Agent-Based Distributed Node2vec

Deepak Sujay Gudiseva

A white paper

submitted in partial fulfillment of the

requirements for the degree of

Master Of Science In Computer Science & Software Engineering

University of Washington

2026

Reading Committee:

Dr. Munehiro Fukuda, Chair

Dr. Min Chen

Dr. Bill Erdly

Program Authorized to Offer Degree:

Computer Science & Software Engineering

University of Washington

ABSTRACT

Agent-Based Distributed Node2vec

Deepak Sujay Gudiseva

Chair of the Supervisory Committee:
Dr. Munehiro Fukuda

Graph representation learning algorithms like Node2Vec generate highly accurate topological embeddings but impose severe memory and computational bottlenecks on centralized architectures. This paper presents a scalable, distributed Node2Vec engine engineered natively within the Multi-Agent Spatial Simulation (MASS) framework. By mapping second-order, biased random walks to autonomous mobile software agents, our architecture efficiently samples complex networks while bypassing single-machine memory limitations. We introduce a novel Compute-Node-Centric training paradigm that pairs isolated Skip-Gram neural network optimization with a decentralized, logarithmic tree-reduction synchronization protocol. Empirical evaluations across benchmark graphs (Cora and OGBL-DDI) demonstrate that this distributed engine achieves close predictive parity with industry-standard PyTorch baselines across key ranking metrics like MAP, Recall.. etc. Ultimately, this architecture successfully outperforms linear methods like FastRP in topological accuracy while comprehensively unlocking the spatial scalability required for parallelized, deep graph learning.

TABLE OF CONTENTS

List of Figures

List of Tables

Chapter 1. Introduction

1.1 Overview of Graph Representation Learning

Graph representation learning has emerged as a cornerstone of modern data science, providing the vital mathematical frameworks necessary to analyze complex, interconnected relational data. Unlike traditional tabular datasets where individual entities act entirely independently, graph networks are defined by irregular, non-Euclidean topologies where nodes derive their fundamental meaning from their surrounding connections and paths. Parsing this massive geometric architecture—ranging from sprawling social networks to dense biochemical molecular interactions—requires algorithms capable of recognizing both tight local clusters and broader, over-arching structural roles. Effectively mapping these continuous topological relationships allows data scientists to mathematically quantify how information, influence, or similarity flows through a system. As real-world networks continue to scale exponentially into millions of vertices and billions of edges, efficiently deciphering this relational data has become exceptionally critical for advanced pattern recognition.

To leverage the full capabilities of modern predictive analytics, complex graph topologies must first be algorithmically transformed into low-dimensional, continuous vector spaces through a process known as embedding. Traditional representations, such as high-dimensional, sparse adjacency matrices, are notoriously rigid and computationally prohibitive to process on a massive scale. By carefully projecting the graph's geometric structure into dense mathematical vectors, embeddings systematically force nodes that share highly similar structural neighborhood contexts to geometrically converge. This abstraction is profoundly powerful because standard machine learning classifiers, regression models, and deep neural networks are fundamentally designed to execute math against continuous numerical vectors rather than discrete graph edges.

By capturing the rich, multi-hop topological nuance of a network inside a constrained coordinate array, researchers or social-network scientists can use the embeddings for community detection, role/role-change analysis, link prediction, node classification, influence ranking and anomaly detection by feeding Node2Vec output into KNN, clustering, or supervised models.

1.2 The Node2Vec

Node2Vec has established itself as a premier graph embedding algorithm by ingeniously combining dynamic network exploration with advanced neural network optimization. To appropriately sample the intricate topology of a dataset, the algorithm relies on sequentially simulating biased random walks across the graph structure [3]. As graphically abstracted in Figure 1, a simulated "walker" navigates edge connections between vertices, generating long, continuous sequences of nodes that effectively map the local neighborhood layout [3]. Node2Vec elegantly treats these generated traversal paths analogously to sentences in natural language processing, feeding the sequences directly into a Skip-Gram neural network architecture [5]. This mathematical model iteratively optimizes the final continuous embeddings by maximizing the probability of accurately predicting a surrounding context of nodes given a specific center node.

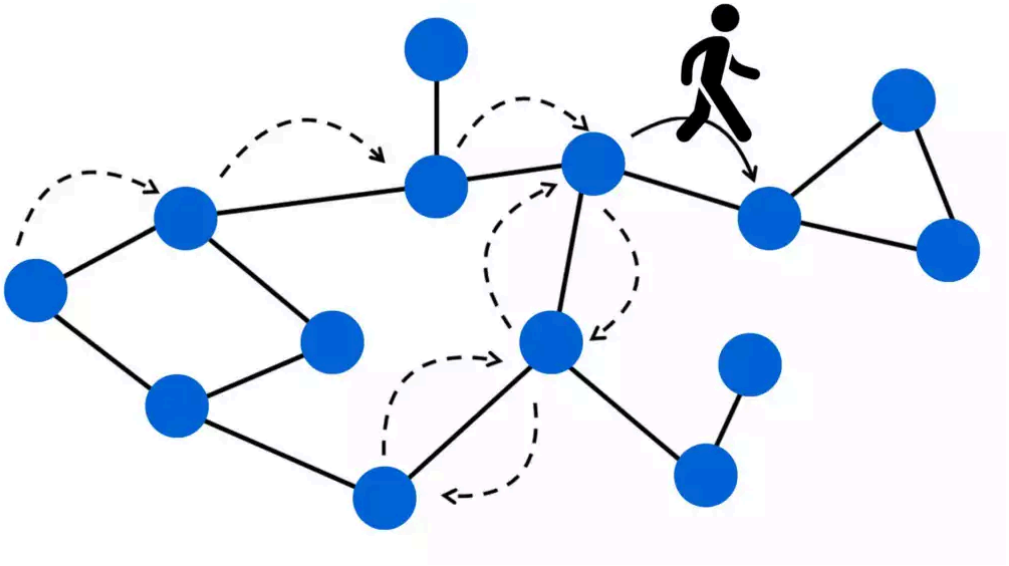

Figure 1. Random walk strategy on a graph network [3].

Despite its superior predictive accuracy and topological flexibility, Node2Vec imposes severe computational and spatial burdens that inherently restrict its scalability on traditional single-machine architectures. To adequately capture a comprehensive structural sample, the algorithm must generate dozens of overlapping random walks of substantial length for every single vertex in the network, mimicking the intensive spatial traversal mechanics shown in Figure 1. This relentless sequence generation consumes immense processing cycles and demands vast reserves of active memory strictly to store the resulting geometric pathways. Furthermore, aggressively training the subsequent Skip-Gram [5] neural network requires continuous, iterative dense matrix updates across massive global embedding arrays via stochastic gradient descent [6]. When applied to industrial-scale datasets comprising millions of interconnected entities, a centralized computing node rapidly exhausts its physical memory limits and processor bandwidth.

1.3 The MASS Library

The Multi-Agent Spatial Simulation (MASS) library is a highly specialized distributed computing framework explicitly engineered to parallelize complex simulations across multi-node hardware clusters [10]. Developed by the Distributed Systems Laboratory at University of Washington, Bothell, the framework abstracts the complexities of low-level socket communications, allowing researchers to seamlessly deploy massive, data-intensive workloads across distributed hardware environments [10]. By securely partitioning data arrays and execution threads across independent Java Virtual Machines (JVMs), MASS inherently bypasses the processing bottlenecks and physical memory ceilings that frequently cripple centralized physical servers. This robust distributed infrastructure acts as an ideal geographic foundation for executing complex algorithms that demand extreme computational scaling and continuous spatial mapping.

To effectively orchestrate these parallel computations, the MASS architecture natively employs a uniquely decentralized spatial paradigm centered entirely around geometric "Places" and mobile "Agents" [10]. Places function as stationary, multidimensional data containers that statically map the foundational environment—such as the discrete vertices of a graph database—stably across the cluster's distributed memory [10]. Operating actively over this static grid, Agents serve as dynamic, programmatic execution units capable of autonomous, intelligent movement [10]. Rather than transmitting massive datasets over the network to a central processing unit, these mobile agents physically migrate across hardware socket boundaries to execute targeted algorithmic logic directly where the spatial data resides. This decoupled relationship between stationary data housing and mobile logic execution provides the exact spatial flexibility required to simulate millions of parallel graph traversals efficiently.

1.4 Problem Statement

While Node2Vec delivers exceptional topological accuracy, its underlying processing mechanics impose severe hardware constraints that critically limit its applicability to modern, large-scale datasets. Generating exhaustive random walks for every network vertex requires immense, localized memory buffers strictly to store the resulting traversal sequences. On a traditional centralized server, attempting to process densely connected, industrial-scale graphs rapidly triggers catastrophic memory bottlenecks, completely preventing researchers from generating embeddings for networks that exceed a single machine's physical hardware capacity.

The partitioned, spatial architecture of the MASS framework presents a distinct opportunity to fundamentally resolve these debilitating memory limitations. By geographically distributing the graph vertices and execution threads across a multi-node hardware cluster, the extreme spatial burden required for both sequence generation and neural network training can be effectively localized and absorbed. This gives explicit spatial scaling and fine-grained locality control—allowing larger walk densities and distributed training. Therefore, the central problem statement of this project is to engineer and integrate a fully distributed Node2Vec engine natively within the MASS framework.

1.5 Project Scope and Objectives

The overarching scope of this capstone is to mathematically design and programmatically integrate a distributed Node2Vec engine specifically tailored for the MASS library environment. Executing this integration will empower researchers to deploy highly complex structural embedding algorithms against expansive graphs that inherently paralyze traditional data science platforms. To accomplish this, the project focuses specifically on mapping the algorithmic

requirements of Node2Vec directly onto the MASS execution paradigm, primarily by utilizing mobile software agents to natively simulate second-order, biased random network walks. By deploying thousands of independent agents concurrently across geographically distributed vertices, the system can systematically harvest massive quantities of topological structure while inherently bypassing single-machine memory restrictions. Compared to other distributed frameworks like Spark [20], MASS was chosen because its agent/place abstraction is better suited for many short, stateful random-walk operations and custom Pack-Reduce-Broadcast synchronization, whereas Spark favor bulk-synchronous, coarse-grained data parallelism and are less natural for walker-agent workflows.

The primary operational objectives defining the success of this distributed architecture are algorithmic correctness and memory scalability. Firstly, the project must establish algorithmic correctness, verifying that the bespoke implementation of second-order random walks, transition probabilities, and Skip-Gram neural network optimization natively within Java executes exactly as intended when evaluated against trusted industry baselines. Secondly, the evaluation seeks to benchmark and prove spatial scalability, verifying that systematically expanding the distributed hardware cluster effectively alleviates isolated memory bottlenecks. Achieving these combined objectives will definitively validate that the engineered framework is scientifically sound while unlocking the architectural capacity to process incredibly dense populations of network traverses.

1.6 White Paper Structure

To systematically explore the engineering and evaluation of this project, the remainder of this white paper is categorically partitioned into four primary chapters. Chapter 2 provides the foundational literature review and technical context necessary to understand the project, detailing the evolution of graph embedding algorithms from basic metrics to Node2Vec, dissecting

distributed machine learning paradigms, and introducing the core spatial architecture of the MASS library. Proceeding to the core technical contribution, Chapter 3 establishes the bespoke Agent-based Node2Vec Implementation; this chapter comprehensively details the Compute-Node-Centric paradigm before sequentially explaining execution phases concerning mobile random walks, logarithmic neural network synchronization, and downstream Link Prediction application bridging. Chapter 4 subsequently presents a rigid, empirical evaluation of this implemented architecture, defining the physical cluster environments and strict evaluation metrics utilized before critically analyzing both predictive learning accuracy and massive structural scalability across multi-node benchmarks. Finally, Chapter 5 synthesizes the overarching conclusions drawn from these performance metrics while proposing critical avenues for future technical optimizations and algorithmic integrations.

Chapter 2. Background & Related Work

2.1 Graph Analytics and Network Topology

In the domain of modern graph analytics, mapping the complex web of real-world relationships requires moving beyond simple node and edge counts to understand deeper architectural patterns. Nodes within a network are rarely isolated entities; rather, their underlying characteristics and latent behaviors are heavily defined by their connectivity within the broader graph structure [1]. According to Hamilton, Ying, and Leskovec, two fundamental principles primarily govern these topological relationships: homophily and structural equivalence [1]. Homophily refers to the statistical tendency of interconnected nodes to share similar characteristics or belong to the same immediate community, often manifesting as dense, highly interconnected local neighborhoods. Conversely, structural equivalence describes nodes that play similar functional or architectural roles within the network, even if they reside in completely different subgraphs and share no direct edges. For example, within a corporate network, homophily tightly connects engineers working within the same immediate department, while structural equivalence mathematically links managers across completely different departments who execute identical hierarchical functions. Traditional graph representations, such as high-dimensional and sparse adjacency matrices, historically struggle to map both of these distinct proximity measures concurrently without suffering from severe computational inefficiencies [1]. Recognizing and mathematically preserving the delicate interplay between these two topological phenomena is essential for understanding how advanced algorithms must simulate network exploration to generate meaningful representations.

Accurately capturing these complex topological features is strictly necessary for establishing high-fidelity predictive modeling in downstream machine learning tasks. Traditional

machine learning architectures typically operate under the assumption that data points are independent and identically distributed (i.i.d.), an assumption that fundamentally collapses when applied to highly correlated relational graph data. When algorithms fail to adequately encode both local neighborhood density (homophily) and global structural roles (equivalence), the resulting predictive models suffer from an acute lack of contextual awareness. For instance, in critical graph applications such as link prediction or dynamic recommender systems, failing to encode homophily might result in missing obvious future connections between locally related entities. Likewise, ignoring structural equivalence severely limits a model's ability to generalize broader functional behaviors or recognize overarching network symmetries. By successfully compressing these rich structural topologies into dense, continuous vector spaces, we provide standard classifiers and neural networks with the nuanced relational context they require to operate accurately. This transformative abstraction bridges the critical gap between discrete, irregular graph structures and the continuous mathematical domains where modern deep learning algorithms thrive. Ultimately, the predictive ceiling of any graph-centric intelligence system—such as the distributed environment built in this project—is directly bound by the topological precision and structural fidelity of its underlying node embeddings.

2.2 Evolution of Graph Embeddings

Following the necessity to map complex topological features, early and competitive embedding techniques like Fast Random Projection (FastRP) emerged as highly scalable solutions. FastRP fundamentally relies on sparse random matrix multiplications and the Johnson-Lindenstrauss lemma to project high-dimensional adjacency matrices into lower-dimensional continuous spaces [11]. Within this algorithmic framework, the capture of homophily and structural equivalence is largely dictated by tuning a transition matrix that aggregates features across multi-hop

neighborhoods. By mathematically modulating the influence of immediate topological neighbors versus distant connections, FastRP implicitly weights local cluster density against global structural roles. However, because FastRP executes these aggregations through generalized, linear matrix operations, it inherently forces a rigid compromise between these two proximity measures rather than dynamically exploring the network. While FastRP is computationally exceptional, this linear rigidity mathematically limits its ability to disentangle highly nuanced homophilic clusters from structurally equivalent nodes in heterogeneous graphs. Consequently, more adaptive, non-linear exploration strategies were required to capture these relationships with higher fidelity.

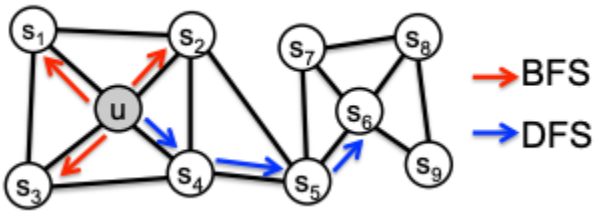

Figure 2. BFS and DFS search strategies from node u [2].

To overcome the limitations of linear projections, researchers developed Node2Vec, a highly adaptive embedding framework that utilizes biased random walks to dynamically sample graph topologies [2]. Node2Vec operates on the principle that simulating network exploration is analogous to reading a text document, where sequences of connected nodes form descriptive sentences [3]. As illustrated in Figure 2, the algorithm designs its random walks to smoothly interpolate between two extreme search strategies: Breadth-First Search (BFS) and Depth-First Search (DFS). A DFS-oriented walk aggressively explores the immediate, localized neighborhood surrounding a starting *node u,* which meticulously maps local cluster densities and effectively captures homophily. Conversely, a BFS-oriented walk rapidly ventures deep into the

broader network, prioritizing the discovery of overarching macroscopic structures and functional roles to accurately capture structural equivalence [1]. By dynamically blending these two distinct traversal strategies, Node2Vec effectively decouples the modeling of local community tightness from global architectural symmetries. Node2Vec provides biased random walks that capture both homophily and structural roles, and its walk-centric design maps naturally onto MASS's agent/place model so we can generate walks in parallel with walker agents and then train in a computer-node-centric fashion.

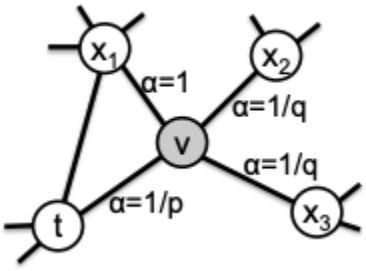

Figure 3. Illustration of the random walk procedure in node2vec [2].

The interpolation between BFS and DFS traversals in Node2Vec is governed mathematically by a second-order random walk utilizing two tunable hyperparameters: the return parameter p and the in-out parameter q [2]. As depicted in Figure 3, consider a random walker that has just traversed from *node t* to *node v*, and is now evaluating its next transitional step to a neighboring node. The unnormalized transition probability α is dictated by the walker's distance from the previous *node t* [3]. If the walker attempts to return directly to *node t*, the probability is weighted by $1/p$, implying that a high p discourages backtracking and promotes forward exploration. If the walker evaluates a *node* x_1 that also shares a direct edge with *node t*, the transition weight is 1, promoting localized BFS behavior and structural equivalence. Crucially, if the walker evaluates a *node* x_2 or *node* x_3 that is entirely disconnected from *node t*, the transition

is scaled by $1/q$; therefore, a low q strongly encourages outward DFS traversal to capture homophily. This hyperparameter-driven transition matrix grants Node2Vec the profound flexibility to adapt its structural sampling to the specific topological requirements of highly diverse datasets.

Once the biased random walkers have generated expansive collections of node sequences, Node2Vec leverages the Skip-Gram neural network architecture—originally developed for Word2Vec—to optimize the final embeddings [4]. The Skip-Gram model functions by maximizing the probability of predicting a surrounding "context" of nodes given a specific "center" node observed in a random walk [5]. Mathematically, this involves updating two weight matrices (representing input and output embeddings) by calculating the dot product of the center node's continuous vector against its neighbors' vectors. Optimization is driven by Stochastic Gradient Descent (SGD), which iteratively adjusts the matrix weights using backpropagation to minimize a binary cross-entropy loss function [6]. To ensure computational feasibility over massive vocabularies, negative sampling [2] is employed, forcing the model to heavily penalize dot products with randomly selected, disconnected nodes while rewarding structurally verified context pairs. This rigorous neural optimization process systematically forces nodes that frequently co-occur in similar structural contexts to converge geometrically in the continuous vector space.

Comparing Node2Vec directly against linear methods like FastRP reveals a stark tradeoff between unparalleled predictive accuracy and raw computational speed. Because FastRP relies entirely on sparse matrix multiplications rather than iterative neural learning, it executes dramatically faster and maintains this high-speed execution. Conversely, Node2Vec consistently achieves superior accuracy benchmarks in downstream tasks because its Skip-Gram optimization

meticulously maps non-linear, multi-dimensional relationships. However, achieving this precision makes Node2Vec exceptionally CPU and memory intensive, as it must relentlessly simulate millions of second-order random walks then constantly update immense embedding matrices via stochastic gradient descent. On a traditional single-machine architecture, scaling these intensive walk generations and continuous gradient updates across massive networks rapidly exhausts available hardware resources. By distributing Node2Vec across the MASS framework, we achieve critical spatial scalability that inherently bypasses these centralized limitations. This distributed paradigm seamlessly parallelizes the computational load, allowing the system to simultaneously generate geometrically more walks across a cluster while preserving the algorithm's superior topological fidelity.

2.3 Distributed Machine Learning Basics

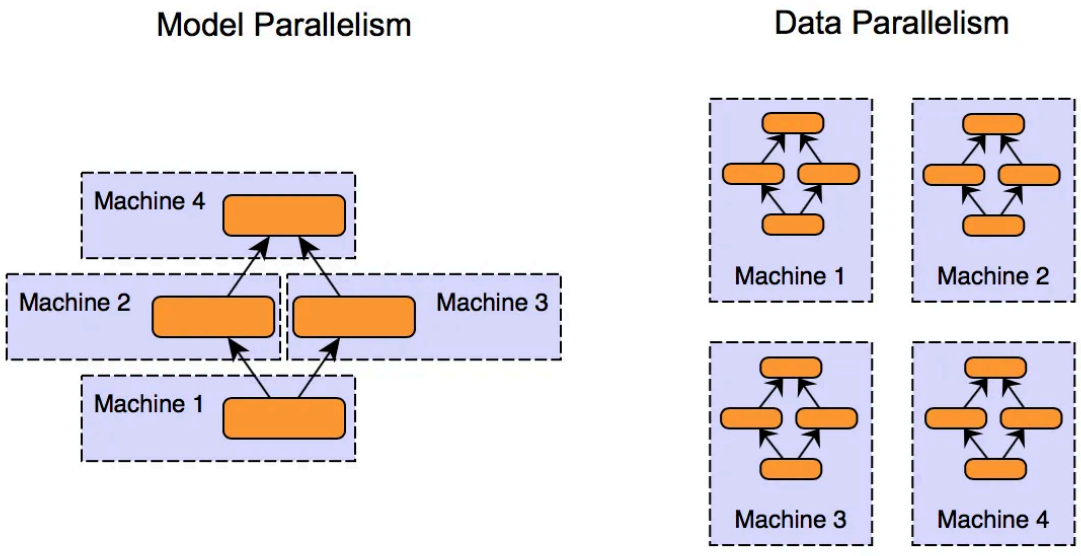

Figure 4. Model Parallelism Vs Data Parallelism [8].

As the computational and spatial demands of modern neural networks exponentially increase, distributing the training workload across multiple autonomous machines has become a fundamental necessity. In distributed deep learning environments, strategies for parallelizing these advanced workloads are broadly categorized into two primary architectures: data parallelism and model parallelism [7]. As clearly illustrated in Figure 4, model parallelism physically partitions a singular neural network architecture, wherein different consecutive computational layers or geometric blocks of the model are segmented across distinct computing nodes [8]. This fragmented approach is predominantly reserved for exceptionally massive architectures, such as modern large language models, whose parameter counts physically exceed the memory capacity of any single machine [8]. Conversely, data parallelism operates by meticulously replicating the exact same, complete neural network model across every available physical machine in the distributed cluster [7]. Instead of fracturing the neural network, the underlying training dataset itself is sliced into smaller localized partitions, allowing each independent machine to train its complete model replica exclusively using its isolated slice of data [7]. Once each autonomous node effectively completes a localized training epoch, the disparate machines must mathematically synchronize their findings across the network to ensure the global model successfully learns from the collective dataset.

The decision between implementing model parallelism or data parallelism ultimately revolves around a strict architectural tradeoff between single-machine memory constraints and cascading network communication overhead. Model parallelism gracefully neutralizes severe hardware memory thresholds, ensuring that no single machine is ever burdened with hosting a multi-billion parameter network in its local RAM [8]. However, because the sequential neural network layers are geographically separated across a cluster, a massive communication penalty is

continuously incurred, as intermediate tensor activations must travel over local area networks during every forward and backward pass [9]. Data parallelism strategically circumvents this sequential lockstep by strictly allowing each node to compute its forward and backward passes independently, dramatically accelerating localized computational throughput [9]. While this isolated processing is exceptionally fast, it inadvertently introduces a demanding network synchronization requirement where millions of localized gradient updates must be simultaneously aggregated [9]. If the parameter matrix is exceptionally large, this sudden burst of gradient synchronization traffic can quickly overwhelm centralized parameter aggregation servers, transforming the highly intended parallel acceleration into a debilitating network latency bottleneck [9].

For the agent-based implementation of Node2Vec within the MASS framework, this project deliberately adopts a data parallelism architecture to maximize spatial scalability. Within the specific context of generating graph embeddings, the underlying training dataset is dynamically constructed using the simulated random walks rather than static relational tables. Data parallelism uniquely allows the system to spatially partition this sampling workload, dispatching millions of distributed biased walks simultaneously across the network without relying on sequential layer rendering. By fully replicating the Skip-Gram neural network on every isolated machine, the framework inherently bypasses the intense layer-to-layer communication hurdles that typically paralyze model parallelism computations. To successfully unify these completely independent models, distributed data parallelism heavily relies on rigid synchronous updates, where all nodes algorithmically halt at a designated barrier to mathematically average their localized weight matrices [9]. Data parallelism pairs perfectly with the spatial traversal mechanics of MASS, decoupling the graph's overall size from

single-machine constraints and providing the scalability required to process millions of independent random walks in massive walks in parallel.

2.4 The MASS Framework Architecture

The Multi-Agent Spatial Simulation (MASS) framework is fundamentally designed to facilitate parallel computing across distributed cluster environments by employing a uniquely localized spatial paradigm [10]. At the core of its architecture, MASS mathematically partitions and maps large-scale simulation environments into discrete, stationary data containers known as Places [10]. These Places typically form a multidimensional grid or network distributed evenly across the physical RAM of multiple independent computing machines. To actively interact with this localized spatial data, MASS introduces dynamic, mobile execution units referred to as Agents [10]. Unlike the static stationary Places, these Agents possess the profound capability to autonomously migrate across the physical hardware boundaries of different computing nodes to execute localized processing. By algorithmically transferring the computational thread directly to the data's geographic residency—rather than transferring massive datasets back to a centralized processing node—the framework drastically reduces detrimental network synchronization overhead [10]. This elegant orchestration of stationary data grids and mobile execution agents inherently bypasses the severe memory and bandwidth bottlenecks typically associated with massive single-machine processing.

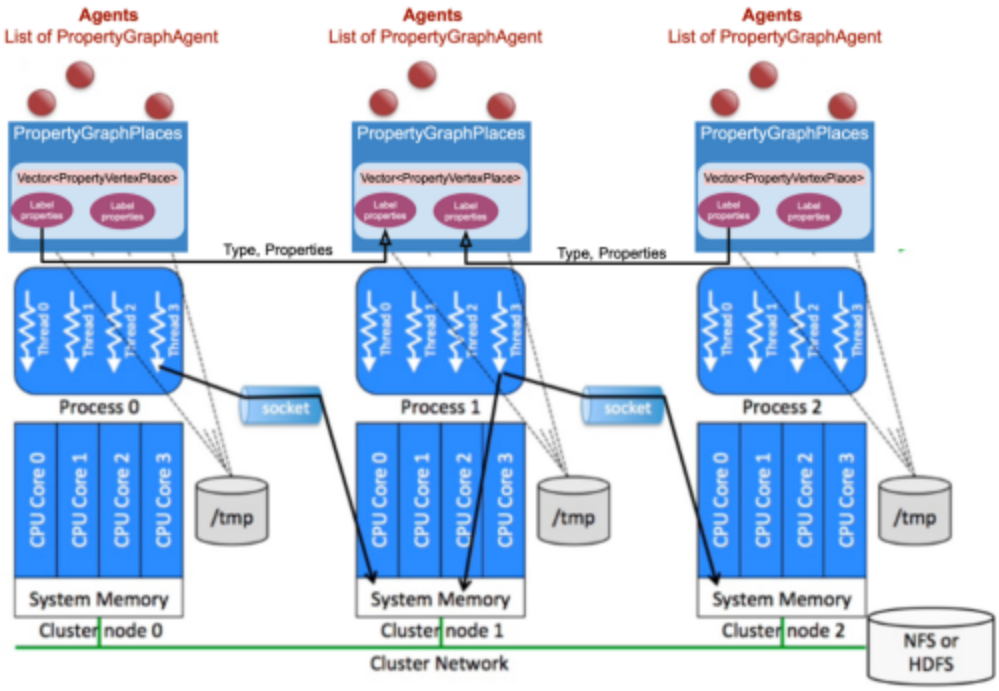

Figure 5. Enhanced agent-based graph DB system with PropertyGraphPlaces [11].

Building upon this foundational spatial architecture, the MASS library has been extensively upgraded to natively support sophisticated graph database structures and analytics [11]. As depicted in Figure 5, the traditional spatial multidimensional grids have been entirely replaced by an interconnected mapping of VertexPlaces and GraphPlaces that accurately mirror complex network topologies. Specifically, the introduction of PropertyGraphPlaces empowers each stationary distributed node to encapsulate rich, multifaceted string identifiers and complex graph-theoretic data attributes. These intricate vertex objects are meticulously distributed across the active computing nodes, deliberately partitioning densely connected graphs to minimize disruptive cross-JVM edge traversals. To actively navigate these distributed topologies, the MASS database framework deploys specialized PropertyGraphAgents inherently capable of traversing the interconnected vertices to execute advanced graph mining algorithms. This enhanced agent-based database environment brilliantly bridges the gap between static spatial

graph storage and dynamic parallelized execution, seamlessly supporting complex operations like link prediction and topological embeddings.

The active execution of these mobile entities across the distributed environment is strictly governed by a rigorously synchronized, lockstep coordination mechanism intrinsic to the MASS ecosystem [10]. The overarching cluster control is ultimately orchestrated through two vital, sequential computational barriers: the *callAll()* and *manageAll()* lifecycle functions. When the distributed coordinator invokes *callAll()*, every active agent deployed across the entire physical cluster simultaneously executes a targeted methodological instruction safely within its current local context. Immediately following this computational phase, the execution thread strikes the critical *manageAll()* barrier, which securely processes all pending state modifications, coordinates socket communications, and physically migrates traveling agents to their newly targeted JVM instances. This bipartite synchronization loop actively ensures that millions of independent executing parallel threads remain mathematically deterministic without inadvertently generating race conditions or asynchronous data corruption. Within the explicit context of this capstone project, this rigid lockstep loop guarantees that millions of distributed random walkers progress in perfect structural unison, providing an intrinsic layer of operational tracking without generating extraneous network polling overhead.

Chapter 3. Agent based Node2vec Design & Implementation

3.1 System Architecture: The Compute-Node-Centric Paradigm

Building directly upon the spatial scalability of data parallelism discussed in Chapter 2, the distributed Node2Vec engine engineered for this project adopts a strict Compute-Node-Centric architectural paradigm. Rather than attempting to fragment the neural network layers across the cluster as necessitated by model parallelism, this architecture strategically instantiates a singular, fully self-contained static Skip-Gram model securely within each separate Java Virtual Machine (JVM). By replicating the entire mathematical embedding matrix on every active hardware node, the system empowers each local computing machine to independently process massive quantities of structural network data without paralyzing the cluster with continuous network communication. This autonomous, JVM-level local model acts as the foundational engine for the system's overarching multiphase execution pipeline, which is systematically decomposed throughout the remainder of this chapter. Phase I, delineated in Section 3.2, explores the distributed generation of biased random walks, focusing heavily on how mobile agents actively traverse the fragmented graph topology to harvest localized sequence data. Following this traversal sampling, Phase II, detailed in Section 3.3, dissects the mathematical training and synchronization of the distributed Skip-Gram networks, highlighting how isolated JVMs mathematically merge their independent weights to forge a globally coherent embedding matrix. Finally, Phase III (Section 3.4) briefly explores the ingestion of these generated embeddings into practical downstream applications, specifically targeting link prediction classification tasks. Ultimately, this compute-node-centric approach brilliantly isolates the heavy neural training workload from the granular spatial complexities of the distributed graph.

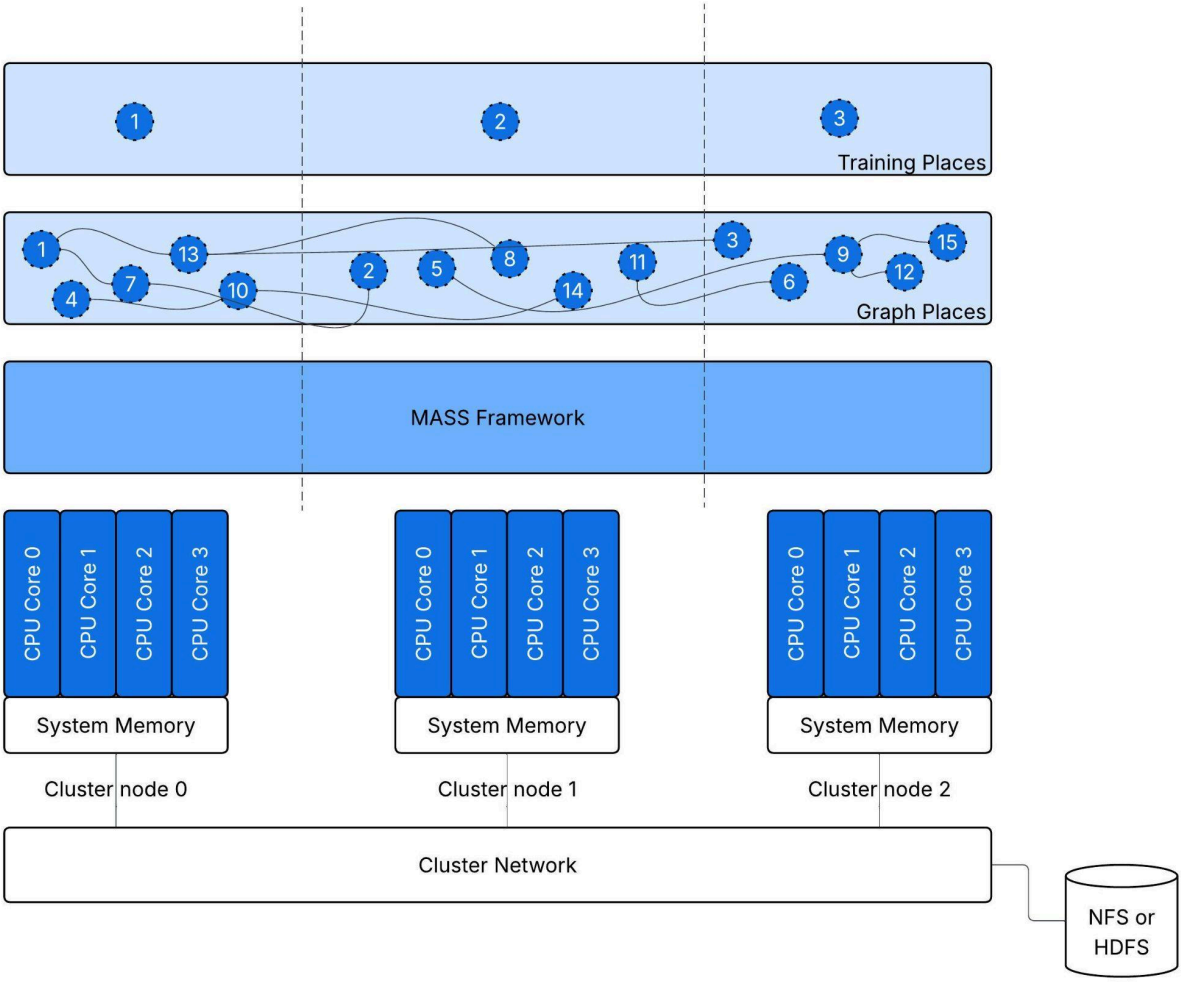

Figure 6. Training Place Vs Graph Place.

To physically orchestrate this JVM-centric training isolation within the MASS framework, the system architecture explicitly segregates the spatial environment into two distinct mapping tiers: Graph Places and Training Places. As clearly illustrated in Figure 6, The Graph Places fundamentally represent the granular, highly distributed vertices of the underlying dataset, spanning horizontally across the network with interconnected, complex topological edges. During Phase I of the architecture, millions of mobile MASS agents continuously migrate back and forth across these lower-tier Graph Places to locally execute the biased random walks

required for topology sampling. However, attempting to execute heavy neural network tensor updates directly from within these granular Graph Places would instantly trigger debilitating communication bottlenecks and implementation complexities. To circumvent this, the architecture introduces a separate tier of overarching Training Places, wherein exactly one macroscopic Training Place is instantiated per computing node. Once the mobile agents successfully complete their walks traversing the bottom-tier Graph Places, they deposit their harvested node sequences directly into the isolated memory buffer of the top-tier Training Place presiding over their current machine. Consequently, during Phase II, the entire localized Skip-Gram optimization process is executed exclusively within the bounds of these sparse, node-level Training Places rather than the highly fractured graph vertices. This strict geographic separation of spatial graph traversal from mathematical neural network training ensures that the algorithmic complexity of the graph does not degrade the high-speed execution of the local model updates.

Because each autonomous JVM trains a mathematically isolated local model using only the specific random walks generated on its physical machine, the system requires a rigid synchronization strategy to coax these disparate networks toward global convergence. This vital global alignment is algorithmically orchestrated through a strict, epoch-based synchronization barrier defined holistically as the Pack-Reduce-Broadcast lifecycle. Once a local machine successfully processes its designated batch of random walks, its Training Place temporarily halts stochastic execution to pack its newly computed neural network weights into a payload. Following this packing phase, the entire distributed cluster engages in a highly coordinated reduction hierarchy, strategically migrating these massive payloads across the network geometry to average their combined delta weights. After the cluster finishes mathematically fusing these

isolated matrices into one singular, globally averaged embedding model, the optimized weights are systematically broadcasted back down to every independent JVM across the entire cluster. This synchronized transmission explicitly guarantees that every local machine initiates the next subsequent computational epoch utilizing an identical, globally informed understanding of the network's overarching structure. By intentionally deferring heavy socket communication to the end of predetermined computational batches, the architecture safely ensures reliable, distributed neural network convergence. Ultimately, this cyclical lifecycle cadence elegantly balances the blistering speed of isolated data parallelism with the strict mathematical necessity of unified global intelligence.

3.2 Phase I: Distributed Biased Random Walks

The foundational phase of the distributed Node2Vec engine involves generating millions of biased random walks across the partitioned spatial network. To execute this massive traversal workload, the architecture deploys specialized *PropertyGraphAgents* designed to navigate autonomously between interconnected vertices. Because Node2Vec strictly requires a second-order Markov chain, an evaluating agent must explicitly know the previous vertex it departed from to accurately calculate the next topological step. To successfully retain this historical context while migrating across distant JVM boundaries, each mobile agent is equipped with a data payload known as *WalkerArgs*. This crucial internal payload acts as the agent's spatial memory, continuously tracking variables such as the *prevPlaceIndex* and an array of *prevNeighborIds*. As graphically illustrated in Figure 7, the blue circles represent the stationary Graph Places housing the network vertices, while the smaller red circles denote these mobile agents carrying their historical payloads across the distributed infrastructure. By physically

carrying this topological history, the agents can seamlessly evaluate complex multi-hop proximities regardless of which physical computing node they currently reside on.

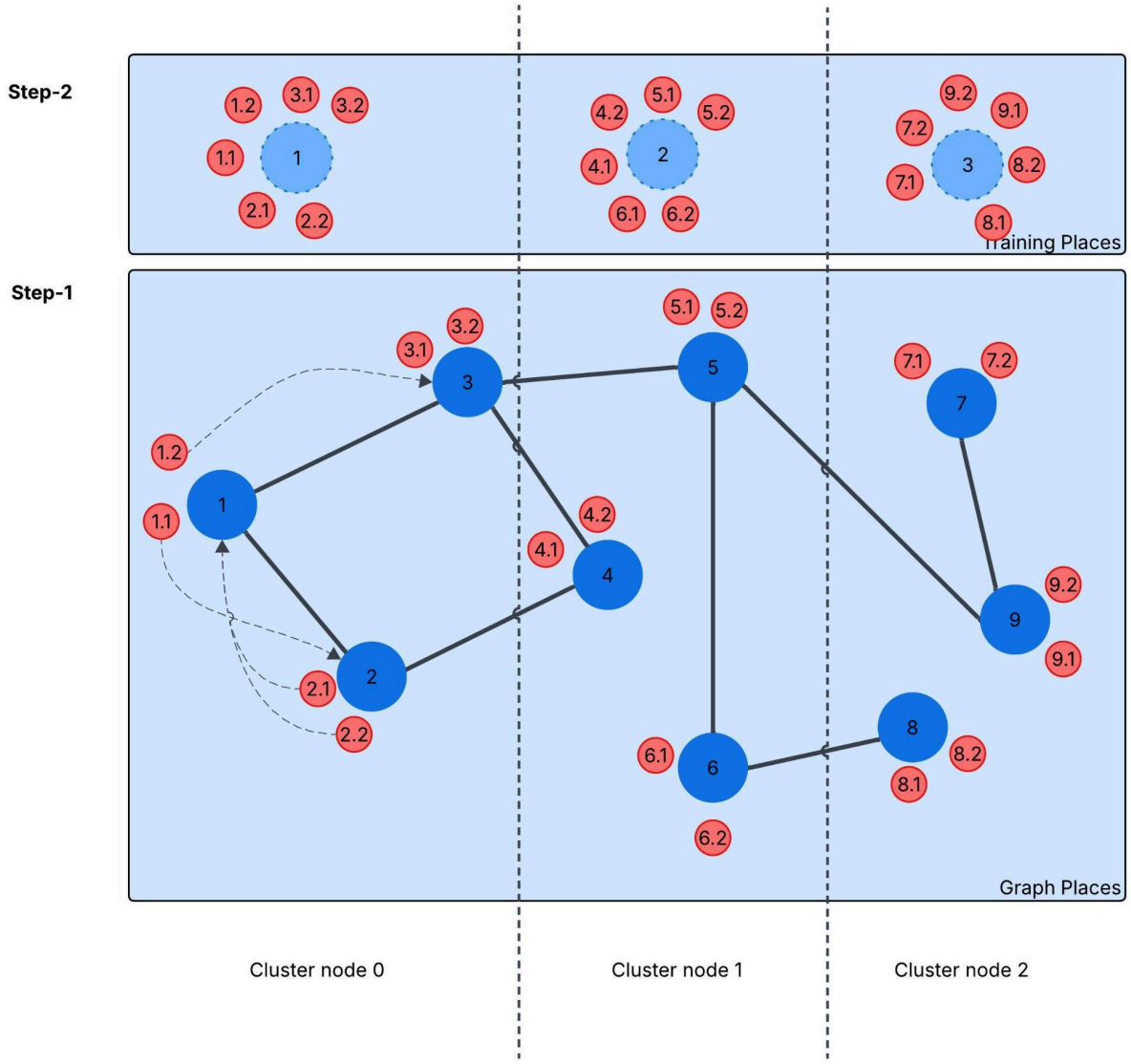

Figure 7. Biased Random Walk Illustration using MASS Agents (WalksPerNode = 2).

Armed with this localized topological memory, the *PropertyGraphAgent* autonomously calculates the normalized transition probabilities for its next movement using the logic outlined in Algorithm 1. When an agent residing at the current vertex evaluates a potential neighboring

target, it actively cross-references the candidate against the array of *prevNeighborIds* stored within its payload. If the candidate target is exactly the previous node the agent just departed from, the unnormalized transition weight is mathematically scaled by the inverse of the return parameter $(1/p)$. If the prospective target is discovered within the payload's array of previous neighbors—indicating a localized structural triangle—the transition probability is scaled neutrally by a factor of 1. Conversely, if the candidate target is entirely absent from the historical payload, implying it resides further outward in the uncharted network, the transition is scaled by the inverse of the in-out parameter $(1/q)$. These dynamically computed weights are then seamlessly multiplied by the underlying graph database's structural edge weights to establish a definitive, stochastic routing array. Finally, the agent executes a biased random selection against this computed probability distribution, definitively choosing its next destination and updating its payload before invoking the MASS migration command.

Algorithm 1 Agent walkStep() Execution Logic

Require: Current Vertex v, WalkerArgs payload (prev_node t, prev_neighbors N_t)

1: **let** N_v = *v.getNeighbors()*
2: **let** *transition_weights = []*
3: **for each** candidate_node x **in** N_v **do**
4: **if** $x == t$ **then**
5: $\alpha = 1 / p$ ▷ Return parameter bias
6: **else if** $x \in N_t$ **then**
7: $\alpha = 1$ ▷ Local neighborhood bias
8: **else**
9: $\alpha = 1 / q$ ▷ In-out exploration bias
10: **end if**

11: *transition_weights.append(α . edge_weight(v, x))*

12: **end for**

13: *next_node = weightedRandomSelect(N_v, transition_weights)*

14: *payload.update(prev_node = v, prev_neighbors = N_v)*

15: *agent.migrate(next_node)*

The decision to actively store and transport the previous node's explicit neighbor list within the *WalkerArgs* payload was a deliberate architectural compromise. Alternatively, the framework could have utilized dynamic "triangle checks," wherein a primary walker temporarily halts and spawns secondary reconnaissance agents to query the original node regarding its connectivity to the prospective targets. While this alternative elegantly minimized mobile memory footprints, the geometric explosion of spawning millions of microscopic sub-agents for every single step of every random walker might instantly paralyzed the cluster's synchronization lifecycle. By forcing the primary agent to continuously carry the topological state of its previous location, the system entirely bypasses the need for these cascading network queries. However, this explicit design choice inherently inflates the serialization weight of the mobile agent. As the agent traverses the network, shifting this heavy array of strings and identifiers across physical JVM boundaries introduces a substantial socket communication overhead.

Throughout the duration of this continuous spatial migration, the *PropertyGraphAgent* meticulously records the unique identifier of every vertex it actively traverses. Instead of discarding this sequence or attempting to blindly transmit it back to a central master node, the agent retains the accumulating pathway of nodes entirely within its internal memory. Once the agent successfully executes a predefined number of maximum traversal steps, the biased random walk is considered structurally complete, and the agent must securely offload its harvested data

at its source node. To maximize decentralized efficiency, the terminating agent deposits its completed localized pathway directly into the overarching Training Place residing on its source physical computing node. This critical offloading mechanism ensures that the dense sequence arrays are safely aggregated within the isolated *allLocalWalks* buffer of the machine's static *Node2VecLocalModel* discussed in Section 3.3. Crucially, this geographic data dumping perfectly aligns the extracted topological graphs with the neural network parameters waiting to train on them. By finalizing the walk cycle entirely at the physical edge of the network, the architecture successfully concludes Phase I computation without inciting any centralized network congestion.

3.3 Phase II: Distributed Skip-Gram and Synchronization

During Phase II of the architecture, the massive arrays of random walks deposited into each local JVM are fed through a distributed Skip-Gram neural network to learn the final multidimensional node embeddings. To execute this natively, each machine independently runs stochastic gradient descent against its local batch of node pairs, mathematically maximizing the probability of predicting valid topological contexts while aggressively penalizing randomly sampled negative nodes. Because these continuous Skip-Gram matrix updates are highly localized, actively verifying that the entire distributed system is safely converging requires a uniquely decentralized approach to calculating the mathematical error. Rather than polling remote workers continuously over the network, each local model inherently tracks its own binary cross-entropy loss and total processed pair count within a rigidly synchronized execution block during the training cycle. Once an overarching training epoch completely concludes, the central Master node simply issues a lightweight, global tally command to aggregate these macroscopic tuples from the outer cluster nodes. By gracefully summing these independently tracked local computations and dividing

them by the global pair count, the Master node intelligently extracts an accurate global training loss metric without inadvertently introducing any mid-epoch network latency.

While distributing the neural network training elegantly parallelizes the heavy computational workload, it inadvertently creates a severe synchronization dilemma when the local models inevitably attempt to unify their learned intelligence. To ensure the distributed network reliably converges into a single, cohesive geometric embedding array, the independent local models must periodically halt their training and mathematically average their localized matrix weights across the cluster. Initially, a naive synchronization approach mandated that every single worker node directly transmit its dense embedding array to the Master node simultaneously for centralized aggregation. However, as the dimensions of the graph vocabulary and the physical scale of the cluster expanded, this simultaneous transmission resulted in catastrophic network congestion. Furthermore, the Master node inherently lacked the physical JVM RAM capacity to hold the unaggregated raw weight arrays of dozens of machines in memory concurrently, routinely suffering from violent Out-Of-Memory (OOM) crashes. To successfully bypass this centralized memory ceiling, the training architecture required a sophisticated aggregation protocol that pushed the massive mathematical additions explicitly out to the edges of the distributed network.

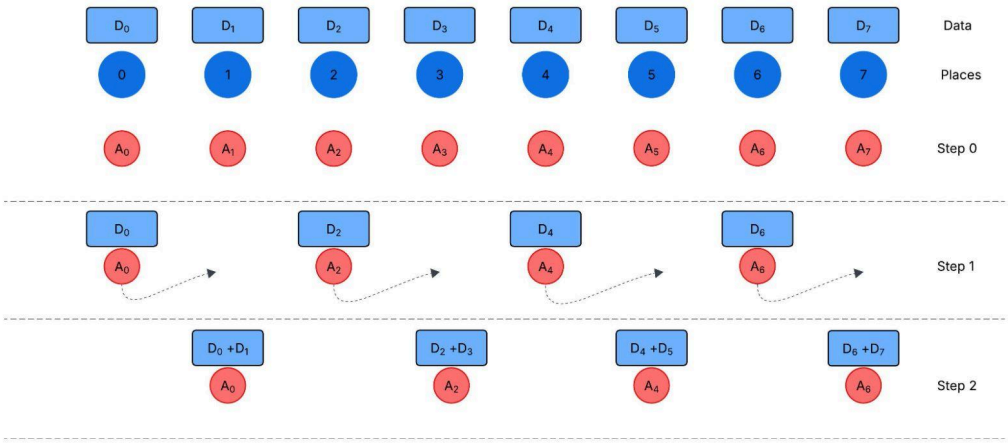

Figure 8. Illustration of Reduce All Architecture.

To completely resolve the memory bottlenecks at the coordinator level, the framework effectively employs a distributed, logarithmic tree-reduction architecture—designated as reduceAll—which leverages the mobile MASS agents to sequentially merge the global network weights in parallel [14]. As clearly depicted in the geometric workflow of Figure 8, the reduction executes across distinct consecutive steps, mathematically halving the active payload footprint during each structural iteration. During Step 0, every active node in the cluster possesses its own localized geometric data arrays (D_0....D_7) and a corresponding mobile agent (A_0....A_7) prepared to execute the synchronization protocol. Transitioning into Step 1, the framework algorithmically bisects the cluster space, commanding half of the mobile agents to operate as outbound senders (e.g., A_0, A_2) while migrating toward their assigned receiver destinations. Step 2 illustrates the mathematical conclusion of this single structural iteration, where the migrating sender agents successfully arrive across the network and directly combine their carried tensor payloads with the receivers' resident data blocks (e.g., $D_0 + D_1$). This meticulously structured pairing phenomenon is executed concurrently across the cluster using the elegant bitwise logic detailed natively in Algorithm 2. When an iteration initiates, an agent dynamically computes its

designated role at Line 1 using the binary shift operation (*((placeId >> iterNo) & 1) == 1*). If this topological evaluation dictates the agent is stationed at a receiver node, it immediately executes Line 3 (*agent.kill()*) to gracefully clear computing capacity for the incoming network traffic. Conversely, an active sender securely packages its neural network tensor and invokes the MASS *migrate(targetPlaceId)* command at Line 7 to traverse the physical JVM computing boundaries. Upon successfully transferring to its targeted receiver place, the sender agent seamlessly executes the element-wise matrix summation defined at Line 12, effectively unifying the distributed weights utilizing strictly localized edge-computed resources.

Algorithm 2 Distributed Tree Reduction (reduceAll) Agent Logic

---

Require: Current iterNo, Agent's current placeId

1: **let** *isReceiver = (((placeId >> iterNo) & 1) == 1)*
2: **if** *isReceiver* **then**
3: *agent.kill()* // Terminate to open receiver capacity
4: **else**
5: **let** *targetPlaceId = placeId + (1 << iterNo)*
6: **let** *payload = getReduceAllData()* // Package neural network embedding weights
7: *agent.migrate(targetPlaceId)* // Migrate data payload across network
8: **end if**
9:
10: // Upon successful arrival at the designated receiver place:
11: **let** *local_weights = currentPlace.getReduceAllPlaceData()*
12: **let** *merged_weights = local_weights + payload* // Element-wise matrix summation
13: *currentPlace.setReduceAllPlaceData(merged_weights)*

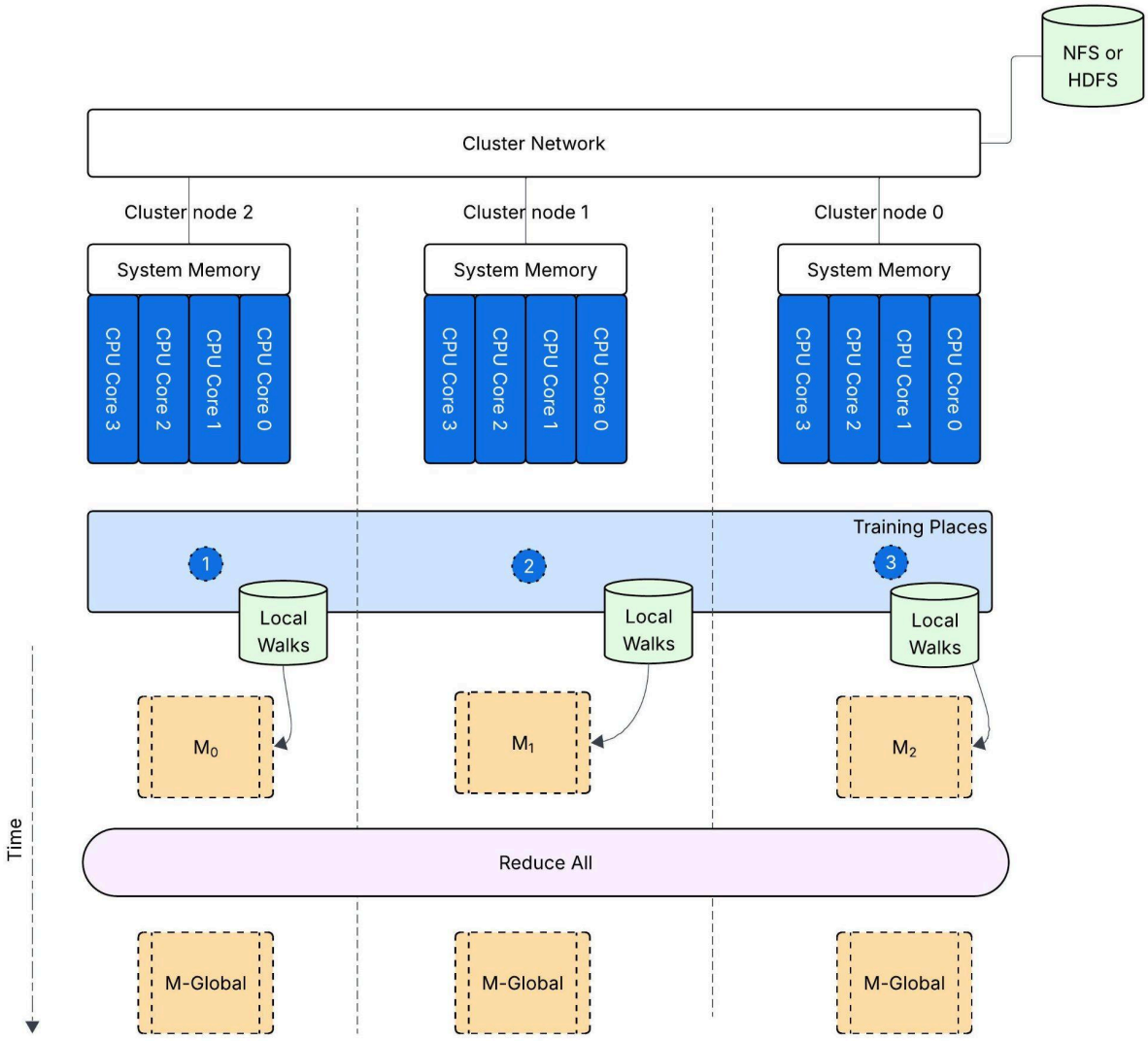

Figure 9. Architecture of Skip-Gram Training.

Although the logarithmic tree reduction effectively eliminates global parameter server crashes, arbitrarily halting the distributed cluster to perform this synchronization inherently introduces a significant data distribution disparity. During Phase I, upon successfully completing a random walk, each mobile agent systematically returns to its original source node to deposit its harvested sequence. Consequently, the total volume of localized walks aggregated on any given computing node is strictly dictated by the number of Graph Places assigned to that specific JVM

multiplied by the universal *walks_per_node* parameter. Because the framework's baseline graph partitioning process may distribute vertices unevenly across the physical cluster, certain JVMs will inevitably compile substantially larger localized datasets than their neighboring nodes. Figure 9 provides a comprehensive structural mapping of the overarching Skip-Gram lifecycle engineered to harmoniously synchronize these wildly disparate data pools. As observed in the diagram's temporal workflow, the unevenly scaled data batches—represented abstractly as the local models M_0, M_1, and M_2—are permitted to train efficiently in complete isolation within their respective Training Places before simultaneously encountering the unified *ReduceAll* communication barrier, which subsequently calculates and broadcasts the singular, globally averaged $M-Global$ tensor array back down sequentially to the workers. To rigorously guarantee that these independent JVM nodes consistently strike this synchronization barrier at exactly the same logical progress threshold, the architecture implements the proportional batching methodology outlined in Algorithm 3. At Line 2 of the algorithm, the global orchestrator translates a user-defined threshold into a globally mandated *sync_interval* that algorithmically dictates the exact topological pacing for the entire cluster. To gracefully accommodate the varying data volumes present across the different simulation servers, each isolated JVM executes Line 6 to dynamically calculate its own *batches_to_train* limit by multiplying this global interval strictly against the localized volume of vertices currently resident in its memory buffers. Establishing this proportionally scaled context allows the system to compute a mathematically flawless execution boundary at Line 10 (*end_idx*), explicitly preventing it from going out of bounds. This highly resilient algorithmic boundary guarantees that all autonomous machines complete their respective proportional data shards concurrently, preserving strict mathematical alignment prior to executing the distributed matrix reduction cascade.

Algorithm 3 Proportional Batching Boundary Calculation

Require: global_sync_frequency (0.0 to 1.0), local_walks array, walks_per_node

```
// Calculate the globally mandated proportional batch cadence
let sync_interval = (walks_per_node - 1) * (1 - global_sync_frequency) + 1.0

// Determine strictly local chunk limits based on resident data volumes
let local_vertices_present = local_walks.size() / walks_per_node
let batches_to_train = sync_interval * local_vertices_present

// Establish perfectly safe execution boundaries for the current epoch
let start_idx = current_walk_index
let end_idx = min(start_idx + batches_to_train, local_walks.size())

// Execute isolated Skip-Gram SGD
train_local_subset(local_walks[start_idx : end_idx])
current_walk_index = end_idx
```

3.4 Phase III: Downstream Applications and Link Prediction

Once the continuous geometric embeddings are generated and safely stored back into the distributed network vertices, they can be actively consumed by practical, downstream machine learning applications. To rigorously evaluate the predictive fidelity of the Node2Vec output, this framework leverages a K-Nearest Neighbors (KNN) classification algorithm, an analytical capability previously integrated into the MASS infrastructure by Hotchandani for link prediction analysis [11]. As visually abstracted in Figure 10, the KNN algorithm predicts the properties of an unknown target node by geometrically calculating its continuous coordinate proximity to the

K closest entities residing within the newly formed embedding space [17]. The algorithm mathematically determines these spatial associations utilizing standard distance metrics and assigns an interaction classification based on the localized neighborhood's structural majority [17]. As explicitly demonstrated in Figure 10's binary classification scenario, tuning this hyperparameter *K* drastically alters the predictive outcome; evaluating merely the immediate inner cluster at *K=3* groups the unknown yellow node with the localized majority of "Class B" green triangles, whereas expanding the perceptive boundary to *K=7* shifts the broader categorical majority to the "Class A" red stars [17]. This visual variance perfectly underscores how selecting an aggressively small *K*-value makes predictions vulnerable to localized statistical noise, while a disproportionately large *K*-value artificially smooths decision boundaries and potentially obscures genuine structural nuance [17]. By computing these KNN distance metrics directly against the generated Node2Vec vectors, the system can definitively ascertain if the decentralized neural network successfully extracted the authentic topological properties of the original dataset.

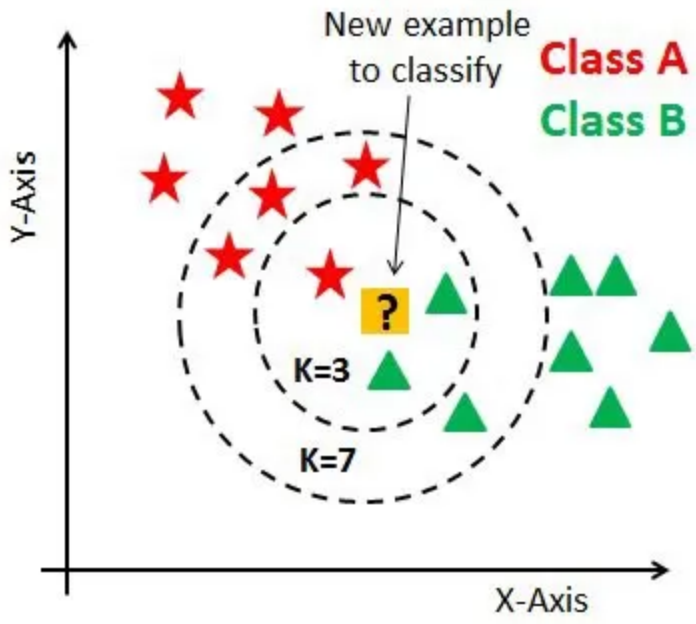

Figure 10. Illustration of K-Nearest Neighbor Algorithm [17].

Figure 11 provides a comprehensive, macroscopic visualization of this complete sequential workflow securely orchestrated within the underlying MASS Java application layer. The architectural pipeline initiates by ingesting the raw, partitioned graph datasets—including node identifiers and active training edges—directly into the *QueryGraphDB* application to establish the stationary spatial environment mapped across Steps 1 through 3. Following successful graph ingestion, the system transitions into the parallelized computing lifecycle, sequentially deploying the synchronized mobile agents to execute the distributed biased random walks and the localized Skip-Gram neural network optimizations in Step 4. Upon completion, the architecture extracts the fully converged global matrix and safely injects the learned continuous vectors directly back into the distributed memory properties of the underlying *PropertyVertexPlaces* in Step 5. With the entire graph structurally embedded, the application seamlessly invokes the decentralized KNN module located in Step 6, which rigorously

cross-references the embedded node proximity distances against a completely withheld testing dataset to probabilistically predict missing connections. Finally, the raw predictive connection pairs generated natively by the MASS agents are exported into an isolated algorithmic evaluation script, mathematically calculating the necessary metrics to scientifically benchmark the architecture against other algorithms.

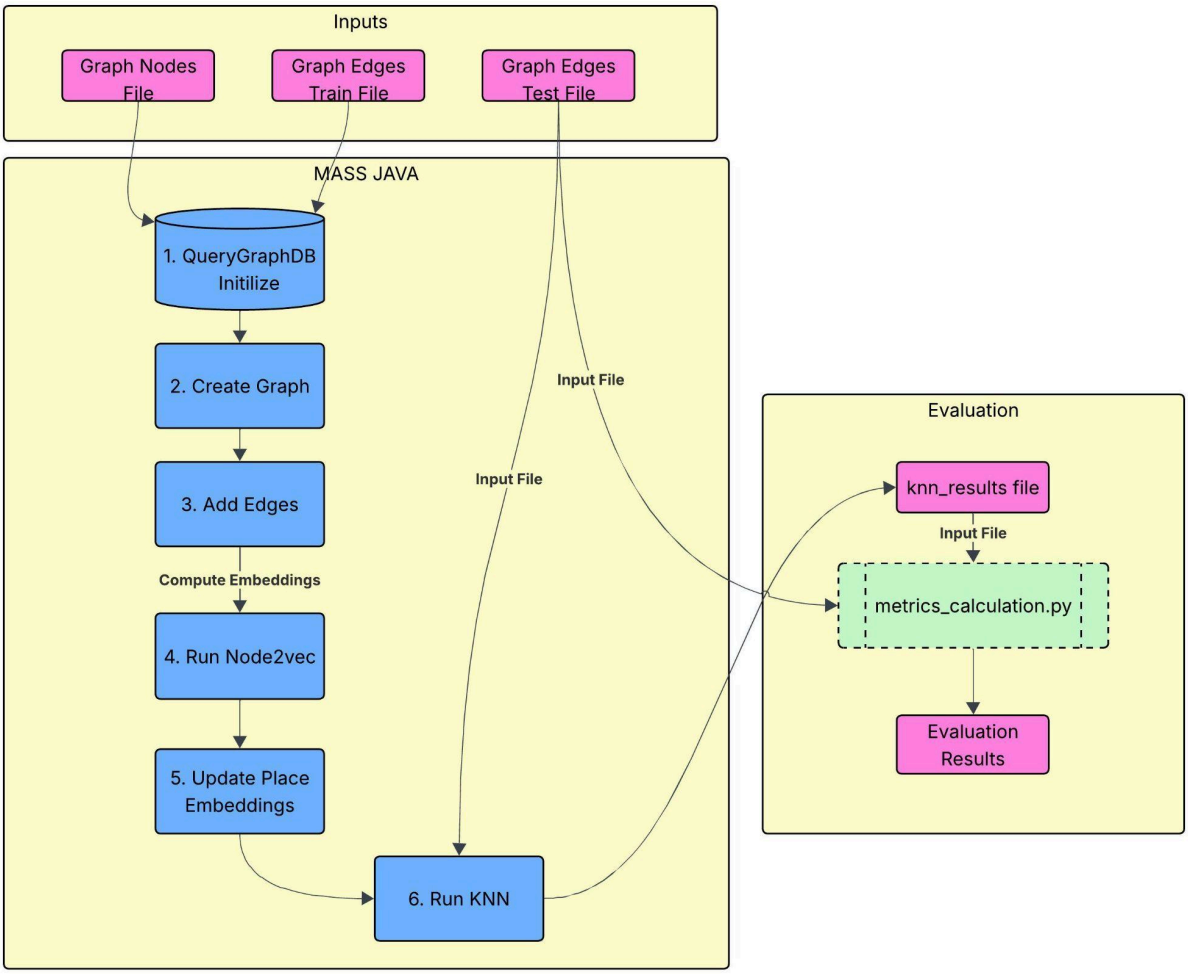

Figure 11. KNN Evaluation Workflow.

Chapter 4. Evaluation of Agent-based Approach

4.1 Measurement Environments

All empirical benchmarks for the MASS Node2Vec implementation and external control baselines were executed exclusively on the virtual machine clusters hosted by the Distributed Systems Laboratory [15]. The primary operating environment was the CSS Remote Hermes Cluster, which encompasses 24 virtual machines, with each node provisioned with 16GiB of RAM [15]. As a secondary execution environment, the project utilized the CSS Remote MPI Cluster, providing an identical scale of 24 virtual machines equipped with 16GiB of RAM [15]. To guarantee runtime software consistency across these distributed networks, every virtual machine was uniformly configured with the OpenJDK 17.0.17. Finally, to ensure accurate baseline comparisons against the underlying Java architecture, all standalone scripts and control algorithms were executed natively on these exact hardware provisions utilizing Python version 3.9.25.

Table 1. Comparison of dataset statistics

Property	**Cora**	**OGBL-DDI**
Nodes	2,708	4,267
Node labels	Paper	Drug
Total Edges	5,429	1,532,370
Train Edges	4343	1,225,896
Test Edges	1086	306474
Edge Types	CITES (directed)	TRAIN POS, VALID POS, VALID NEG, TEST POS, TEST NEG

To rigorously isolate and validate the predictive capabilities of the distributed Node2Vec architecture two distinct benchmark graphs were used: the Cora citation network and the Open Graph Benchmark Drug-Drug Interaction (OGBL-DDI) network, as summarized in Table 1 [12],

[18]. Cora is a small, well-known citation benchmark for reproducibility and qualitative comparison, while OGBL is a large, real-world graph that stresses scale and distribution—together they expose both correctness and scalability behaviors. The Cora dataset represents a comparatively sparse academic citation topology, consisting of 2,708 nodes designated as published papers that are structurally connected by 5,429 directed "CITES" edges [18]. Conversely, the OGBL-DDI dataset introduces a significantly denser, highly clustered evaluation environment, comprising 4,267 nodes designated as interacting drugs bridged by an overwhelming 1,532,370 edge associations [12]. Prior to executing any embeddings extraction, the structural edges within both datasets were algorithmically subjected to a standardized empirical allocation, randomly partitioning 80% of the network connections strictly for model training while securely withholding the remaining 20% exclusively for downstream prediction testing.

4.2 Evaluation Metrics

To rigorously validate the performance and accuracy of the distributed MASS Node2Vec implementation, the evaluation compares it against two established baseline algorithms. The first baseline is a standalone Python script utilizing the PyTorch Geometric (PyG) library, which executes an optimized, single-machine implementation of Node2Vec [13]. This PyG baseline serves as the primary control environment, allowing direct comparisons of execution time and performance metrics under identical algorithmic hyperparameter configurations. The second baseline is the Agent-based Fast Random Projection (FastRP) algorithm previously integrated into the MASS framework by Hotchandani [11]. Comparing MASS Node2Vec directly against MASS FastRP provides a clear evaluation of the computational scaling tradeoffs and topological accuracy differences within the exact same distributed spatial environment.

Table 2. Evaluation Metrics for KNN

Metric	Definition
Recall@k	$\frac{Number\ of\ relevant\ items\ in\ K}{Total\ number\ of\ relevant\ items}$
Precision@k	$\frac{Number\ of\ relevant\ items\ in\ K}{Total\ number\ of\ items\ in\ K}$
Hit Rate@k	$\frac{Number\ of\ queries\ with\ at\ least\ one\ correct\ link\ in\ top{-}k\ predictions}{Total\ number\ of\ queries}$
MAP (Mean Average Precision)	$\frac{1}{Q}\sum_{q=1}^{Q}(\frac{1}{N}\sum_{k=1}^{K} Precision(k)\ .\ rel(k))$, where $Q = Total\ Queries\ \&\ N = Total\ no\ of\ relevant\ items$
MRR (Mean Reciprocal Rank)	$\frac{1}{Q}\sum_{q=1}^{Q}\frac{1}{RR_q}$, where $RR = \frac{1}{Rank\ of\ the\ first\ relevant\ item}\ \&\ Q = Total\ Queries$
Training Time	Time taken for training
Walk Generation Time	Time taken to complete walks generation phase
Walks Per Node	Number of walks that need to be generated starting from each graph node

The predictive fidelity of the generated node embeddings is quantitatively assessed using five standard ranking metrics, as formulated in Table 2. Figure 12 illustrates the mathematical calculation of these retrieval metrics by evaluating specific query examples against a ranked list of relevant and non-relevant predictions [16]. As demonstrated in the figure's first query “Sweet Pastry”, Precision@k measures the exact fraction of the top-k predicted links that are actually relevant, while Recall@k calculates the fraction of all available relevant links successfully captured within that top-k suggestion window. Furthermore, the Hit Rate calculates whether at least one correct link appears anywhere within the top-k predictions, acting as a binary success indicator for targeted queries. To evaluate the priority and ordering of the predictions, the

framework utilizes the Mean Reciprocal Rank (MRR), which averages the reciprocal of the specific rank position where the first relevant link is initially identified. Finally, the Mean Average Precision (MAP) aggregates the precision scores across all relevant ranks to provide a comprehensive evaluation that penalizes algorithms for placing incorrect link predictions ahead of valid ones.

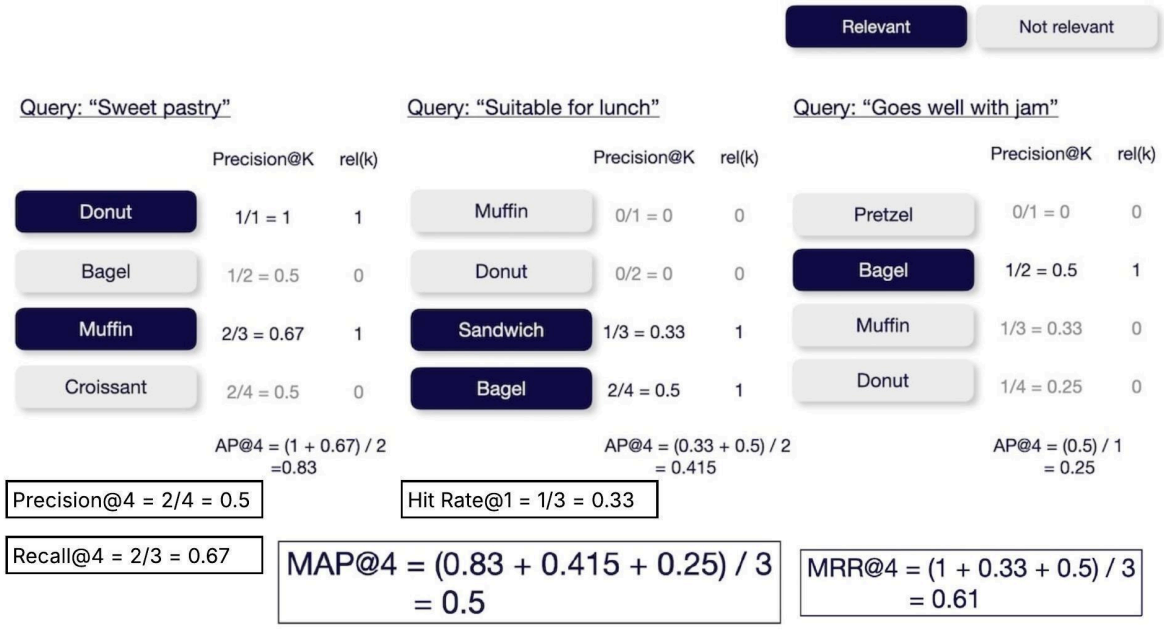

Figure 12. Illustration of Precision, Recall, Hit Rate, MAP & MRR [16].

4.3 Application-based Performance

In this section we evaluate the performance of MASS Node2Vec against baseline algorithms using established classification metrics, with all initial comparisons strictly executed on a single-node configuration. Furthermore, we expand this analysis to examine how the predictive accuracy of the MASS implementation behaves as the number of distributed computing nodes increases. When comparing our results to the standard Python implementation, readers should

anticipate differences in exact absolute values due to two significant architectural variations. First, Python relies on a degree-based unigram distribution [13] for negative sampling to heavily penalize highly connected "hub" nodes, whereas MASS currently employs uniform random sampling [2] where every node gets equal probability. Second, Python incorporates frequent node subsampling [13] to randomly ignore overrepresented nodes within context windows, a smoothing mechanism currently absent in our distributed approach. Consequently, while these algorithmic nuances introduce slight deviations in strict metric values, the overarching performance trends and relative scaling behaviors remain the primary focus of this evaluation.

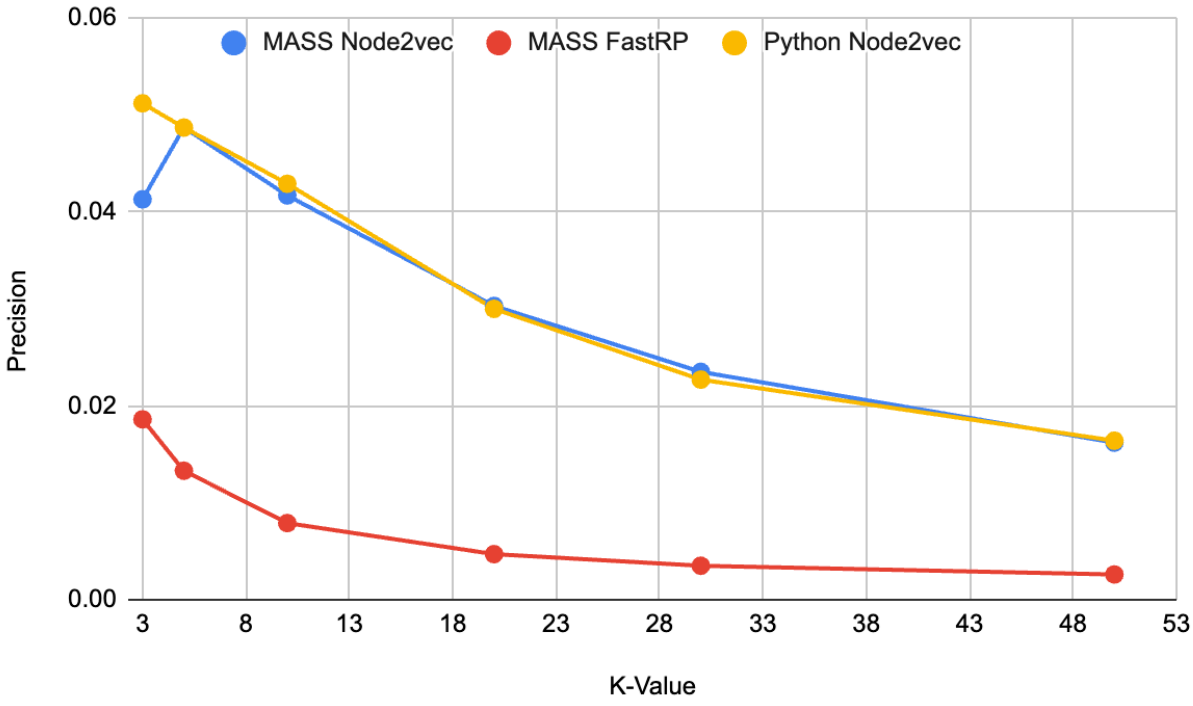

Figure 13. Cora - Precision@k by algorithm.

The initial phase of performance evaluation focused strictly on validating the predictive correctness of the distributed Node2Vec engine against the Cora citation dataset, measuring precision, recall, and hit rate across a scalable range of evaluation thresholds (K). As illustrated

in Figure 13, the Precision@K metrics for both MASS Node2Vec and the Python Node2Vec baselines predictably decay as the classification threshold expands from *K=3* to *K=50*, reflecting the inherent difficulty of maintaining strict accuracy when evaluating broader topological neighborhoods. Crucially, the MASS Node2Vec graph mirrors the highly precise trajectory of the Python implementation almost perfectly, aggressively outperforming the MASS FastRP baseline across the entire plotted spectrum. A sudden dip in precision is observable at *K=3* for the MASS Node2Vec implementation. Upon closer inspection of the resulting KNN predictions, it was observed that true positive neighbors were frequently displaced to rank 4 or 5 by slightly noisier nodes. This localized embedding jitter is likely a byproduct of the uniform negative sampling distribution and epoch-based learning rate decay discussed earlier, which occasionally fail to perfectly resolve micro-distances among the absolute closest neighbors. This superior topological learning is further reinforced by the Recall trajectory mapped in Figure 14. While the linear FastRP algorithm essentially flatlines, failing to capture relevant edges beyond a rigid 10% ceiling even at *K=50*, the distributed Node2Vec architecture demonstrates a steep, continuous geometric climb, successfully identifying over 55% of all relevant structural connections. Finally, the Hit Rate trajectories charted in Figure 15 validate the targeted effectiveness of the learned embeddings; the distributed MASS Node2Vec configuration guarantees that at least one correct structural connection is identified within the top 20 predictions for over 50% of the evaluated target nodes.

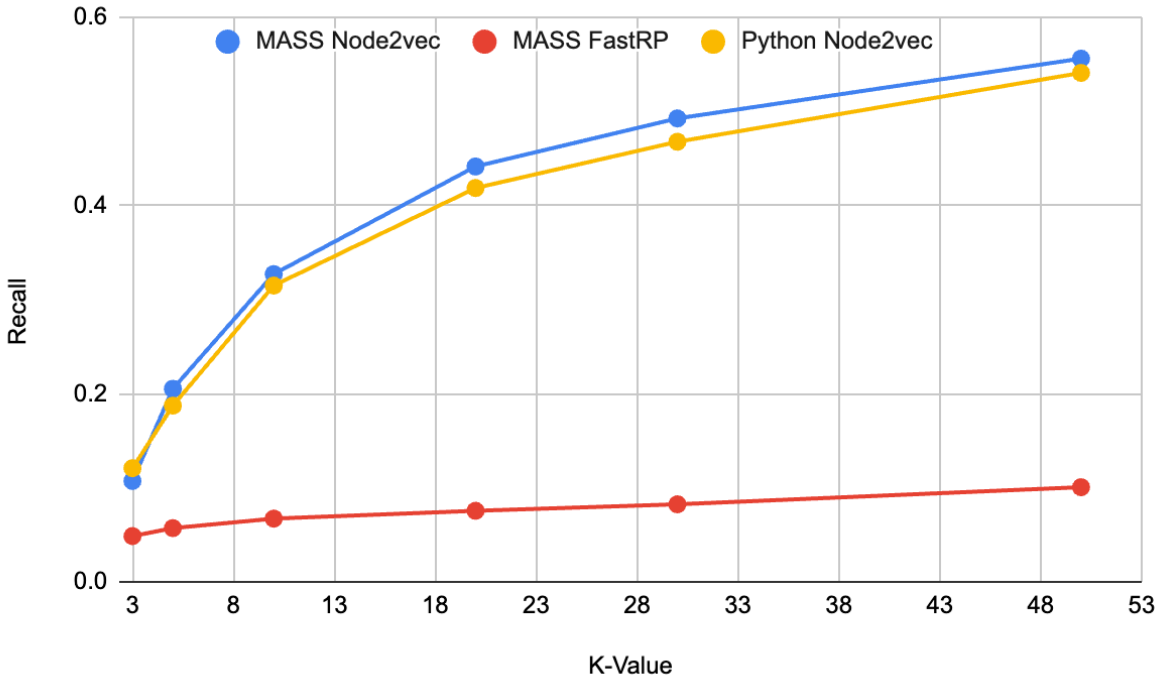

Figure 14. Cora - Recall@k by algorithm.

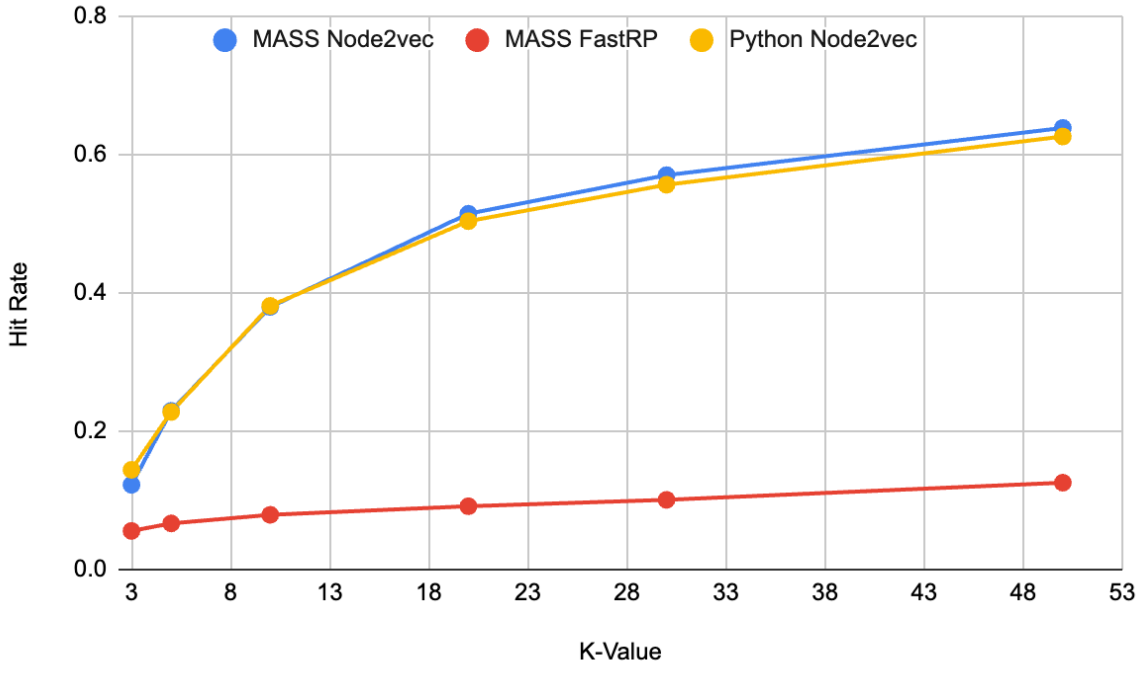

Figure 15. Cora - Hit Rate@k by algorithm.

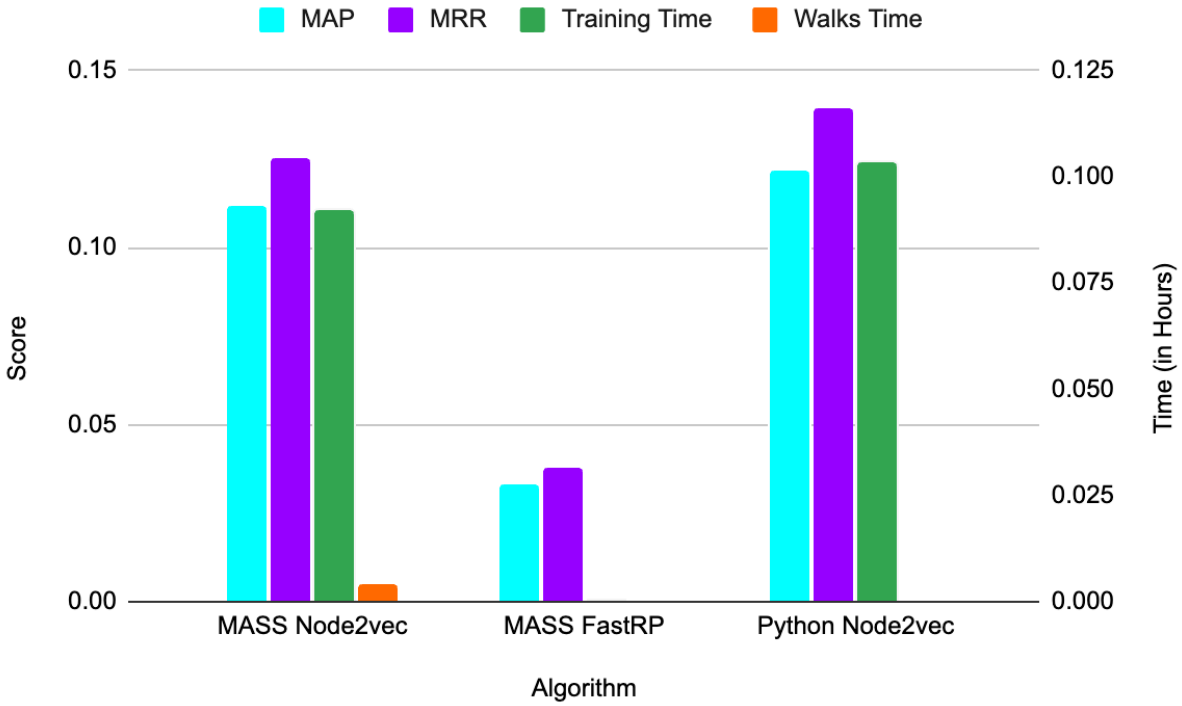

Figure 16. Cora - MAP, MRR & Exec Time by algorithm.

Expanding the evaluation to measure rigorous ranking quality and computational latency, Figure 16 compares the Mean Average Precision (MAP), Mean Reciprocal Rank (MRR), and execution times partitioned by algorithm. The chart clearly demonstrates that MASS Node2Vec closely matches the high-ranking fidelity of the single-machine Python baseline, generating MAP and MRR scores that vastly exceed the outputs achieved by MASS FastRP. However, evaluating the secondary y-axis reveals that both Node2Vec implementations incur a substantial computational penalty during the continuous neural network training phase, requiring significantly more execution time than the nearly instantaneous FastRP matrix operations. To specifically dissect the latency introduced by physical network distribution, Figure 17 tracks the predictive metrics alongside execution times utilizing a static threshold of $K=5$ as the MASS

Node2Vec cluster scales from 1 to 20 designated computing nodes. While the topological learning metrics (Precision, Recall, and Hit Rate) remain reassuringly stable regardless of the physical cluster size even doing slightly better on multi-node setups due to slight overfitting discussed in much detail later in this section, the execution times display a dramatic structural inflection. The neural network training phase predictably accelerates as it is parallelized across additional processing cores, visually shrinking on the chart; conversely, the walk generation time experiences a severe geometric surge, expanding substantially as the mobile agents are forced to serialize and migrate their state payloads across an increasingly fractured multi-node geographic JVM boundaries.

Figure 17. Cora - Precision, Recall, Hit Rate & Exec Time for MASS (*K*=*5*).

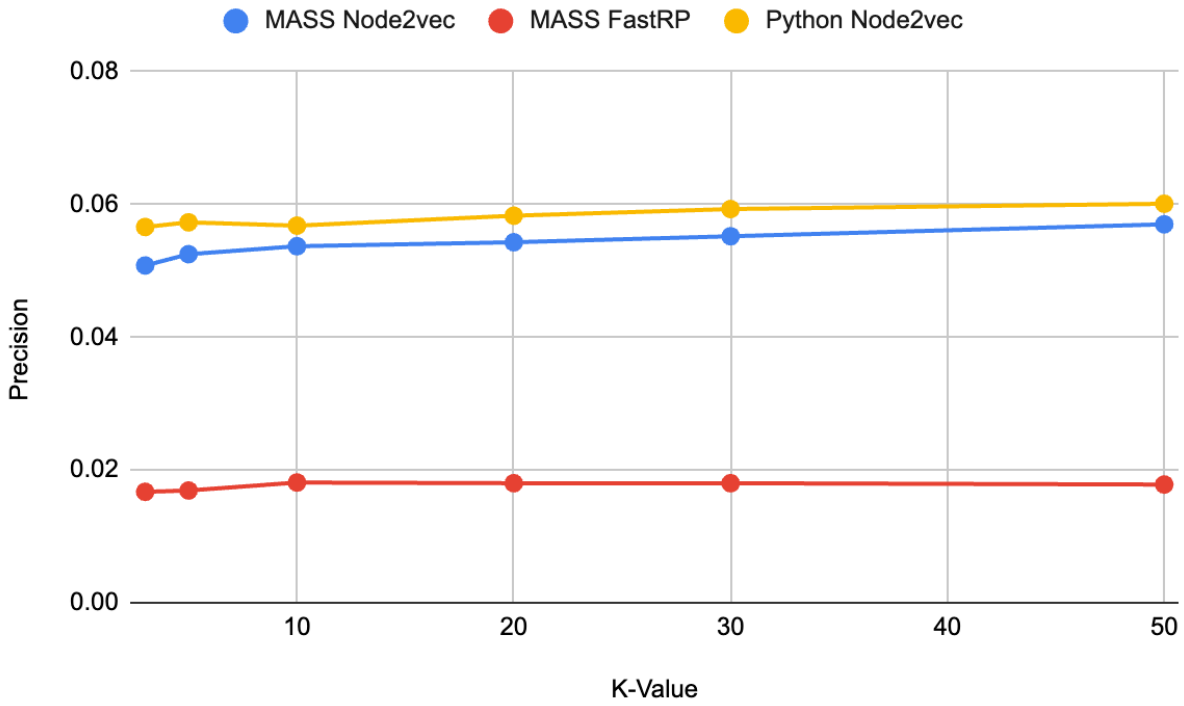

Figure 18. OGBL-DDI - Precision@k by algorithm.

To definitively validate the architectural stability and predictive fidelity against industrial-scale network densities, the evaluation transitioned to analyzing the massively clustered OGBL-DDI dataset. As illustrated in Figure 18, evaluating the Precision@K trajectories exposes a striking behavioral contrast to the sparser configurations observed earlier; rather than rapidly decaying, the precision metrics for both Node2Vec implementations remain remarkably elevated and stable as the topological evaluation threshold scales out toward K=50. Within this dense pharmaceutical network, the MASS Node2Vec architecture inherently maintains an aggressive predictive edge over MASS FastRP, successfully preserving parity with the Python control environment. Furthermore, Figure 19 outlines the Recall distributions for the algorithm, definitively exposing the devastating limitations of localized linear approximations. The matrix-based FastRP algorithm severely struggles to penetrate the massive web of

interconnected edges, producing a profoundly anemic recall progression that fails to break the 0.02 boundary. In contrast, the MASS Node2Vec vectors successfully untangle the non-linear dimensional clusters, demonstrating accelerating Recall velocity that ultimately outpaces even the Python configuration at expanding K-values. This is because Python Node2Vec utilizes a continuous linear learning rate decay to precisely fine-tune immediate neighbors, whereas MASS implements an epoch-based step decay that maintains larger gradient updates throughout each epoch. While these larger updates in MASS act as a regularizer that captures broader macro-neighborhoods and improves recall at expanded boundaries, this approach also increases the risk of the model overstepping the global optima and failing to perfectly converge at microscopic distances. This topological dominance is finally solidified in Figure 20, where the Hit Rate trajectory confirms that the distributed Skip-Gram embeddings guarantee valid target identification for nearly 80% of all evaluated pharmaceutical nodes at maximum thresholds, providing a massive structural advantage over the rigid approximations generated by FastRP.

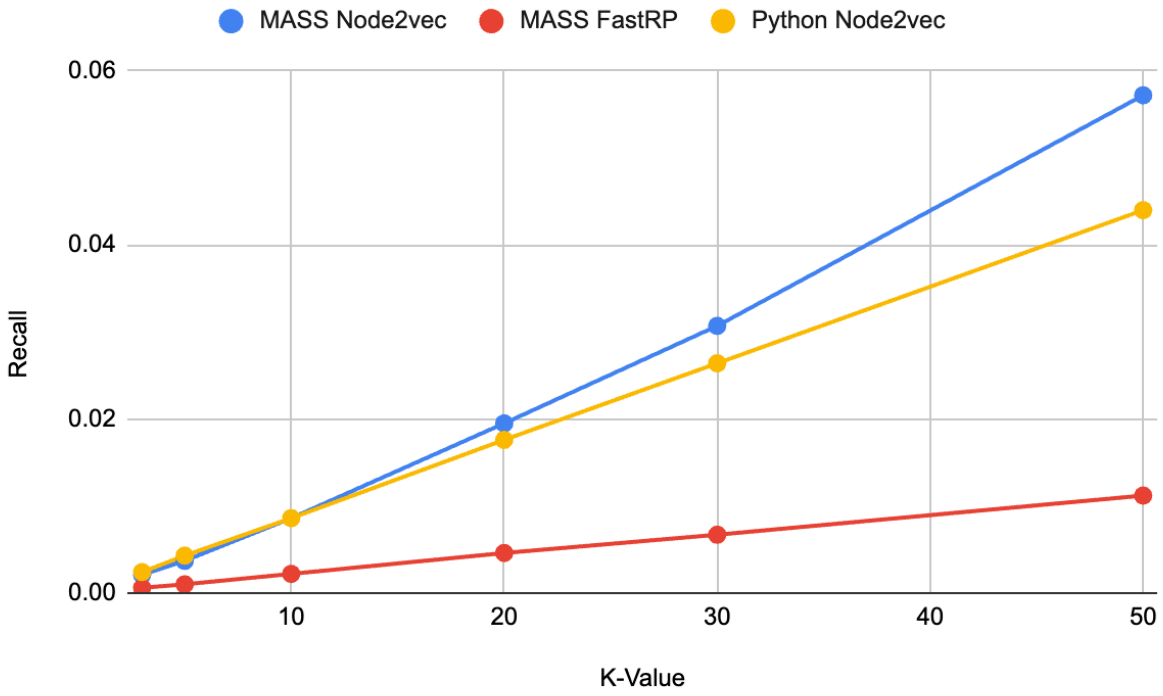

Figure 19. OGBL-DDI - Recall@k by algorithm.

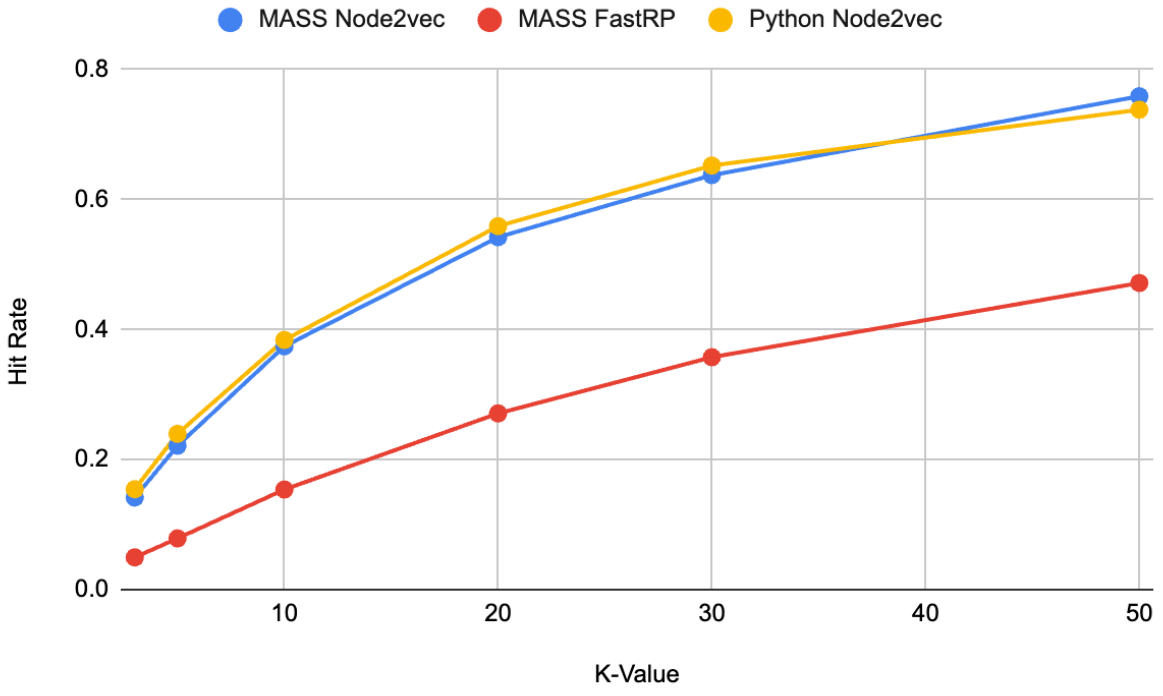

Figure 20. OGBL-DDI - Hit Rate@k by algorithm.

Expanding the evaluation of the OGBL-DDI dataset to map ranking quality and computational bottlenecks, Figure 21 correlates the Mean Average Precision (MAP) and Mean Reciprocal Rank (MRR) distributions against total execution latency. Conforming to the trends observed previously, the MAP and MRR scores achieved by the distributed MASS Node2Vec engine strongly align with the Python baseline and definitively outclass the outputs generated by MASS FastRP. Crucially, examining the secondary y-axis exposes a massive spike in the Walk Generation execution time specifically for MASS Node2Vec, severely eclipsing the identical phase observed on smaller datasets. Because the OGBL-DDI graph topological structure is exponentially denser, containing over 1.5 million relationships, the mobile agents are forced to load incredibly expansive *prevNeighborIds* arrays into their *WalkerArgs* memory constraints during every single traversal hop. Explicitly carrying this massive historical neighborhood list across physical JVM boundaries introduces a violent, compounded network serialization overhead, perfectly corroborating the architectural tradeoffs anticipated in Section 3.2. Finally, Figure 22 visualizes the structural and temporal dynamics of processing this dense workload at $K=5$ while expanding the distributed cluster from a singular node out to 20 nodes.

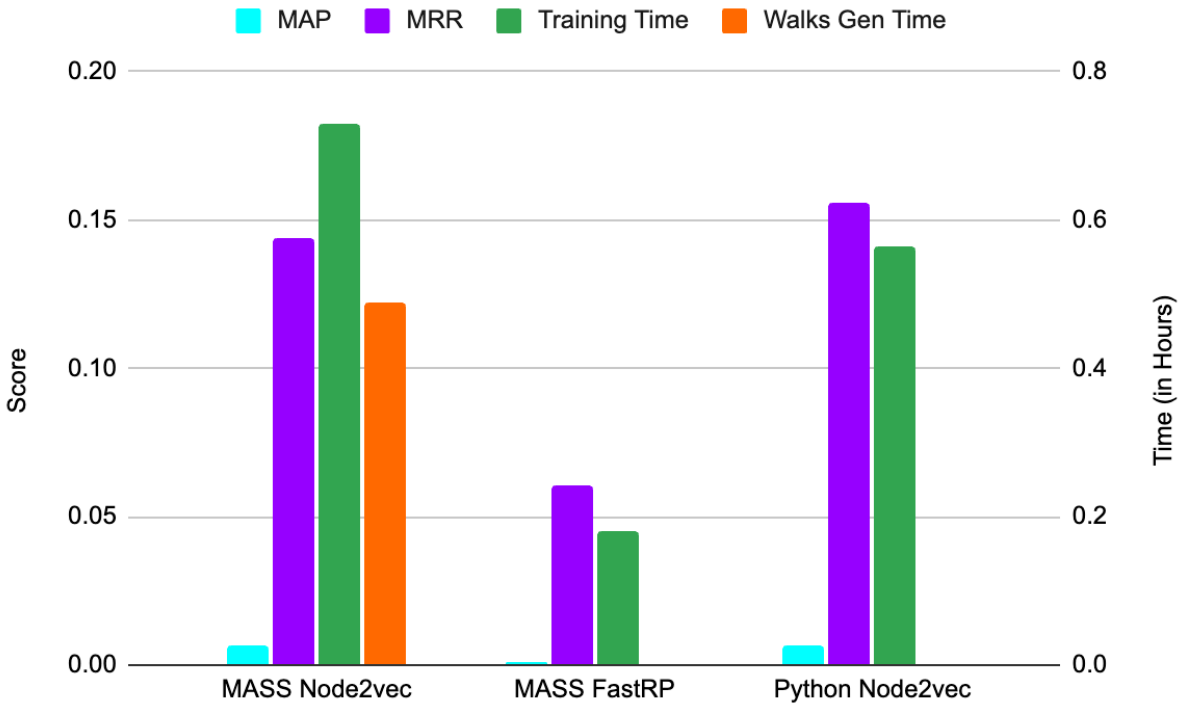

Figure 21. OGBL-DDI - MAP, MRR & Exec Time by algorithm.

Finally, Figure 22 visualizes the structural and temporal dynamics of computing this dense subgraph explicitly at *K=5* while expanding the distributed cluster from a singular node out to 20 nodes. Notably, the walk generation time experiences a severe initial spike when moving from 1 to 5 nodes due to the sudden architectural shock of serializing massive agent payloads across newly introduced network boundaries. However, this latency reliably recedes and stabilizes at 10 nodes and beyond, as the expanded cluster physical hardware provides sufficient CPU cores and network interfaces to efficiently absorb the distributed traffic. Furthermore, rather than remaining strictly flat, the predictive learning accuracy metrics exhibit a slight upward trajectory as the distributed cluster scales. Because the multi-node OGBL-DDI evaluation relies on a highly noisy, randomly sampled subgraph, aggregating a larger quantity of geographically isolated models during the reduceAll synchronization phase effectively functions

as an ensemble regularizer. This extensive distributed averaging mathematically smooths out the localized overfitting occurring on individual JVMs, inadvertently improving the global model's generalized predictive accuracy. Following the expected parallel scaling trajectory, the isolated neural network training time generally decreases as nodes scale outward, before experiencing slight latency rebounds near the maximum cluster scale as the efficiency of geometric reduction gradually diminishes against raw communication hurdles.

Figure 22. OGBL-DDI - Precision, Recall, Hit Rate & Exec Time for MASS (*K=5*).

It is critical to consider that for these specific multi-node executions, the evaluated OGBL-DDI graph was restricted to a randomly sampled subgraph comprising 106,791 training edges and 26,698 testing edges. This reduction was strictly necessary because the underlying MASS framework currently fails to reliably distribute and instantiate networks of extreme

density across multiple machines. As previously documented by Hotchandani [11], MASS initializes graphs sequentially by parsing edges on a single coordinator and dispatching targeted creation messages to remote nodes; for extraordinarily massive datasets, this sequential hand-off becomes an insurmountable bottleneck that currently renders full-scale multi-node graph ingestion impractical.

4.4 Scalability Performance

To explicitly validate the spatial scalability of the distributed architecture, benchmarking shifted toward maximizing the structural sampling capacity across expanding cluster footprints. Figure 23 illustrates the maximum permissible volume of walks per node the system can successfully simulate, mapped alongside the corresponding computational execution times for the Cora dataset. As the physical computing ecosystem expands from a singular machine out to twenty active processing nodes, the distributed framework accommodates a progressively higher threshold of generated walks per vertex. This upward scaling explicitly proves the architecture's spatial scalability, demonstrating that fragmenting the graph topology actively alleviates single-machine memory constraints to permit significantly denser topological sampling. Examining the corresponding temporal metrics reveals that while the walk generation time spikes immediately upon introducing the initial cross-JVM network communication boundary, it subsequently stabilizes even as the cluster scaling absorbs heavier sampling workloads. Conversely, the localized neural network training time exhibits a steady, compounding upward trajectory throughout the structural expansion. This continuous rise in training latency does not signify a failure in parallel processing efficiency; rather, it directly occurs because the deliberate

scaling of the *walks_per_node* parameter proportionately inflates the total aggregate data volume that the downstream Skip-Gram models must mathematically optimize.

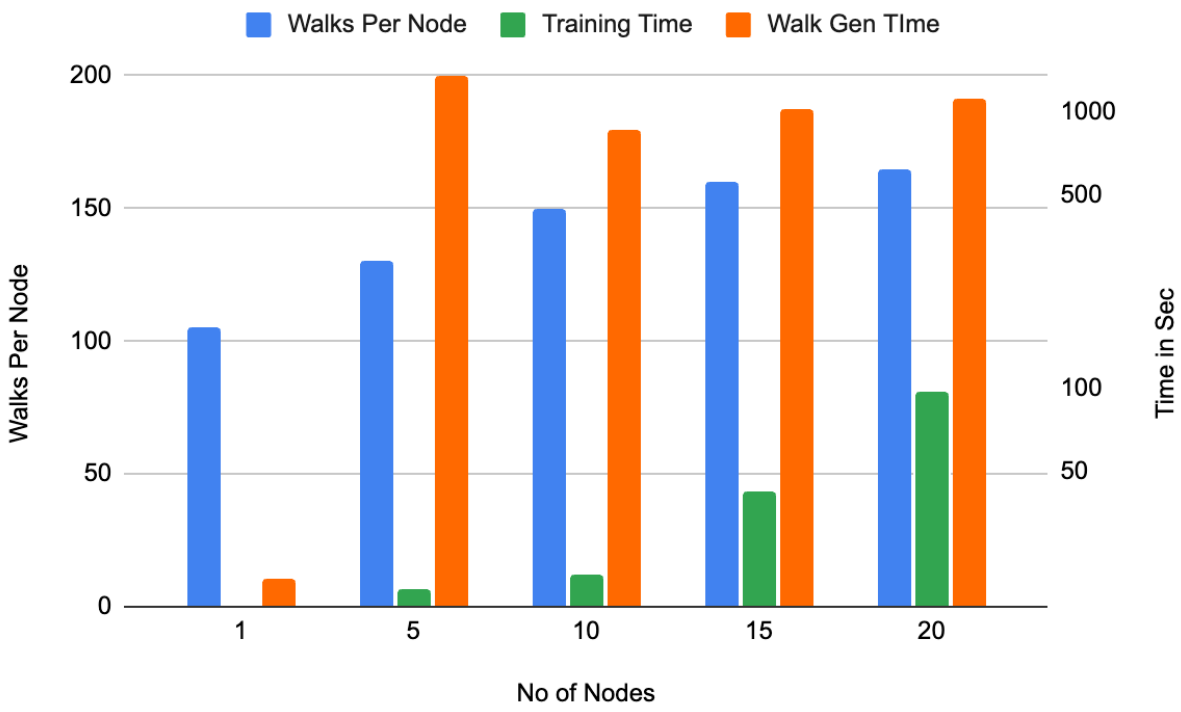

Figure 23. Cora - Embeddings Generation MASS Benchmarks**.**

To further substantiate this spatial scalability against significantly denser network topologies, Figure 24 maps the maximum embedding generation limits across the OGBL-DDI dataset. Mirroring the fundamental behavior observed in the Cora evaluation, distributing the network across 20 active computing nodes inherently provides the physical memory relief necessary to process a substantially higher volume of walks per node. The corresponding execution metrics demonstrate that walk generation time incurs a massive initial penalty when transitioning from a single JVM to a distributed cluster, reflecting the severe network serialization costs of propagating mobile agents across the highly connected pharmaceutical dataset. Similarly, the localized training time exhibits a steady geometric upward trajectory as the

distributed cluster expands, directly correlating with the deliberately inflated walk volumes that the independent Skip-Gram models must continuously optimize. It is imperative to reiterate that, precisely like the application benchmarks described in Section 4.3, this specific multi-node scalability evaluation was executed strictly against a randomly sampled subgraph composed of 106,791 training edges and 26,698 testing edges. This data restriction was maintained because the underlying MASS framework's sequential graph initialization bottleneck fundamentally prevents the reliable instantiation and geographic parsing of the complete, unpartitioned 1.5-million edge OGBL-DDI dataset across a distributed array.

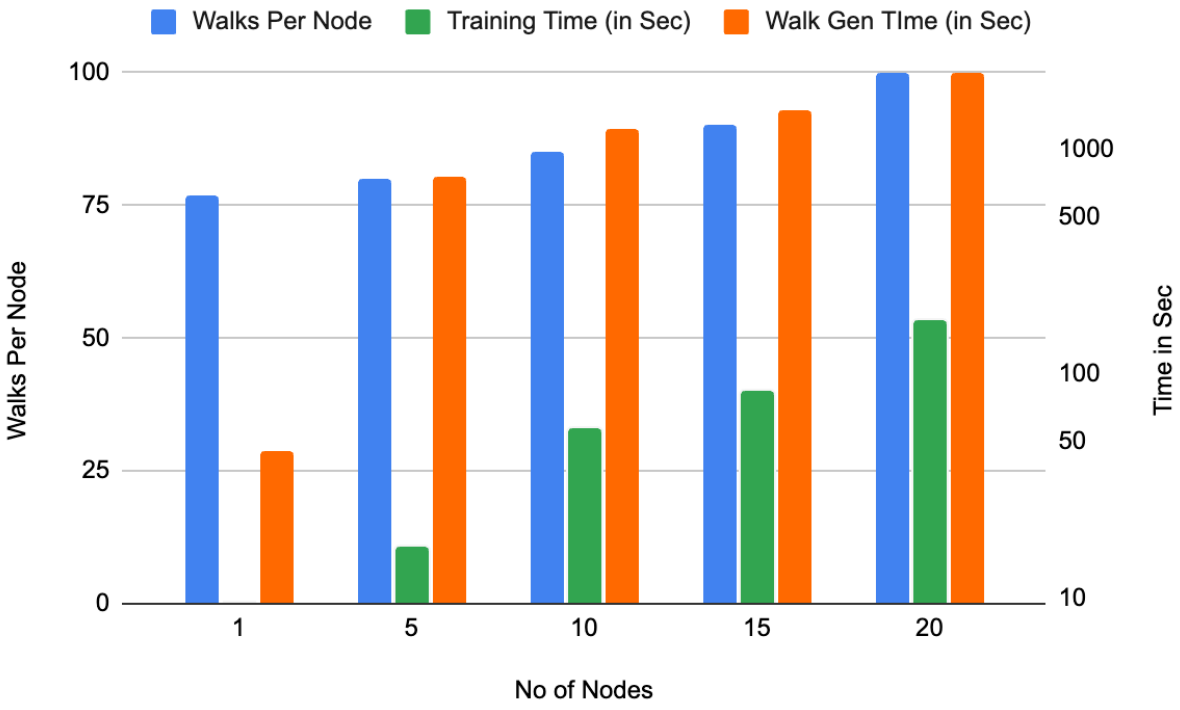

Figure 24. OGBL-DDI - Embeddings Generation MASS Benchmarks.

4.5 Discussion and Limitations

The empirical benchmarks comprehensively validate that the distributed MASS Node2Vec engine successfully generates high-fidelity topological embeddings, mostly matching the industry-standard PyTorch baseline in MAP, MRR, Precision, Recall and Hit Rate across both sparse and intensely dense networks. Furthermore, the architecture predictably outperforms the linear MASS FastRP algorithm, proving that simulating biased, second-order random walks is vastly superior at untangling complex localized clusters. Interestingly, the evaluation also revealed that aggregating numerous isolated models during the synchronization phase natively acts as an ensemble regularizer, mathematically smoothing out localized topological noise and occasionally boosting global predictive accuracy. In terms of cluster expansion, the framework demonstrated critical spatial scalability, explicitly allowing the network to simulate a significantly higher density of topological walks by distributing the traversal requirements across wider geographic hardware. However, this spatial decentralization ultimately incurs a massive network communication overhead. Because millions of autonomous software agents must continuously serialize their states and migrate across physical sockets simultaneously, the raw network bandwidth becomes heavily saturated. Consequently, the actual memory scalability of the system did not expand as efficiently as initially anticipated, as the immense communication latency required to orchestrate the distributed traversal heavily counteracted the computational benefits of the localized memory relief.

Despite these architectural successes, several strict operational capabilities presently limit the framework's absolute scalability. Foremost, generating true second-order random walks introduces a crushing serialization penalty, as every migrating software agent must explicitly carry a heavy array of its previous topological neighbors to maintain its historical path.

Furthermore, adopting the data-parallelism paradigm requires that every single active computing machine in the cluster possesses enough raw physical RAM to instantiate, hold, and independently train the entire localized Skip-Gram embedding model. From a mathematical synchronization perspective, while the proportional batching boundaries effectively align the disparate data sets, they theoretically cap the gradient descent update frequency to structural iteration boundaries rather than true, per-sample continuous optimization. Finally, the foundational MASS framework introduces a severe structural bottleneck regarding initial data ingestion. Because the underlying infrastructure sequentially constructs the distributed graph grid from a singular orchestrating coordinator, the system is fundamentally incapable of reliably parsing and loading unpartitioned, massive datasets—such as the complete OGBL-DDI network—across a multi-node cluster without encountering fatal crashes.

Chapter 5. Conclusion & Future work

This white paper detailed the architectural design, algorithmic implementation, and empirical evaluation of a distributed Node2Vec engine engineered natively within the Multi-Agent Spatial Simulation (MASS) framework. Driven by the core objective to overcome the crippling memory bottlenecks of centralized graph representation learning, the project successfully deployed a Compute-Node-Centric architecture that elegantly isolates spatial traversal from intense neural network training. By mapping biased random walks to mobile execution agents and utilizing a decentralized, logarithmic tree-reduction for Skip-Gram synchronization, the framework systematically bypassed traditional single-machine hardware limitations. The empirical evaluations comprehensively validated the system's algorithmic correctness, proving that this distributed engine achieves close predictive parity with highly optimized, single-node PyTorch baselines across critical ranking metrics—such as MAP, MRR, , Precision, Hit Rate and Recall in most cases—while decisively anticipating and outperforming linear matrix projections like FastRP. Crucially, the architecture successfully achieved its primary goal of spatial scalability, demonstrating the profound capacity to support increasingly massive densities of topological walks simply by expanding the geographic bounds of the hardware cluster. By scaling to 20 computing nodes, the system successfully handled 57% more walks per node on the Cora dataset and 43% more on the OGBL dataset compared to a single-node configuration, ultimately establishing a highly robust foundation for distributed deep graph learning.

Moving forward, several targeted architectural optimizations and algorithmic expansions are necessary to further elevate the framework's scalability and predictive capabilities. To directly mitigate the severe network serialization overhead observed during multi-node walk generation, future iterations should explore dynamic "triangle checks" utilizing localized

micro-agents, potentially eliminating the requirement for primary walkers to carry massive, serialized arrays of their previous neighbors. Additionally, the distributed Skip-Gram synchronization efficiency could be drastically improved by transmitting only the specifically updated gradient deltas across the network rather than exchanging and reducing the entire comprehensive embedding matrix every epoch. To systematically enhance the neural network's geometric generalization, the architecture would also benefit from an intermediate data-shuffling phase, redistributing the harvested walk sequences amongst different computing nodes prior to each training epoch. Furthermore, Fast-Node2Vec [21]—an efficient variant of Node2Vec—can be implemented to further optimize memory efficiency. At the foundational infrastructure level, engineering a parallel data ingestion protocol is required to resolve the MASS framework's sequential graph creation bottleneck; while this initialization limitation does not explicitly restrict the computational capabilities of the Node2Vec engine, resolving it is strictly necessary to enable future benchmarking against complete, unpartitioned industrial datasets like the full OGBL-DDI network. Node2Vec is well suited for capturing graph structural/relational patterns (community, role, link prediction) but less ideal for encoding explicit relational semantics—knowledge-graph models like TransE[19] can be implemented which is preferable when relation semantics are primary. Finally, the successful validation of this decentralized engine establishes the operational groundwork to integrate more advanced, inductive representation learning models, such as GraphSAGE, effectively expanding the MASS ecosystem to support a continuously widening array of complex downstream machine learning applications.

Bibliography

[1] William L. Hamilton, Rex Ying, and Jure Leskovec. “Representation Learning on Graphs: Methods and Applications”. Bulletin of the IEEE Computer Society Technical Committee on Data Engineering. 2017. url: https://www-cs.stanford.edu/people/jure/pubs/graphrepresentation-ieee17.pdf (visited on 04/16/2026).

[2] A. Grover and J. Leskovec. “node2vec: Scalable Feature Learning for Networks”. In: Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining. ACM, Aug. 2016, pp. 855–864. url: https://snap.stanford.edu/node2vec/ (visited on 04/26/2026).

[3] Spot Intelligence. “Node2Vec: Biased Random Walks for Node Embeddings”. url: https://spotintelligence.com/2024/01/18/node2vec/ (visited on 04/26/2026).

[4] Wikipedia contributors. “Word2vec”. Wikipedia, The Free Encyclopedia. url: https://en.wikipedia.org/wiki/Word2vec (visited on 04/26/2026).

[5] Spot Intelligence. “Skip-gram Models Explained & How To Create Embeddings in Word2Vec”. July 2023. url: https://spotintelligence.com/2023/07/11/skip-gram-models-explained-how-to-create-embeddings-in-word2vec/ (visited on 04/26/2026).

[6] Wikipedia contributors. “Stochastic gradient descent”. Wikipedia, The Free Encyclopedia. url: https://en.wikipedia.org/wiki/Stochastic_gradient_descent (visited on 04/16/2026).

[7] Microsoft. “Distributed training - Azure Machine Learning”. en. url: https://learn.microsoft.com/en-us/azure/machine-learning/concept-distributed-training?view=azureml-api-2 (visited on 04/16/2026).

[8] Đồng Nguyễn Minh ANH. “Megatron LM — How Model Parallelism Is Pushing Language Models to New Heights !!”. Medium. Mar. 2023. url: https://medium.com/@minhanh.dongnguyen/megatron-lm-how-model-parallelism-is-pushing-language-models-to-new-heights-c21a5343e06a (visited on 04/16/2026).

[9] Vishakh Hegde and Sheema Usmani. “Parallel and Distributed Deep Learning”. Stanford University. May 2016. url: https://web.stanford.edu/~rezab/classes/cme323/S16/projects_reports/hedge_usmani.pdf (visited on 04/16/2026).

[10] University of Washington Distributed Systems Lab. “Multi-Agent Spatial Simulation (MASS) Framework”. url: https://depts.washington.edu/dslab/MASS/ (visited on 04/26/2026).

[11] S. Hotchandani. “Agent-based Link Prediction in Graph Databases using FastRP”. CSS 595 Capstone Project Report. University of Washington, Distributed Systems Laboratory, 2025. url: https://depts.washington.edu/dslab/MASS/reports/SumitHotchandani_wi25.pdf (visited on 04/26/2026).

[12] Neo4j. “Common Datasets”. Neo4j Graph Data Science Client Documentation, version 1.14. url: https://neo4j.com/docs/graph-data-science-client/1.14/common-datasets/ (visited on 04/26/2026).

[13] PyG Team. “torch_geometric.nn.models.Node2Vec”. PyTorch Geometric documentation, version 2.7.0. url: https://pytorch-geometric.readthedocs.io/en/2.7.0/generated/torch_geometric.nn.models.Node2Vec.html (visited on 04/16/2026).

[14] Guy Blelloch and Umut Acar. “Divide-and-Conquer and Reduce”. 15-210: Parallel and Sequential Data Structures and Algorithms. Carnegie Mellon University. Jan. 2023. url: https://www.cs.cmu.edu/~15210/notes/divide-and-conquer.pdf (visited on 04/16/2026).

[15] University of Washington Bothell CSS Wiki. “CSS Computing Resources”. url: https://csswiki.uwb.edu/css-computing-resources/ (visited on 04/16/2026).

[16] Leonie Monigatti. “Evaluation Metrics for Search and Recommendation Systems”. Weaviate Blog. May 2024. url: https://weaviate.io/blog/retrieval-evaluation-metrics (visited on 04/16/2026).

[17] Antony Christopher. “K-Nearest Neighbor. A complete explanation of K-NN”. The Startup (Medium). Feb. 2021. url: https://medium.com/swlh/k-nearest-neighbor-ca2593d7a3c4 (visited on 04/16/2026).

[18] Orbifold Consulting. “The Cora Dataset”. Graph Consulting. 2025. url: https://graphsandnetworks.com/the-cora-dataset/ (visited on 04/16/2026).

[19] Antoine Bordes, Nicolas Usunier, Alberto Garcia-Durán, Jason Weston, and Oksana Yakhnenko. “Translating Embeddings for Modeling Multi-relational Data”. In: Advances in Neural Information Processing Systems (NIPS 26). 2013. url: https://proceedings.neurips.cc/paper_files/paper/2013/file/1cecc7a77928ca8133fa24680a88d2f9-Paper.pdf (visited on 06/01/2026).

[20] Apache Software Foundation. “Apache Spark™ - Unified Engine for large-scale data analytics”. url: https://spark.apache.org/ (visited on 06/01/2026).

[21] Dongyan Zhou, Songjie Niu, and Shimin Chen. “Efficient Graph Computation for Node2Vec”. arXiv:1805.00280 [cs.DC]. May 2018. url: https://arxiv.org/abs/1805.00280 (visited on 06/01/2026).

Appendix A: Benchmarks

Table 3. Cora MASS multi-node embedding generation benchmarks

No of Nodes	Walks Per Node	Training Time (in Sec)	Walk Gen TIme (in Sec)
1	105	16.302	20.529
5	130	18.766	1352.622
10	150	21.355	865.352
15	160	42.243	1024.485
20	165	97.267	1116.57

Table 4. OGBL-DDI MASS multi-node embedding generation benchmarks

No of Nodes	Walks Per Node	Training Time (in Sec)	Walk Gen TIme (in Sec)
1	77	9.44	45.245
5	80	17.026	758.334
10	85	57.133	1243.989
15	90	83.722	1479.978
20	100	172.889	2194.408

Table 5. Cora Recall@k, Precision@k, Hit Rate@k MASS Node2vec benchmarks

K-Value	**Precision**	**Recall**	**Hit Rate**
3	0.0413	0.1072	0.1225
5	0.0487	0.2052	0.2295
10	0.0417	0.3272	0.3798
20	0.0303	0.4416	0.5147
30	0.0235	0.4927	0.5705
50	0.0162	0.5563	0.6388

Table 6. Cora Recall@k, Precision@k, Hit Rate@k MASS FastRP benchmarks

K-Value	Precision	Recall	Hit Rate
3	0.0186	0.0486	0.0558
5	0.0133	0.0571	0.0667
10	0.0079	0.0672	0.0791
20	0.0047	0.0755	0.0915
30	0.0035	0.0824	0.1008
50	0.0026	0.1006	0.1256

Table 7. Cora Recall@k, Precision@k, Hit Rate@k Python Node2vec benchmarks

K-Value	Precision	Recall	Hit Rate
3	0.0512	0.1208	0.1442
5	0.0487	0.1873	0.2279
10	0.0429	0.315	0.3814
20	0.03	0.4187	0.5039
30	0.0227	0.4679	0.5566
50	0.0164	0.5411	0.6264

Table 8. Cora MAP, MRR & Exec Time by algorithm

Algorithm	MAP	MRR	Walk Gen Time (in Sec)	Training Time (in Sec)
MASS Node2vec	0.1119	0.1253	14.263	331.942
MASS FastRP	0.0330	0.0380	N/A	1.366
Python Node2vec	0.1217	0.1394	0.01	371.5438

Table 9. Cora Precision, Recall, Hit Rate and Exec Time on MASS

No of Nodes	MASS Node2vec (k=5)				
	Precision	**Recall**	**Hit Rate**	**Training Time (in sec)**	**Walk Gen Time (in sec)**
1	0.045	0.1877	0.2124	342.201	17.243
5	0.0521	0.2171	0.2481	81.867	1355.316
10	0.0502	0.2085	0.2372	129.038	2305.644
15	0.0509	0.208	0.2403	146.411	2448.21
20	0.0527	0.2142	0.2481	183.248	2810.373

Table 10. OGBL-DDI Recall@k, Precision@k, Hit Rate@k MASS Node2vec benchmarks

K-Value	Precision	Recall	Hit Rate
3	0.0508	0.0021	0.1416
5	0.0525	0.0037	0.2213
10	0.0537	0.0086	0.3742
20	0.0543	0.0195	0.5421
30	0.0552	0.0307	0.6375
50	0.057	0.0572	0.7589

Table 11. OGBL-DDI Recall@k, Precision@k, Hit Rate@k MASS FastRP benchmarks

K-Value	Precision	Recall	Hit Rate
3	0.0167	0.0006	0.0495
5	0.0169	0.001	0.0787
10	0.0181	0.0022	0.1539
20	0.018	0.0046	0.271
30	0.018	0.0067	0.3575
50	0.0178	0.0112	0.4716

Table 12. OGBL-DDI Recall@k, Precision@k, Hit Rate@k Python Node2vec benchmarks

K-Value	Precision	Recall	Hit Rate
3	0.0566	0.0024	0.1544
5	0.0573	0.0043	0.2396
10	0.0568	0.0086	0.384
20	0.0583	0.0176	0.5591
30	0.0593	0.0264	0.6523
50	0.0601	0.044	0.7382

Table 13. OGBL-DDI MAP, MRR & Exec Time by algorithm

Algorithm	MAP	MRR	Walk Gen Time (in Sec)	Training Time (in Sec)
MASS Node2vec	0.0066	0.1442	1760	2624
MASS FastRP	0.0011	0.0608	N/A	646.309
Python Node2vec	0.0066	0.1561	0.1	2028.9094

Table 14. OGBL-DDI Precision, Recall, Hit Rate and Exec Time on MASS

	MASS Node2vec (k=5)				
No of Nodes	**Precision**	**Recall**	**Hit Rate**	**Training Time (in sec)**	**Walk Gen Time (in sec)**
1	0.0045	0.0027	0.0225	213.216	45.464
5	0.0033	0.0036	0.0165	58.855	678.807
10	0.0051	0.0035	0.0244	46.861	386.871
15	0.0074	0.0052	0.0365	48.868	359.438
20	0.0061	0.0049	0.0298	74.394	426.473

Appendix B: Documentation And Sample Execution Guide

B.1 Documentation

Repository: mass_java_core, gdeepak/node2vec

URL: https://bitbucket.org/mass_library_developers/mass_java_core/branch/gdeepak/node2vec

Import Path: edu.uw.bothell.css.dsl.MASS.graph.db.ml.embeddings.node2vec.Node2Vec

Initialization Parameters:

1. *Graph Places:* The primary PropertyGraphPlaces instance containing the distributed vertex and edge data.
2. *Training Places:* A secondary PropertyGraphPlaces instance deployed with one place per computing node, strictly utilized for aggregating and reducing local machine weights during the distributed model training phase.
3. *Dimensions:* The size of the output embedding vectors.
4. *Walks Per Node:* The total number of spatial random walks initiated from each vertex.
5. *Walk Length:* The maximum number of hops a walker agent can take during a single random walk.
6. *Context Size:* The symmetric Skip-Gram window size used for generating positive target-context pairs.
7. *Return Parameter (p):* Controls the likelihood of a walker returning to the immediate previous node.
8. *In-Out Parameter (q):* Controls the likelihood of a walker exploring outward versus staying local.
9. *Epochs:* The total number of distributed training passes over the generated walk corpus.

Configuration Methods: Before initiating the training, the algorithm behavior can be optionally tuned using several configuration setters

1. *setWeightReductionFrequency(double n):* Configures the synchronization frequency of the neural network weights across the cluster. A value of 1.0 synchronizes after every walk iteration, whereas 0.0 minimizes synchronization to only occur after all walk batches are processed within an epoch.
2. *setNegativeSampling(int k):* Sets the number of negative samples drawn to update the model against each true context pair, accelerating the softmax computation.
3. *setLearningRate(double learning_rate):* Defines the initial step size for the gradient descent updates.
4. *setDecayRate(double decay):* Sets the multiplicative decay factor applied to the learning rate at the conclusion of each epoch.
5. *setIncludeEdgeTypes(String... types):* Restricts the walker agents to solely traverse edges matching the specified string descriptors, allowing for metapath-based training on heterogeneous graphs.

Execution:

1. *learnFeatures():* Executing this phase triggers the entire distributed pipeline. It initiates agent-based random walk generation, constructs a consistent clustered vocabulary, executes distributed batches of Skip-Gram training, performs global tree reductions for loss evaluation, and finally distributes the completed embeddings back to every Graph Place accessible at PropertyVertexPlace.embedding.

B.2 Sample Execution

For Sample execution, we use the MASS Java application repository codebase, which contains code used to evaluate this paper and benchmarks. Please follow below steps in order:

1. Download MASS Java core codebase from : https://bitbucket.org/mass_library_developers/mass_java_core/branch/gdeepak/node2vec
2. Download MASS Java application codebase from : https://bitbucket.org/mass_application_developers/mass_java_appl/branch/gdeepak/node2vec
3. Install MASS Java Core:
    a. Execute: *cd ~/mass_java_core/*
    b. Execute: *mvn clean package install*
4. Package MASS Java Application:

    Important: After rebuilding mass java core, update the mass.version tag in the pom.xml of mass java appl to match the new version.

    a. Execute: *cd ~/mass_java_appl/QueryGraphDB*
    b. Execute: *mvn package*
5. Finally run build_run_node2vec.sh script to run sample execution with preset values. Command: "*sh ./build_run_node2vec.sh*".
6. This will execute entire pipeline and generate KNN results file at *target/classes/knn_analysis_results_test_node2vec.csv.*