{b} a_i/P$$

be the expected load for bin j. In addition, say $\sigma^2 = (1/P)\sum_{i=1}^{b} a_i^2$ is the variance (assuming reasonably large P).[3] We can now apply Bernstein's Inequality on Y_j:

$$\Pr(Y_j - \mu \geq \epsilon) \leq \exp\Big(\frac{-\epsilon^2}{2\sigma^2 + (2/3)\epsilon a_{\max}}\Big)$$

.

Suppose we want this probability to be less than $1/P^2$. Therefore, we want to solve for ϵ such that

$$\exp\Big(\frac{-\epsilon^2}{2\sigma^2 + (2/3)\epsilon a_{\max}}\Big) = 1/P^2$$

Taking a logarithm on both sides, we get a quadratic equation in ϵ. We can solve this equation to get $\epsilon = O(\sigma\sqrt{\log P} + a_{\max}\log P)$. Therefore, each individual bin has maximum load of

$$\frac{\sum_{i=1}^{b} a_i}{P} + O\Big(\sqrt{\frac{\sum_{i=1}^{b} a_i^2}{P}\log P} + a_{\max}\log P\Big)$$

with probability at least $1 - 1/P^2$. Applying the union bound across all P bins yields the stated result. □

Adversary Strategy. We now show why this balls-into-bins guarantee is the right target for batch-oblivious placement. The adversary knows the placement strategy, but not its random coins, and chooses a batch after the placement strategy has been fixed. The adversarial batches make Push no more expensive than moving the object for every active object, so the lower-bound term reduces to the maximum load of active objects on their initial host PIM modules.

Theorem 6.3 (Batch-oblivious placement lower bound). *For any batch-oblivious placement strategy, possibly randomized, there exists an adversarial distribution over task batches for which the placement-dependent lower bound matches the balls-into-bins load-balance barrier. Thus no batch-oblivious placement strategy can asymptotically improve the worst-case placement-dependent term beyond the high-probability guarantee achieved by uniform random placement.*

Proof. Fix any number b of active objects and any common load $\ell < \min_k s_k$. Fix a realized placement function h and a set A of active objects. For each object $k \in A$, let the adversary choose a value $\ell_k \in (0, s_k]$, which will become the object load L_k in the constructed batch. Create one task targeting each object $k \in A$ with total descriptor, PIM execution, and reply load equal to ℓ_k; for example, set q_i and r_i to arbitrarily small positive values with $q_i + r_i < \ell_k$, and set $w_i = \ell_k - q_i - r_i$. Then $Q_k + W_k + R_k = \ell_k \leq s_k$, so executing all tasks on object k by Push costs no more than pulling object k to the CPU, and by the definition of L_k we have $L_k = \ell_k$. For this batch, the placement-dependent term in the lower bound from §3.5 is

$$T_{PerPIM} = \max_j \sum_{k\in A: h_k = j} \min(Q_k + W_k + R_k, s_k) = \max_j \sum_{k\in A: h_k = j} \ell_k.$$

By Theorem 3.1, every schedule has makespan $\Omega(T_{PerPIM})$.

The construction above works for any chosen active set A; we now choose A adversarially to obtain the lower bound for batch-oblivious placement. Consider an adversarial distribution that selects b active objects uniformly at random from all objects and sets $\ell_k = \ell$ for every selected object. Conditioned on any balanced placement, where each PIM module stores the same number of objects, the induced host-PIM loads have the same distribution as placing equal-weight balls uniformly at random into P bins. If a placement is unbalanced, the adversary can instead choose active objects from an overfull PIM module, so imbalance only makes the worst-case batch easier to construct. Any high-load event that occurs with positive probability under the distribution above implies the existence of at least one fixed batch in its support with the same placement-dependent lower bound. □

[3] To be precise, the variance is $(1 - 1/P)(1/P)\sum_{i=1}^{b} a_i^2$, but we only need an upper bound in this analysis.

7 Evaluation

Our evaluation studies the effectiveness and robustness of our scheduling algorithms across both modeled and real-world PIM settings. Because there are no prior algorithms for the PIM Scheduling problem, we compare against natural but naïve baselines. We focus on the performance gains of our clairvoyant and non-clairvoyant algorithms over these baselines, as well as the robustness of these gains to different hardware regimes and workload settings. For this evaluation, we use both an analytical PIM simulator and a real-world 2048-module UPMEM-PIM machine [29].

Simulator. We build a simulator that serves as an analytical evaluator for the BSP cost model: given a workload and a schedule, it simulates CPU-PIM data transfer time, PIM execution time, and host CPU execution time. We sweep various configurations of hardware parameters to simulate different PIM system regimes.

UPMEM. We also evaluate on a server equipped with UPMEM PIM [29]. The server has two Intel Xeon Silver 4215 CPUs (32 threads total, 2.5 GHz, 22 MB L3 cache). Meanwhile, the server includes 32 UPMEM PIM ranks (2048 PIM modules) in total, providing 128 GB of PIM memory, with PIM cores running at 350 MHz.

Workload. On both platforms, each batch consists of m tasks targeting n objects. Tasks target objects according to a Zipf distribution [41] with parameter θ. By default, all tasks perform constant amounts of computation work, but we also evaluate settings where the per-task computation work follows exponential and Pareto distributions.

Baselines. We compare our two scheduling algorithms—Clairvoyant (*Clv*) and Non-clairvoyant (*NCV*)—against the following four baselines. *AllPush* sends every task to the PIM module that holds its

Table 2: Simulator: Varying P at $\phi(P) = P/32$. Method columns show normalized runtime. Speedup columns show the multiplicative gain of Clv and NCV over the best baseline.

	Baselines				Ours		Speedup	
P	AllPush	AllPull	MTL-Obj	MTL-Tsk	Clv	NCV	Clv	NCV
32	4.69	27.63	27.63	4.69	**2.25**	**2.36**	2.08×	1.99×
64	8.91	27.61	27.61	18.90	**2.37**	**2.51**	3.76×	3.55×
256	33.57	27.41	27.41	30.99	**2.81**	**2.82**	9.75×	9.72×
1024	127.22	27.03	27.03	57.96	**3.27**	**3.65**	8.26×	7.41×
2048	249.13	26.54	26.54	92.09	**3.33**	**4.05**	7.97×	6.55×
4096	486.09	25.91	25.91	158.44	**3.36**	**6.10**	7.72×	4.25×
8192	905.84	24.21	24.21	275.53	**4.31**	**9.63**	5.62×	2.51×
16384	1732.85	23.20	23.20	507.79	**6.28**	**14.26**	3.70×	1.63×

Table 3: Simulator: Varying CPU power ϕ at $P = 2048$.

	Baselines				Ours		Speedup	
ϕ	AllPush	AllPull	MTL-Obj	MTL-Tsk	Clv	NCV	Clv	NCV
64	249.13	26.54	26.54	92.09	**3.33**	**4.05**	7.97×	6.55×
128	249.13	13.29	13.29	81.20	**3.16**	**4.06**	4.21×	3.27×
192	249.13	8.88	8.88	77.56	**3.00**	**4.06**	2.96×	2.19×
256	249.13	6.67	6.67	75.78	**2.83**	**4.07**	2.36×	1.64×

Table 4: Simulator: Varying target object distribution skew θ at $P = 2048$, $\phi = 256$.

	Baselines				Ours		Speedup	
θ	AllPush	AllPull	MTL-Obj	MTL-Tsk	Clv	NCV	Clv	NCV
0	7.75	6.67	6.67	7.32	**2.85**	**4.73**	2.34×	1.41×
0.1	8.39	6.67	6.67	7.07	**2.88**	**4.67**	2.32×	1.43×
0.3	10.56	6.67	6.67	8.44	**2.82**	**4.84**	2.36×	1.38×
0.5	32.84	6.67	6.67	14.40	**2.80**	**4.56**	2.38×	1.46×
0.8	128.89	6.67	6.67	41.03	**2.87**	**4.27**	2.32×	1.56×
0.99	249.13	6.67	6.67	75.78	**2.83**	**4.07**	2.36×	1.64×
1.2	441.18	6.67	6.68	127.73	**2.85**	**4.01**	2.34×	1.66×
1.5	763.94	6.74	6.75	214.64	**2.88**	**4.40**	2.34×	1.53×

target object. *AllPull* pulls every object to the host CPU and performs the computation there. *MTL-Obj* and *MTL-Tsk* apply the minimize-total-load heuristic described in §3.4 at per-object and per-task granularity, respectively.

7.1 Simulation Results

Each table reports the *normalized runtime* of different methods, defined as execution runtime over estimated lower bound (lower is better). For the estimated lower bound, we use $T_{PerPIM} + T_{Load} + T_{Single}$ from Theorem 3.1, ignoring the constant factors. Each table also reports the speedup of Clv/NCV over the *best* baseline (higher is better). By default, we use $P = 2048$, $m = 100$k, $n = 1000$, $\theta = 0.99$, and $w_i = 5$ arithmetic operations. Tables 2–5 sweep different P, ϕ, θ, and work distribution. **Both Clv and NCV outperform every baseline in all configurations**: 1.28×–9.75× for Clv, and 1.01×–9.72× for NCV.

7.2 Evaluation on UPMEM

We implement the same algorithms on real UPMEM hardware [29], which is in a weak-CPU regime with $P = 2048$ and $\phi \approx 900$. By default, we use $n = 512$, $m = 131{,}072$, $w_i = 2$M arithmetic operations, and $\theta = 0.99$. T/O entries indicate a per-invocation timeout, with a normalized runtime of at least 40. Tables 6–8 sweep different n, w_i, and θ. **Both Clv and NCV outperform every baseline in all configurations**: 1.72×–1.81× for Clv, and 1.23×–1.42× for NCV.

Table 5: Simulator: Varying per-task work distribution at $P = 2048$, $\phi = 384$. "exp" is Exponential with mean 5. "pareto-α" is bounded Pareto with tail index α.

	Baselines				Ours		Speedup	
Dist.	AllPush	AllPull	MTL-Obj	MTL-Tsk	Clv	NCV	Clv	NCV
Constant	238.97	3.52	3.52	238.97	**2.57**	**2.80**	1.37×	1.26×
Exp	244.26	3.56	3.56	141.46	**2.78**	**3.53**	1.28×	1.01×
Pareto-1.2	250.77	4.66	4.66	51.08	**2.34**	**4.24**	1.99×	1.10×
Pareto-1.5	249.13	4.46	4.46	73.92	**2.61**	**4.08**	1.71×	1.09×
Pareto-2.0	246.32	4.22	4.22	109.58	**2.62**	**3.96**	1.61×	1.07×

Table 6: UPMEM: Varying object count n. T/O denotes a per-invocation timeout.

	Baselines				Ours		Speedup	
n	AllPush	AllPull	MTL-Obj	MTL-Tsk	Clv	NCV	Clv	NCV
128	T/O	2.19	2.20	2.20	**1.23**	**1.59**	1.79×	1.38×
256	T/O	2.20	2.20	2.20	**1.22**	**1.59**	1.80×	1.38×
512	T/O	2.19	2.20	2.20	**1.23**	**1.59**	1.79×	1.38×
1024	T/O	2.20	2.21	2.20	**1.23**	**1.60**	1.79×	1.38×
2048	T/O	2.20	2.20	2.20	**1.23**	**1.79**	1.79×	1.23×

Table 7: UPMEM: Varying per-task work w_i.

	Baselines				Ours		Speedup	
w_i	AllPush	AllPull	MTL-Obj	MTL-Tsk	Clv	NCV	Clv	NCV
500K	T/O	2.21	2.21	2.22	**1.22**	**1.56**	1.81×	1.42×
1M	T/O	2.21	2.20	2.21	**1.22**	**1.56**	1.81×	1.41×
2M	T/O	2.20	2.20	2.21	**1.23**	**1.59**	1.79×	1.38×
4M	T/O	2.19	2.18	2.13	**1.24**	**1.62**	1.72×	1.31×

Table 8: UPMEM: Varying target object distribution skew θ.

	Baselines				Ours		Speedup	
θ	AllPush	AllPull	MTL-Obj	MTL-Tsk	Clv	NCV	Clv	NCV
0	13.71	2.23	2.23	2.25	**1.23**	**1.59**	1.81×	1.40×
0.5	T/O	2.21	2.18	2.15	**1.23**	**1.59**	1.75×	1.35×
0.99	T/O	2.22	2.23	2.23	**1.23**	**1.59**	1.81×	1.40×
1.2	T/O	2.20	2.19	2.22	**1.23**	**1.59**	1.78×	1.38×
1.5	T/O	2.20	2.20	2.20	**1.22**	**1.59**	1.80×	1.38×

8 Conclusion

Processing-in-memory (PIM) architectures offer a compelling architectural solution to the memory latency/bandwidth wall by bringing computation closer to data. To fully exploit the capability of PIM systems, we introduce the *PIM Scheduling* problem, capturing the asymmetry between the host CPU and the PIM side.

We establish fundamental lower bounds on the makespan for a batch of tasks and design a non-preemptive clairvoyant algorithm achieving a constant-factor competitive ratio. More importantly, we develop a non-clairvoyant scheduling algorithm whose asymptotic makespan differs from the clairvoyant lower bound by only a small

additive term corresponding to the heaviest PIM-side load (i.e., Push cost) of a single task in the batch. We further analyze the random object placement strategy and obtain performance guarantees. Our implementation and evaluation show that our algorithms achieve up to a 9.8× speedup in simulation and up to a 1.8× speedup on real PIM hardware. In summary, this work sets up a formal theoretical foundation for scheduling in PIM systems, and presents (nearly) constant competitive scheduling algorithms that are effective in practice.

Acknowledgments

This research was supported by National Key Research & Development Program of China (2024YFB4505600), Department of Energy award DE-ACO5-000R22725 (subcontract #CW62109), National Science Foundation Grants (PPoSS-2119027 and CCF-2106699), and the Parallel Data Lab Consortium (Bloomberg LP, Everpure, Google, Jane Street, LayerZero Labs, Meta, Microsoft Research, Oracle Corporation, Salesforce, Uber, and Western Digital). This work was done in part while the first, second, third, and fifth authors were visiting the Simons Institute for the Theory of Computing. We thank the anonymous reviewers for their helpful feedback.

References

[1] Umut A. Acar, Guy E. Blelloch, and Robert D. Blumofe. 2000. The data locality of work stealing. In *Proceedings of the Twelfth Annual ACM Symposium on Parallel Algorithms and Architectures* (Bar Harbor, Maine, USA) *(SPAA '00)*. Association for Computing Machinery, New York, NY, USA, 1–12. doi:10.1145/341800.341801

[2] Atul Adya, William J. Bolosky, Miguel Castro, Gerald Cermak, Ronnie Chaiken, John R. Douceur, Jon Howell, Jacob R. Lorch, Marvin Theimer, and Roger P. Wattenhofer. 2002. FARSITE: Federated, Available, and Reliable Storage for an Incompletely Trusted Environment. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI)*. USENIX Association, Boston, MA, 1–14. https://www.usenix.org/conference/osdi-02/farsite-federated-available-and-reliable-storage-incompletely-trusted-environment

[3] Kunal Agrawal, William Kuszmaul, Zhe Wang, and Jinhao Zhao. 2024. Distributed Load Balancing in the Face of Reappearance Dependencies. In *Proceedings of the 36th ACM Symposium on Parallelism in Algorithms and Architectures* (Nantes, France) *(SPAA '24)*. Association for Computing Machinery, New York, NY, USA, 321–330. doi:10.1145/3626183.3659968

[4] Junwhan Ahn, Sungpack Hong, Sungjoo Yoo, Onur Mutlu, and Kiyoung Choi. 2015. A scalable processing-in-memory accelerator for parallel graph processing. In *2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)*. 105–117. doi:10.1145/2749469.2750386

[5] David Applegate, Aaron Archer, Vijay Gopalakrishnan, Seungjoon Lee, and K. K. Ramakrishnan. 2010. Optimal content placement for a large-scale VoD system. In *Proceedings of the 6th International COnference* (Philadelphia, Pennsylvania) *(CoNEXT '10)*. Association for Computing Machinery, New York, NY, USA, Article 4, 12 pages. doi:10.1145/1921168.1921174

[6] Aaron Archer, Kevin Aydin, Mohammad Hossein Bateni, Vahab Mirrokni, Aaron Schild, Ray Yang, and Richard Zhuang. 2019. Cache-aware load balancing of data center applications. *Proc. VLDB Endow.* 12, 6 (Feb. 2019), 709–723. doi:10.14778/3311880.3311887

[7] Yossi Azar, Andrei Z. Broder, Anna R. Karlin, and Eli Upfal. 1999. Balanced Allocations. *SIAM J. Comput.* 29, 1 (1999), 180–200. doi:10.1137/S0097539795288490

[8] Petra Berenbrink, Tom Friedetzky, Zengjian Hu, and Russell Martin. 2008. On Weighted Balls-into-bins Games. *Theoretical Computer Science* 409, 3, 511–520. doi:10.1016/j.tcs.2008.09.023

[9] Petra Berenbrink, Friedhelm Meyer auf der Heide, and Klaus Schröder. 1997. Allocating weighted jobs in parallel. In *Proceedings of the Ninth Annual ACM Symposium on Parallel Algorithms and Architectures* (Newport, Rhode Island, USA) *(SPAA '97)*. Association for Computing Machinery, New York, NY, USA, 302–310. doi:10.1145/258492.258522

[10] Robert D. Blumofe and Charles E. Leiserson. 1993. Space-efficient scheduling of multithreaded computations. In *Proceedings of the Twenty-Fifth Annual ACM Symposium on Theory of Computing* (San Diego, California, USA) *(STOC '93)*. Association for Computing Machinery, New York, NY, USA, 362–371. doi:10.1145/167088.167196

[11] Robert D. Blumofe and Charles E. Leiserson. 1999. Scheduling multithreaded computations by work stealing. *J. ACM* 46, 5 (Sept. 1999), 720–748. doi:10.1145/324133.324234

[12] Shuang Chen, Yi Jiang, Christina Delimitrou, and José F. Martínez. 2022. PIM-Cloud: QoS-Aware Resource Management of Latency-Critical Applications in Clouds with Processing-in-Memory. In *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. 1086–1099. doi:10.1109/HPCA53966.2022.00083

[13] Sitian Chen, Amelie Chi Zhou, Yucheng Shi, Yusen Li, and Xin Yao. 2025. UpANNS: Enhancing Billion-Scale ANNS Efficiency with Real-World PIM Architecture. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '25)*. Association for Computing Machinery, New York, NY, USA, 789–804. doi:10.1145/3712285.3759777

[14] Xinyu Chen, Zhongle Xie, Huan Li, Ke Chen, Lidan Shou, Dawei Jiang, and Gang Chen. 2025. PimShare: Scheduling for Multi-DNN Inference on Processing-in-memory Accelerated Edge Server. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* (2025), 1–1. doi:10.1109/TCAD.2025.3641876

[15] Guohao Dai, Tianhao Huang, Yuze Chi, Jishen Zhao, Guangyu Sun, Yongpan Liu, Yu Wang, Yuan Xie, and Huazhong Yang. 2019. GraphH: A Processing-in-Memory Architecture for Large-Scale Graph Processing. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 38, 4 (2019), 640–653. doi:10.1109/TCAD.2018.2821565

[16] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. 2007. Dynamo: Amazon's Highly Available Key-Value Store. In *Proceedings of Twenty-First ACM SIGOPS Symposium on Operating Systems Principles* (Stevenson, Washington, USA) *(SOSP '07)*. Association for Computing Machinery, New York, NY, USA, 205–220. doi:10.1145/1294261.1294281

[17] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. 2003. The Google file system. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles* (Bolton Landing, NY, USA) *(SOSP '03)*. Association for Computing Machinery, New York, NY, USA, 29–43. doi:10.1145/945445.945450

[18] Inyong Hwang, Donghyeon Kim, Seokwon Kang, Taehyeong Park, Taehoon Kim, Jiwon Seo, Hanjun Kim, Youngsok Kim, and Yongjun Park. 2025. PIM-CARE: A Compiler-Assisted Dynamic Resource Allocation Framework for Real-world DRAM PIM. In *Proceedings of the 39th ACM International Conference on Supercomputing (ICS '25)*. Association for Computing Machinery, New York, NY, USA, 458–472. doi:10.1145/3721145.3725777

[19] 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* (Virtual Event, USA) *(SPAA '21)*. Association for Computing Machinery, New York, NY, USA, 295–306. doi:10.1145/3409964.3461816

[20] Hongbo Kang, Yiwei Zhao, Guy E. Blelloch, Laxman Dhulipala, Yan Gu, Charles McGuffey, and Phillip B. Gibbons. 2022. PIM-Tree: A Skew-Resistant Index for Processing-in-Memory. *Proc. VLDB Endow.* 16, 4 (Dec. 2022), 946–958. doi:10.14778/3574245.3574275

[21] Hongbo Kang, Yiwei Zhao, Guy E. Blelloch, Laxman Dhulipala, Yan Gu, Charles McGuffey, and Phillip B. Gibbons. 2023. PIM-Trie: A Skew-Resistant Trie for Processing-in-Memory. In *Proceedings of the 35th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA '23)*. 1–14. doi:10.1145/3558481.3591070

[22] Hongbo Kang, Yiwei Zhao, Guy E Blelloch, Laxman Dhulipala, Yan Gu, Charles McGuffey, and Phillip B Gibbons. 2025. PIM-tree: A Skew-resistant Index for Processing-in-Memory. *The VLDB Journal* 34, 6 (Dec. 2025), 66. doi:10.1007/s00778-025-00937-5

[23] Donghyeon Kim, Taehoon Kim, Inyong Hwang, Taehyeong Park, Hanjun Kim, Youngsok Kim, and Yongjun Park. 2023. Virtual PIM: Resource-Aware Dynamic DPU Allocation and Workload Scheduling Framework for Multi-DPU PIM Architecture. In *2023 32nd International Conference on Parallel Architectures and Compilation Techniques (PACT)*. 112–123. doi:10.1109/PACT58117.2023.00018

[24] Hyoungjoo Kim, Yiwei Zhao, Andrew Pavlo, and Phillip B. Gibbons. 2025. No Cap, This Memory Slaps: Breaking Through the Memory Wall of Transactional Database Systems with Processing-in-Memory. *Proc. VLDB Endow.* (2025), 4241–4254. doi:10.14778/3749646.3749690

[25] Dongjae Lee, Bongjoon Hyun, Taehun Kim, and Minsoo Rhu. 2024. PIM-MMU: A Memory Management Unit for Accelerating Data Transfers in Commercial PIM Systems. In *2024 57th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 627–642. doi:10.1109/MICRO61859.2024.00053

[26] Young Sik Lee and Tae Hee Han. 2021. Task Parallelism-Aware Deep Neural Network Scheduling on Multiple Hybrid Memory Cube-Based Processing-in-Memory. *IEEE Access* 9 (2021), 68561–68572. doi:10.1109/ACCESS.2021.3077294

[27] Michael Mitzenmacher. 2001. The power of two choices in randomized load balancing. *IEEE Transactions on Parallel and Distributed Systems* 12, 10 (2001), 1094–1104. doi:10.1109/71.963420

[28] David Pisinger and Paolo Toth. 1998. Knapsack Problems. *Handbook of Combinatorial Optimization: Volume1–3* (1998), 299–428. doi:10.1007/978-1-4613-0303-9_5

[29] Qualcomm. 2026. UPMEM Technology. https://github.com/upmem. Accessed May 2026.

[30] Shunchen Shi, Xueqi Li, Zhaowu Pan, Peiheng Zhang, and Ninghui Sun. 2024. CoPIM: A Collaborative Scheduling Framework for Commodity Processing-in-memory Systems. In *2024 IEEE 42nd International Conference on Computer Design (ICCD)*. 44–51. doi:10.1109/ICCD63220.2024.00018

[31] Kunal Talwar and Udi Wieder. 2007. Balanced allocations: the weighted case. In *Proceedings of the Thirty-Ninth Annual ACM Symposium on Theory of Computing* (San Diego, California, USA) *(STOC '07)*. Association for Computing Machinery, New York, NY, USA, 256–265. doi:10.1145/1250790.1250829

[32] Dufy Teguia, Jiaxuan Chen, Stella Bitchebe, Oana Balmau, and Alain Tchana. 2024. vPIM: Processing-in-Memory Virtualization. In *Proceedings of the 25th International Middleware Conference (Middleware '24)*. 417–430. doi:10.1145/3652892.3700782

[33] Boyu Tian, Qihang Chen, and Mingyu Gao. 2023. ABNDP: Co-optimizing Data Access and Load Balance in Near-Data Processing. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3 (ASPLOS 2023)*. 3–17. doi:10.1145/3582016.3582026

[34] Leslie G. Valiant. 1990. A bridging model for parallel computation. *Commun. ACM* 33, 8 (Aug. 1990), 103–111. doi:10.1145/79173.79181

[35] Zhe Wang, Jinhao Zhao, Kunal Agrawal, He Liu, Meng Xu, and Jing Li. 2023. Provably Good Randomized Strategies for Data Placement in Distributed Key-Value Stores. In *Proceedings of the 28th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming* (Montreal, QC, Canada) *(PPoPP '23)*. Association for Computing Machinery, New York, NY, USA, 27–38. doi:10.1145/3572848.3577501

[36] Sage A. Weil, Scott A. Brandt, Ethan L. Miller, Darrell D. E. Long, and Carlos Maltzahn. 2006. Ceph: A Scalable, High-Performance Distributed File System. In *7th USENIX Symposium on Operating Systems Design and Implementation (OSDI 06)*. USENIX Association, Seattle, WA. https://www.usenix.org/conference/osdi-06/ceph-scalable-high-performance-distributed-file-system

[37] Puqing Wu, Minhui Xie, Enrui Zhao, Dafang Zhang, Jing Wang, Xiao Liang, Kai Ren, and Yunpeng Chai. 2025. Turbocharge ANNS on real processing-in-memory by enabling fine-grained per-PIM-core scheduling. In *Proceedings of the 2025 USENIX Conference on Usenix Annual Technical Conference* (Boston, MA, USA) *(USENIX ATC '25)*. USENIX Association, USA, Article 72, 19 pages. https://www.usenix.org/conference/atc25/presentation/wu-puqing

[38] Yiwei Zhao, Hongbo Kang, Yan Gu, Guy E. Blelloch, Laxman Dhulipala, Charles McGuffey, and Phillip B. Gibbons. 2025. Optimal Batch-Dynamic kd-trees for Processing-in-Memory with Applications. In *Proceedings of the 37th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA '25)*. 350–366. doi:10.1145/3694906.3743318

[39] Yiwei Zhao, Hongbo Kang, Ziyang Men, Yan Gu, Guy E. Blelloch, Laxman Dhulipala, Charles McGuffey, and Phillip B. Gibbons. 2026. PIM-zd-tree: A Fast Space-Partitioning Index Leveraging Processing-in-Memory. In *Proceedings of the 31st ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming* (Sydney, NSW, Australia) *(PPoPP '26)*. Association for Computing Machinery, New York, NY, USA, 480–495. doi:10.1145/3774934.3786411

[40] Yiwei Zhao, Qiushi Lin, Hongbo Kang, Guy E. Blelloch, Laxman Dhulipala, Yan Gu, Charles McGuffey, and Phillip B. Gibbons. 2025. TD-Orch: Scalable Load-Balancing for Distributed Systems with Applications to Graph Processing. arXiv:2511.11843 [cs.DC]

[41] George Kingsley Zipf. 1949. *Human Behavior and the Principle of Least Effort: An Introduction to Human Ecology*. Addison-Wesley Press, Cambridge, MA. https://www.mpi.nl/publications/item2407822/human-behavior-and-principle-least-effort-introduction-human-eoclogy