Brief Announcement: Optimal Dispersion Under Asynchrony

Debasish Pattanayak
Department of Computer Science and Engineering, Indian Institute of Technology Indore, India

Ajay D. Kshemkalyani
Department of Computer Science, University of Illinois Chicago, IL, USA

Manish Kumar
Department of Computer Science and Engineering, Indian Institute of Technology Madras, Chennai, India

Anisur Rahaman Molla
Indian Statistical Institute, Kolkata, India

Gokarna Sharma
Department of Computer Science, Kent State University, OH, USA

Abstract

We study the dispersion problem in anonymous port-labeled graphs: $k \leq n$ mobile agents, each with a unique ID and initially located arbitrarily on the nodes of an n-node graph with maximum degree Δ, must autonomously relocate so that no node hosts more than one agent. Dispersion serves as a fundamental task in the distributed computing of mobile agents, and its complexity stems from key challenges in local coordination under anonymity and limited memory.

The goal is to minimize both the time to achieve dispersion and the memory required per agent. It is known that any algorithm requires $\Omega(k)$ time in the worst case, and $\Omega(\log k)$ bits of memory per agent. A recent result [9] gives an optimal $O(k)$-time algorithm in the synchronous setting and an $O(k \log k)$-time algorithm in the asynchronous setting, both using $O(\log(k + \Delta))$ bits. We close the complexity gap in the asynchronous setting by presenting the first dispersion algorithm that runs in optimal $O(k)$ time using $O(\log(k + \Delta))$ bits of memory per agent. Our solution relies on a novel technique for constructing a port-one tree in anonymous graphs, which may be of independent interest.

2012 ACM Subject Classification Theory of computation → Distributed algorithms

Keywords and phrases Distributed algorithms, mobile agents, local communication, dispersion, asynchrony, port-one tree, time and memory complexity

Digital Object Identifier 10.4230/LIPIcs.DISC.2025.63

Related Version *Full Version*: `https://arxiv.org/abs/2507.01298` [16]

Funding *Manish Kumar*: Supported by the Centre for Cybersecurity, Trust and Reliability (CyStar), IIT Madras.

1 Introduction

The *dispersion* problem, denoted as DISPERSION, involves $k \leq n$ mobile agents initially placed arbitrarily on the nodes of an n-node anonymous graph with maximum degree Δ. The agents must autonomously relocate so that each occupies a distinct node. The objective is to design algorithms that optimize both time and memory complexities. Time complexity is the total time to achieve dispersion, while memory complexity is the maximum bits stored per agent. Fundamental limits exist: certain topologies require $\Omega(k)$ time, and $\Omega(\log k)$ memory bits per agent are necessary for unique identifiers [2].

The dispersion problem has been extensively studied, both in arbitrary undirected graphs [2, 7, 8, 9, 10, 12, 13, 18], directed graphs [6] and in graphs with restricted topologies such as trees [2], grids [3, 4, 11], and dynamic rings [1, 15]. Several works have also examined fault-tolerant variants of the problem [5, 14, 15, 17]. Focusing on the arbitrary undirected graph, the state-of-the-art results of Kshemkalyani et al. [9], presented an optimal $O(k)$-round synchronous ($\mathcal{SYNC}$) algorithm and an $O(k \log k)$-epoch asynchronous ($\mathcal{ASYNC}$) algorithm, both using only $O(\log(k + \Delta))$ bits per agent. Here, an epoch denotes the minimal time interval in which each agent completes at least one computation cycle, and is equivalent to a round in $\mathcal{SYNC}$. Motivated by this line of work, we address the central open question: *Can an optimal $O(k)$-epoch solution be designed for* DISPERSION *in the $\mathcal{ASYNC}$ setting?*

Contributions. We answer this question affirmatively by providing an optimal $O(k)$-epoch solution in $\mathcal{ASYNC}$ with $O(\log(k+\Delta))$ bits per agent, showing that synchrony is not essential for time-optimal dispersion. This is achieved through a novel construction of a *Port-One tree* (P1TREE), which prioritizes edges connected to a port labeled '1' at either endpoint. This prioritization allows agents to find empty neighbor nodes in $O(1)$ epochs, even in $\mathcal{ASYNC}$, a task that previously required $O(\log k)$ epochs. Since settling k agents requires visiting k empty nodes, and each visit takes $O(1)$ epochs, the entire algorithm achieves an optimal $O(k)$ time complexity.

Challenges. The primary obstacle blocking the path to optimal dispersion in $\mathcal{ASYNC}$ is finding empty neighboring nodes in $O(1)$-epochs. The state-of-the-art technique relied on an "oscillation" mechanism where agents made periodic trips to distinguish empty nodes from temporarily unoccupied ("vacated") nodes. This timing-based coordination fundamentally fails in $\mathcal{ASYNC}$ because agents lack a shared sense of time. This creates a barrier to achieving $O(k)$ epoch solutions analogous to $\mathcal{SYNC}$ using a DFS-based traversal strategy.

Our key contribution is to bypass this time-based coordination strategy with a structural approach. We introduce the Port-One Tree (P1TREE), a structure that allows for agents to distinguish between "vacated" and "empty" nodes by visiting port-one neighbors. Further, we overcome several non-trivial challenges to make this technique viable:

1. **Building a** P1TREE**:** We design a modified DFS that prioritizes "port-1" edges while avoiding cycles.
2. **Selection of "vacated" nodes:** We devise a selection rule that guarantees at least $\lfloor l/3 \rfloor$ vacant nodes in a tree of size l, providing enough scouts for probing.
3. **Keeping track of vacated nodes:** Information about a vacated node and its port 1 neighbor is stored on an agent (that had originally settled there), which travels with the DFS head, avoiding the high memory cost of storing it at neighbors.
4. **Parallel probe with vacated nodes:** Our probing protocol involves a multi-hop check (up to 3 hops). The properties of our vacant node selection guarantee that a scout can distinguish an empty node from a vacated one by checking for the presence of settled agents at its port 1 neighbor or in the scout pool. This check completes in $O(1)$ epochs.
5. **Returning scouts home:** After dispersion is achieved, scouts re-traverse the constructed P1TREE in a post-order fashion to return to their originally assigned nodes, a process we show takes $O(k)$ epochs.

For general initial configurations with multiple starting locations, we extend the size-based merging technique from prior work [13]. When two DFS explorations meet, the one with fewer settled agents is "absorbed" by the larger one, ensuring that a single, monotonically growing tree eventually covers all k agents within the $O(k)$ time bound.

2 Model and Preliminaries

Graph. We consider a simple, undirected, connected graph $G = (V, E)$, where $n = |V|$ and $m = |E|$. The graph nodes are *anonymous* (lack unique identifiers) but are *port-labeled*: at each node v, incident edges have distinct local labels from 1 to δ_v. An edge $\{u, v\}$ is associated with two port numbers, p_{uv} at u and p_{vu} at v, which are assigned independently. Nodes are memoryless.

Edge and Node Terminology. We encode an edge $\{u, v\}$ as a 4-tuple $e = [u, p_{uv}, p_{vu}, v]$. We define the type of an edge $\{u, v\}$ by $\texttt{type}(\{u, v\})$ based on its port numbers:

- `t11`, if $p_{uv} = 1$ and $p_{vu} = 1$.
- `tp1`, if $p_{uv} \neq 1$ and $p_{vu} = 1$.
- `t1q`, if $p_{uv} = 1$ and $p_{vu} \neq 1$.
- `tpq`, if $p_{uv} \neq 1$ and $p_{vu} \neq 1$.

Agents. The system has $k \leq n$ mobile agents, $A = \{a_1, \ldots, a_k\}$, each with a unique ID, a_i.ID. Agents are initially located arbitrarily. Communication is local: an agent at a node can only interact with other co-located agents.

Time and Complexity. Agents operate in asynchronous "Communicate-Compute-Move" (CCM) cycles. An *epoch* is a minimal time interval where each agent completes at least one CCM cycle. A subsequent epoch begins after the end of the current one. Two epochs may have different lengths of time. Time complexity is measured in epochs, and memory complexity is the maximum bits stored in an agent's persistent memory.

3 Port-One Tree and its Construction

In this section, we define the Port-One Tree (P1Tree), prove its existence, and describe a distributed construction method that forms the basis of our dispersion algorithm.

Port-One Tree (P1Tree). Intuitively, every vertex in a P1Tree is incident to at least one tree edge connected to a port 1 at one of its endpoints.

▶ **Definition 1** (Port-One Tree (P1Tree)). *Let $G = (V, E)$ be an anonymous port-labeled graph. A tree $\mathcal{T} \subseteq E$ is a* P1Tree *if each vertex $v \in \mathcal{T}$ has at least one incident edge $\{v, w\} \in \mathcal{T}$ such that $\texttt{type}(\{v, w\}) \in \{$`tp1`, `t11`, `t1q`$\}$.*

A P1Tree may not contain all edges of the eligible types, as including them might create cycles. Figure 1 shows an example. Since every node has a port labeled 1, it has an edge of type `t11` *or* `t1q`.

▶ **Lemma 2.** *For any port-labeled graph G, there exists at least one* P1Tree *$\mathcal{T}$.*

3.1 DFS-based Construction

We now describe a modified DFS traversal, `DFS_P1Tree()`, that constructs a P1Tree. This algorithm forms the backbone of our agent-based dispersion strategy. Each node has two states: EMPTY and OCCUPIED. We further categorize each node $v \in G$ into one of four types:

- **unvisited**: v has not yet been visited by the DFS.
- **fullyVisited**: v has been visited, and all its neighbors have also been visited.

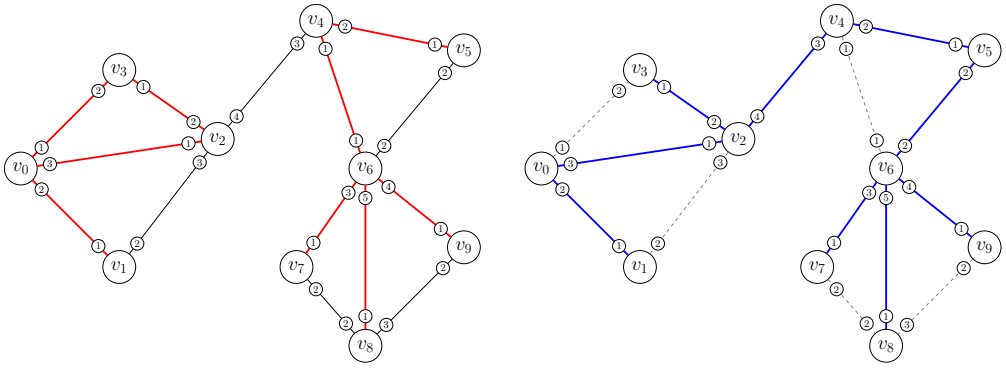


Figure 1 An example of a Port-One Tree. Left: Edges incident to a port 1 are highlighted in red. Right: A valid P1Tree is shown with tree edges in blue (solid) and non-tree edges in gray (dashed).

- **partiallyVisited**: v was reached via a `tpq` edge, and all of its unvisited neighbors are also reachable only via `tpq` edges.
- **visited**: v has been visited but is not **partiallyVisited** or **fullyVisited**.

`DFS_P1Tree()` prioritizes edges at any node in the order: `tp1` $\succ$ `t11` $\sim$ `t1q` $\succ$ `tpq`. The traversal proceeds like a standard DFS, moving forward to **unvisited** neighbors of **visited** nodes and backtracking from **fullyVisited** nodes. The key modification handles **partiallyVisited** nodes to ensure the P1Tree property. Suppose the DFS head reaches node u. The type of u is determined by inspecting its neighbors $N(u)$. Based on the type of u, the DFS head proceeds as follows:

(D0) All nodes are initially **unvisited**.

(D1) If u is **fullyVisited**, the DFS head backtracks to the parent of u.

(D2) If u is **visited**, the DFS head visits an **unvisited** neighbor of u along the highest priority edge.

(D3) If u is **partiallyVisited**, the DFS head backtracks to the parent of u.

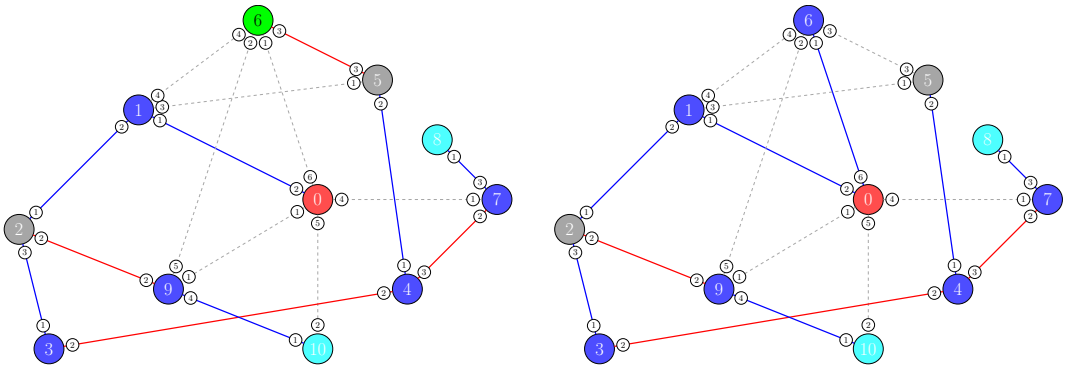
(D4) A **partiallyVisited** node u is treated as "re-visitable". When the DFS traversal later reaches u from a neighbor w via an edge of type `tp1` or `t11`, a *reconfiguration* occurs: the original parent edge of u (which was of type `tpq`) is removed from the tree, and the new edge $\{w, u\}$ is added. The parent of u is now swapped to w, and u's type is changed to **visited**.

This parent-swap operation is crucial, and swapping the parent of a leaf does not create a cycle. This mechanism guarantees that every node eventually acquires a port-1 incident edge in the final tree. Figure 2 illustrates this process.

▶ **Theorem 3.** `DFS_P1Tree()` *produces a* P1Tree $\mathcal{T}$ *of a port-labeled graph* G.

4 Agent-based P1Tree Construction

We now describe how mobile agents execute the `DFS_P1Tree()` algorithm to construct a P1Tree and disperse. The key challenge is performing neighborhood searches efficiently in an asynchronous environment to find the next edge to traverse. Our solution relies on selecting some nodes to be VACATED and using their settled agents as scouts for a fast parallel probing mechanism.



(a) DFS backtracks at ⑥ marking it **partiallyVisited**. Its parent edge $\{⑥, ⑤\}$ is `tpq` and its only **unvisited** neighbor ⑨ is connected via a `tpq` edge.

(b) Reconfiguration: the `tpq` edge $\{⑥, ⑤\}$ is removed and the edge $\{⓪, ⑥\}$ of type `tp1` is added, making ⓪ the new parent of ⑥.

Figure 2 Illustration of reconfiguration on a P1Tree $\mathcal{T}$. A `tpq` edge is swapped for an edge of type `tp1` or `t11`.

4.1 Executing the DFS with Agents

Agents collectively act as the "DFS head". Initially, all agents are at a root node v_0 and are *unsettled*. The agent with the highest ID settles at v_0. The remaining unsettled agents form a scout pool. To decide the next move, these scouts perform a neighborhood search. When the DFS head moves to an unvisited node, the highest-ID unsettled agent settles there. This process continues, with agents settling at new nodes, until all unsettled agents have settled. The reconfiguration of **partiallyVisited** nodes is handled by the settled agent at that node, which updates its parent information when visited by the DFS head from its port-1 neighbor.

4.2 Selecting Vacant Nodes for Scouting

To perform neighborhood searches in $O(1)$ epochs, we need multiple agents to probe neighbor ports in parallel. We achieve this by designating certain nodes as VACATED. The agent that would have stayed at a VACATED node instead joins the scout pool and travels with the DFS head. A node's state is either EMPTY (not yet visited), OCCUPIED (an agent is settled and remains there), or VACATED (settled agent is travelling with the DFS head). The decision to vacate is made locally based on the state of itself and its port-1 neighbor. Starting with an OCCUPIED root node, a node can become VACATED if its port-1 neighbor is occupied and it is not acting as the port-1 neighbor for a previously vacated node. These rules are designed to ensure that information about a vacated node can be retrieved from a nearby occupied node, which is essential for the probing mechanism described next.

▶ **Lemma 4.** *Let $\mathcal{T}$ be the partial tree of size k constructed by the agents. At least $\lfloor k/3 \rfloor$ of these k vertices are in state* VACATED.

4.3 Parallel Probing for $O(1)$-epoch Neighborhood Search

As established, the central challenge under asynchrony is distinguishing an empty node from a vacated one in constant time. Our `Parallel_Probe()` mechanism solves this directly. With a pool of scouts available (from vacated nodes and remaining unsettled agents), the DFS head can perform a neighborhood search in $O(1)$ epochs. `Parallel_Probe()` assigns available scouts to probe unexplored ports of the current node x. A scout agent a traversing

port p_{xy} to a neighbor y must determine if y is OCCUPIED, VACATED, or genuinely EMPTY (i.e., unvisited by any DFS). The scout agent determines a node x is empty by visiting the port-1 neighbor y of x, and the port-1 neighbor z of y if y is also empty. Each scout travels at most 3 edges away from x and returns, taking a constant number of moves. Since there are at least $\lceil (k-2)/3 \rceil$ scouts, all neighbors of a node can be probed in a constant number of parallel waves, thus taking $O(1)$ epochs.

▶ **Lemma 5.** `Parallel_Probe()` *at a node $x \in V$ correctly determines the state of all its neighbors in $O(1)$ epochs.*

5 The Optimal Asynchronous Dispersion Algorithm

The complete asynchronous dispersion algorithm, `RootedAsync()`, integrates the concepts of P1TREE construction, node vacating, and parallel probing. We first describe the rooted case and then extend it to general initial configurations.

5.1 Rooted Dispersion

The algorithm unfolds in two phases: exploration and retrace. In the exploration phase, agents build a P1TREE rooted at v_0 by traversing the graph and settling at nodes. At each node, the exploring DFS head checks if that node can be vacated or not. It settles an unsettled agent from the scout pool if the parallel neighborhood search finds an empty node. In the retrace phase, scouts return to their original nodes. The retrace works by having the agents move in a post-order traversal of the constructed P1TREE, detaching from the group when they reach their home nodes. The post-order traversal is possible due to the presence of parent information, the most recently visited child, and the immediate sibling of each settled agent. Given a group of settled agents, we prioritize the immediate sibling, then the most recently visited child. Afterwards, the child is detached. When no child remains, the agent group takes the port to reach the parent. Both phases take $O(k)$ epochs.

Summarizing the above discussion, we obtain the following results.

▶ **Theorem 6.** *The* `RootedAsync()` *algorithm achieves dispersion in $O(k)$ epochs with $O(\log k + \Delta)$ bits of memory per agent.*

5.2 General Dispersion

When agents start at multiple nodes, each group of co-located agents (a "multiplicity") initiates its own independent instance of the rooted dispersion algorithm. Each exploration is identified by a unique "treelabel" based on its root agent's ID. When two explorations meet (i.e., a scout from one tree probes a node occupied by an agent from another tree), a merger protocol is invoked. We call this `GeneralAsync()`. The merger strategy is based on a size-based subsumption rule. When exploration T_1 with k_1 agents meets T_2 with k_2 agents, the smaller exploration is absorbed by the larger one. For instance, if $k_1 < k_2$, all agents from T_1 (both settled and scouts) will abandon their exploration, gather at their root if possible, and then join the DFS head of T_2 as new unsettled agents. The larger exploration T_2 pauses briefly to absorb the new agents and then continues its DFS to find homes for them. This process ensures that the total number of agents in a single exploration grows monotonically. The overhead for collecting agents from an absorbed tree of size k_i is $O(k_i)$. A careful analysis shows that the total time spent in mergers across the entire execution is bounded by $O(k)$, preserving the overall optimal time complexity. From the above discussion, we have the following results.

▶ **Theorem 7.** *The* `GeneralAsync()` *algorithm achieves the dispersion in anonymous graph for arbitrary configuration in* $O(k)$ *epochs with* $O(\log k + \Delta)$ *bits of memory per agent.*

References

1 Ankush Agarwalla, John Augustine, William K Moses Jr, Sankar K Madhav, and Arvind Krishna Sridhar. Deterministic dispersion of mobile robots in dynamic rings. In *ICDCN*, pages 1–4, 2018. doi:10.1145/3154273.3154294.

2 John Augustine and William K. Moses Jr. Dispersion of mobile robots: A study of memory-time trade-offs. In *ICDCN*, pages 1:1–1:10, 2018. doi:10.1145/3154273.3154293.

3 Rik Banerjee, Manish Kumar, and Anisur Rahaman Molla. Optimizing robot dispersion on unoriented grids: With and without fault tolerance. In *ALGOWIN*, pages 31–45, 2024. doi:10.1007/978-3-031-74580-5_3.

4 Rik Banerjee, Manish Kumar, and Anisur Rahaman Molla. Optimal fault-tolerant dispersion on oriented grids. In *ICDCN*, pages 254–258, 2025. doi:10.1145/3700838.3700842.

5 Prabhat Kumar Chand, Manish Kumar, Anisur Rahaman Molla, and Sumathi Sivasubramaniam. Fault-tolerant dispersion of mobile robots. In *CALDAM*, pages 28–40, 2023. doi:10.1007/978-3-031-25211-2_3.

6 Giuseppe F. Italiano, Debasish Pattanayak, and Gokarna Sharma. Dispersion of mobile robots on directed anonymous graphs. In *SIROCCO*, pages 191–211, 2022. doi:10.1007/978-3-031-09993-9_11.

7 Ajay D. Kshemkalyani. Dispersion of mobile robots on graphs in the asynchronous model. *Theor. Comput. Sci.*, 1044:115272, 2025. doi:10.1016/j.tcs.2025.115272.

8 Ajay D. Kshemkalyani and Faizan Ali. Efficient dispersion of mobile robots on graphs. In *ICDCN*, pages 218–227, 2019. doi:10.1145/3288599.3288610.

9 Ajay D Kshemkalyani, Manish Kumar, Anisur Rahaman Molla, Debasish Pattanayak, and Gokarna Sharma. Dispersion is (almost) optimal under (a) synchrony. In *SPAA*, 2025. doi:10.1145/3694906.3743317.

10 Ajay D. Kshemkalyani, Anisur Rahaman Molla, and Gokarna Sharma. Fast dispersion of mobile robots on arbitrary graphs. In *ALGOSENSORS*, pages 23–40, 2019. doi:10.1007/978-3-030-34405-4_2.

11 Ajay D. Kshemkalyani, Anisur Rahaman Molla, and Gokarna Sharma. Dispersion of mobile robots on grids. In *WALCOM*, pages 183–197, 2020. doi:10.1007/978-3-030-39881-1_16.

12 Ajay D. Kshemkalyani, Anisur Rahaman Molla, and Gokarna Sharma. Dispersion of mobile robots using global communication. *J. Parallel Distributed Comput.*, 161:100–117, 2022. doi:10.1016/j.jpdc.2021.11.007.

13 Ajay D. Kshemkalyani and Gokarna Sharma. Near-optimal dispersion on arbitrary anonymous graphs. *Journal of Computer and System Sciences*, 152:103656, 2025. doi:10.1016/j.jcss.2025.103656.

14 Anisur Rahaman Molla, Kaushik Mondal, and William K. Moses Jr. Byzantine dispersion on graphs. In *IPDPS*, pages 942–951, 2021. doi:10.1109/IPDPS49936.2021.00103.

15 Anisur Rahaman Molla, Kaushik Mondal, and William K. Moses Jr. Optimal dispersion on an anonymous ring in the presence of weak byzantine robots. *Theor. Comput. Sci.*, 887:111–121, 2021. doi:10.1016/j.tcs.2021.07.008.

16 Debasish Pattanayak, Ajay D. Kshemkalyani, Manish Kumar, Anisur Rahaman Molla, and Gokarna Sharma. Optimal dispersion under asynchrony. *CoRR*, abs/2507.01298, 2025. doi:10.48550/arXiv.2507.01298.

17 Debasish Pattanayak, Gokarna Sharma, and Partha Sarathi Mandal. Dispersion of mobile robots in spite of faults. In *SSS*, pages 414–429, 2023. doi:10.1007/978-3-031-44274-2_31.

18 Yuichi Sudo, Masahiro Shibata, Junya Nakamura, Yonghwan Kim, and Toshimitsu Masuzawa. Near-linear time dispersion of mobile agents. In *DISC*, pages 38:1–38:22, 2024. doi:10.4230/LIPIcs.DISC.2024.38.