ScaleRunner: A Fast MPI-Based Random Walk Engine for Multi-CPU Systems

Florian Willich[1] and Henning Meyerhenke[1,2(✉)]

[1] Humboldt-Universität zu Berlin,
Berlin, Germany
f.willich@hu-berlin.de
[2] Karlsruhe Institute of Technology (KIT),
Karlsruhe, Germany
meyerhenke@kit.edu

Abstract. Random walks (RWs) on graphs have a plethora of applications, both in theory and practice. One of the currently most important applications is representation learning (RL) – finding a suitable embedding of a graph into some low-dimensional geometric space. The demand for fast RW algorithms lead to a variety of RW engines targeting different computing architectures. In this paper, we address multi-CPU systems and aim at improving upon existing random walk engines such as KnightKing when running first- and second-order RW algorithms. To this end, we introduce ScaleRunner, a C++ library with full CMake integration that executes random walks in parallel.

Our main acceleration techniques for ScaleRunner are: (i) each random walk is modeled as a task deployed to a thread-pool, balancing the work load on each CPU separately; (ii) integration of the dynamic graph data structure DHB to speed up graph data caching operations; (iii) collective MPI I/O routines to speed up graph input, path output, and postprocessing operations. Our experiments use a variety of popular benchmark graphs to execute RW algorithms commonly used in RL applications. On average, ScaleRunner speeds up first-order RWs by one order of magnitude and second-order RWs by two orders compared to KnightKing.

Keywords: Distributed Random Walks · Parallel Graph Computations · Rejection Sampling · MPI Collective I/O

1 Introduction

A random walk (RW) on a graph $G = (V, E)$ is a stochastic process for which the random variable X_i denotes the vertex $v \in V$ visited in iteration$i$. In i ts simplest form, a random walk chooses the next vertex X_{i+1} uniformly at random from the neighbors of$v$, but many variations exist. While in first-order RWs the transition probabilities between neighbors are fixed, these probabilities depend in second-order random walks on X_{i-1}, i. e., the vertex that was visited before the current one X_i. RWs on graphs have a plethora of usages and applications,

both in theory [13] and in practice [23]. The RW-based measure *commute time*, for example, is closely related to *effective resistances* in electrical networks [13]. Both form a metric on a graph, used for example for graph clustering [26].

One of the currently most popular applications of RWs is representation learning (RL), whose goal (in our context) is to embed a graph into a lower-dimensional geometric space. The resulting embedding shall facilitate a correct data classification while minimizing the amount of data points required for feature representation. Several successful RL techniques for graphs are based on random walks. For example, the algorithms *node2vec* [10] and *deepwalk* [17] use the co-occurrence of graph vertices in short random walks to determine their proximity in the embedding; these two algorithms and follow-up variations have been shown to yield high-quality graph embeddings [3].

To obtain high-quality embeddings, sufficiently many random walks are needed. For graphs of non-trivial size, this becomes compute-intensive. Since each RW is independent, executing them in parallel seems obvious and beneficial. It thus does not come as a surprise that RW engines have been written for a wide range of parallel architectures, with some of the most common ones being multi-core SINGLE-CPU [12,19,21,24], MULTI-CPU [25], and GPU [14,20]. Each architecture has its advantages and disadvantages. While GPUs offer a particularly good speed-energy tradeoff, they are limited in terms of memory. For massive graphs it is thus necessary to employ multi-core machines with large shared memory or, better yet in terms of scalability and therefore parallel performance, MULTI-CPU systems with distributed memory. The conceptual starting point for our work was KNIGHTKING [25], a widely used RW engine and a popular choice for MULTI-CPU systems. While it offers some required functionality, we spotted room for improvement in terms of usability and performance: (i) as KNIGHTKING is built as a standalone application, it cannot be integrated easily into other applications or computational pipelines; (ii) its I/O routines do not rely upon MPI; (iii) the granularity of the thread parallelism employed may often lead to load imbalances.

Contributions. In this paper, we introduce SCALERUNNER, a modular C++ library with full CMake integration that uses hybrid parallelism (MPI+OpenMP) to execute first- and second-order RWs on MULTI-CPU systems. Its open-source code and a demo application showcasing how to use SCALERUNNER can be found on GitHub[1]. Our neighborhood sampling methods provide functionality for both unweighted and weighted graphs. As part of SCALERUNNER, we also contribute the I/O layer GDSB. It reads in graphs in parallel (and does so 2.5× faster on average than highly optimized text-based graph POSIX I/O), stores them in the dynamic graph data structure DHB [9], and writes path data to disk. DHB, in turn, is chosen because it works particularly well for graphs with skewed degree distributions. In our systematic experiments on 17 popular benchmark graphs, we compare SCALERUNNER with the MPI-based RW engine KNIGHTKING. Our results show that SCALERUNNER

[1] https://github.com/hu-macsy/ScaleRunner.

is on average significantly faster than KnightKing. When taking the geometric mean of all algorithmic speedups vs KnightKing, ScaleRunner's throughput is 16.7× higher for first-order RWs and 112.0× to 124.7× higher for different second-order RWs.

2 Preliminaries and Related Work

We consider simple directed graphs $G = (V, E)$ with $n := |V|$ vertices, $m := |E|$ edges, and density $\rho(G) := \frac{m}{n(n-1)}$. A weighted graph $G = (V, E, \omega)$ additionally carries a weight function $\omega : E \to \mathbb{R}$. The outgoing neighborhood of a vertex u is defined as $N^+(u) := \{v \in V \mid (u, v) \in E\}$; its cardinality is called the (outgoing) degree of u: $\deg^+(u) := |N^+(u)|$. We denote the maximum (outgoing) degree of graph G as $\deg^+_{\max}(G)$ and the average as $\overline{\deg^+}(G)$. If a graph is unweighted but a weight is required, we assign the weight function $\omega : E \to \{1\}$.

2.1 Random Walks

A random walk on a graph G begins on some start vertex $v_0 \in V$ and then visits vertices of G iteratively in a random manner, thereby creating a walk $\chi = (v_0, v_1, \ldots, v_L) \in V$ with L steps [13]. When visiting $u \in V$, the next vertex in the walk is chosen based on a probability distribution. In so-called *first-order* random walks, this distribution only depends on u, the current vertex. One often uses a transition probability matrix P to store the distribution: entry $p_{u,v}$ denotes the probability of traversing the edge (u, v) from vertex u to vertex v. In *second-order* random walks, the probability to move from the current vertex u to the next one not only depends on u, but also on its predecessor in the walk.

Random walks are very useful in both theory and in practice. (Mainly) Theoretical works use them for example in spectral graph theory [5], exploit them for provably efficient solvers of certain linear systems [11] and related problems [2], or for the analysis of randomized diffusion processes [7] – to name just a few. On the practical side, a recent survey [23] mentions many machine learning applications such as computer vision, text analysis, recommender systems, network analysis, and representation learning.

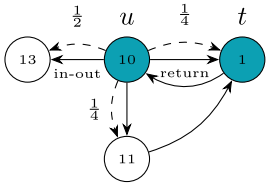


Fig. 1. Single-step example for node2vec ($p = 1$ and $q = \frac{1}{2}$); the walk is located on vertex $u = 10$, coming from $t = 1$. The probabilities for the next vertex are given next to the dashed arrows.

One distinguishes different types of random walks based on their probability distribution. The classic random walk (CRW) defines that the next vertex after u is chosen uniformly at random from $N^+(u)$: $p_{u,v} = \frac{1}{\deg^+(u)}$ [23]. Several variations exist, e. g. lazy random walks that stay on the current vertex with probability > 0 or the classic random walk with restart (RWR), which can always jump back to the start vertex with probability $c > 0$. Other popular first-order algorithms are PageRank and personalized PageRank, see [23].

The node2vec RW algorithm [10] is a prominent example for second-order random walks: in addition to the currently visited vertex u, the walk stores the previously visited vertex t. Moreover, node2vec uses two parameters, p (return) and q (in-out). The transition probability for edge (u, v) in node2vec, which is now extended by one dimension, is then defined as $p_{u,v,t} = \frac{\beta(u,v,t)}{\sum_{i \in N^+(u)} \beta(u,i,t)}$, where $\beta(u, v, t) = \frac{1}{p}$ if $v = t$, $= 1$ if $t \in N^+(v)$, and $= \frac{1}{q}$ if $t \notin N^+(v)$, see Fig. 1 for an example. Walking similarly to a breadth-first manner requires $p \leq 1$ and $q > 1$, whereas mimicking a depth-first manner requires $p \geq 1$ and $q < 1$. The co-occurrence of vertices in fixed-length second-order RWs can be used as a similarity measure, which is employed in representation learning to compute a structure-preserving embedding for further analysis, also see [23].

2.2 Related Random Walk Engines

Given the breadth of applications, there are several requirements for a random walk engine: (i) It should support a variety of RW types, at least first- and second-order RWs. The implementation should be generic, but allow the instantiation of specific RWs by the user via a convenient API. (ii) To integrate the engine into some larger workflow, it is advantageous if the engine is provided as a library. (iii) The computed walks should be provided as output in a single file or in some data structure returned from a library call. (iv) High speed by using parallelism since many applications require a large number of RWs. Table 1 lists the portrayed engines and the respective requirements they fulfill.

Table 1. List of relevant random walk engines and their features.

Name & Reference	RW Type		Library	Path Accessibility	Parallelism		Architecture
	First-Order	Second-Order			Threads	Processes	
GraphWalker [21]	✔	✘	✘	✘	✔	✘	single-CPU
ThunderRW [19]	✔	✔	✘	✘	✔	✘	single-CPU
FlashMob [24]	✔	✔	✘	✘	✔	✘	single-CPU
GraSorw [12]	✔	✔	✘	✘	✔	✘	single-CPU
Skywalker [20]	✔	✔	✘	✘	✔	✘	GPU
FlowWalker [14]	✔	✔	✘	✘	✔	✘	GPU
KnightKing [25]	✔	✔	✘	✘	✔	✔	multi-CPU
Ours: ScaleRunner	✔	✔	✔	✔	✔	✔	multi-CPU

GraphWalker [21] is a multi-threaded engine for single-CPU systems, where each thread handles the next unprocessed random walk – and this happens independently from other threads and random walks (*walk-centric perspective*). It addresses several implementation challenges by (i) using a *state-aware I/O model* which dynamically loads subgraphs from disk depending on the current state of a walk, and (ii) buffering and offloading walk states asynchronously to

disk to reduce the memory footprint. Like GraphWalker, ThunderRW [19] is multi-threaded for shared memory. Based on the observation that a walk-centric perspective can lead to memory stalls, ThunderRW implements a *step-centric model*. In this model, prefetching ensures that a thread processes only vertices whose data required to compute the next step is already in cache.

The third multi-threaded engine FlashMob [24] addresses memory latency issues in a different way: it partitions the input graph hierarchically to fit the resulting subgraphs into different cache levels. In their experiments, the FlashMob authors use graphs that fit into L1 to L3 cache and those that must be streamed from DRAM. After analyzing the frequency of vertex visits running a first-order RW for multiple graphs, they emphasize the importance of addressing high-degree vertices for obtaining high speed [24, Sec. 3]. They also implemented an *edge sampling* method to mitigate certain memory locality issues.

GraSorw [12] was written to compute RWs on large graph files that do not fit into the RAM of a single-CPU system, similar to GraphWalker. To fulfill this requirement, they partition the original graph into blocks and dynamically load these blocks into memory. When holding a subgraph in memory, they make progress for all scheduled RWs currently walking on vertices cached in memory.

Skywalker [20] is made for single-GPU systems. Instead of using a sampling method for computing the next step, Skywalker queries an alias table, which takes constant time. This remedies caching problems occurring with graphs that have a non-negligible number of high-degree vertices. FlowWalker is also a single-GPU engine, but it follows a sampler-centric RW computation model [14]. The program addresses memory space issues of GPU engines such as Skywalker and implements a special sampling technique for GPUs.

The above mentioned engines scale on a single compute node using multi-threading or single-GPU parallelism. In order to reach higher speeds (especially for massive graphs), an additional scaling layer along compute nodes is required.

KnightKing [25] targets multi-CPU systems and is composed of C++ programs that employ hybrid parallelism using MPI+OpenMP. The graph is 1D-distributed over the compute nodes, where each compute node owns a disjoint subgraph (plus ghost vertices). Initially, each compute node is responsible for a fair share of random walks. Yet, when a random walk leaves a node's subgraph, it is transferred to the neighboring compute node. Other than Skywalker, KnightKing uses acceptance-rejection sampling [4] to compute the next step. Even though this may suffer from a high number of rejections, it can help to avoid a large alias table.

As argued above and visible from Table 1, none of the portrayed engines fulfills all requirements. GPU-based engines cannot handle massive graphs, shared-memory engines do not exploit multi-CPU scalability, while KnightKing is a collection of programs with usability deficiencies (such as not separating file I/O routines from the walk engine), which limits the possible usage areas.

Note that DistGer-Pipe [6] is a recent multi-CPU engine we became aware of too late to include it in detail, e. g. in our experiments. It offers first- and second-order RWs and end-to-end embedding functionality for RL. In their RW

experiments, the authors observe an average speedup compared to KnightKing of 3.32. Our average speedup on KnightKing, in turn, is at least 16 (Sect. 4).

3 ScaleRunner RW Engine: Design and Implementation

3.1 Challenges and Overview

A parallel random walk engine does not only need fast RW computations, it should also be easy to use for implementing application-specific RW algorithms. The challenge here is to find the right balance between (i) a simple and powerful API to implement RW algorithms, and (ii) requirements the user of the engine has to fulfill in order to accelerate the implemented algorithm. For example, the user may need to provide program data in specified ways, such as thread-private or -shared data to support race-free multi-threading. Also, to handle large data sets smoothly, RW engines should provide fast I/O routines.

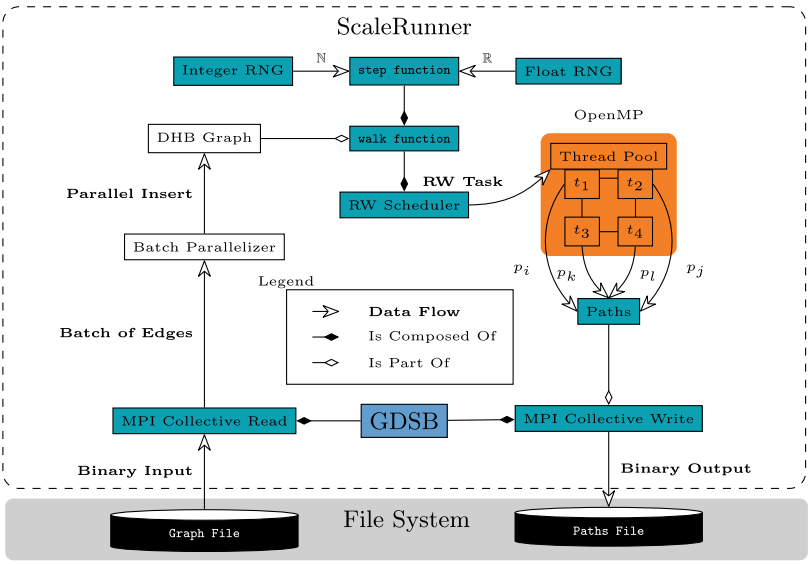


Fig. 2. Overview of our software architecture. ScaleRunner is designed such that the I/O components can easily be exchanged. By default, file I/O is handled using MPI collective routines provided by our libray GDSB: after inserting all edges into the graph data structure DHB, the *RW Scheduler* deploys tasks to a thread pool executing the specified RW algorithm. An RW algorithm is separated into two functions: (i) the `walk function`, which calls (ii) the `step function` to make a step based on random numbers from random number generators (RNG). Every task inserts the generated path data into a pre-allocated path object, which allows for streaming of path data using MPI file output routines. Finally, all paths are written to disk using MPI collective file write routines ready for postprocessing.

Furthermore, when randomly sampling the neighborhood for second-order RW algorithms, the edge transition probabilities depend on the previous step; (re)setting the probabilities accordingly requires careful optimization. Finally, the produced paths have to be stored to memory without fragmentation to (a) stream path data to disk fast, or (b) use in downstream software components.

Figure 2 shows an overview of our software architecture, together with a short explanation of all layers and components in the caption. Details are given in Sect. 3.2, its structure follows the data flow from input over processing to output.

3.2 ScaleRunner Components

Binary Input Using GDSB. For handling the graph and path I/O for ScaleRunner, we additionally contribute the Graph Data Structures & Benchmark (GDSB) library as separate tool, designed as an open-source C++ library[2]. GDSB provides an ecosystem for experimental graph algorithmics including graph data structures, graph file I/O routines with full MPI functionality, as well as utilities to implement and run experiments.

When reading in graph data, relying on the original text files provided by graph repositories such as NetworkRepository (NR) or SNAP is undesired for two reasons: (i) a non-negligible number of instances does not follow the file format meticulously; (ii) reading in large text files is slow and thus inhibits the overall processing pipeline. To address these issues, we convert all graphs to the binary format of GDSB. This format does not only improve the read performance, but also enables our tool to read edge data using MPI file I/O routines.

Using GDSB, ScaleRunner reads in batches of edges collectively on all compute nodes. By not reading in all edges at once into the dynamic graph data structure DHB described below, we limit the intermediate buffer size of the edges that are read. Especially for large graph data sets this is an important aspect to limit the amplitude of the memory footprint for graph data input routines.

DHB. The dynamic graph data structure DHB [9], short for *dynamic hashed blocks*, stores vertex neighborhoods both array- and hash table-based. This approach allows constant-time edge access and update routines. DHB allows us to stream data from disk by inserting batches of edges in parallel using multiple threads, which is handled by the new *Batch Parallelizer*.

By collectively reading in the same edge batches on all compute nodes, we replicate the graph data on each node. We do not consider this a severe limitation since most graphs fit into the memory of each compute node of a modern cluster when considering the compute intensity of downstream tasks such as graph machine learning. Using the Batch Parallelizer, DHB's block-based architecture allows to insert edges with different source ID in a multi-threaded context. To this end, the Batch Parallelizer first sorts the edges by their source and then distributes the sorted array over all threads to be inserted.

[2] GDSB Release Version 1.0: https://github.com/hu-macsy/graph-ds-benchmark.

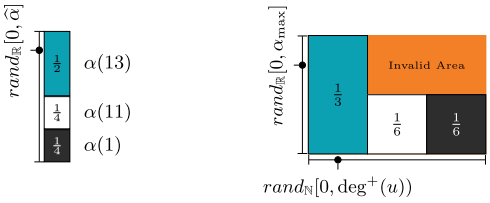
DHB is specifically developed for reducing the cache locality issues of random neighborhood accesses in graphs with a skewed degree distribution. Those graphs usually have a small, but non-negligible number of high-degree vertices (HDVs). As observed by Yang et al. [24, Sec. 3], those HDVs are the vertices with the highest visit frequency in RWs. Since the block-based design of DHB efficiently caches these HDVs, RWs are less likely to yield cache misses on such graphs.

Random Number Generation. Each call of the `step function` (representing a single step of a RW) requires random numbers in certain intervals such as $[0, N(u))$ and a sampling algorithm that determines the next vertex accordingly.

When sampling W random walks in total, each of length L, one requires $\Theta(W \cdot L)$ random numbers to be drawn. Hence, the random number engine employed as well as the concrete way to use it has significant impact on its execution time [15]. In our implementation, each thread owns a *Mersenne Twister* engine as well as a single distribution object per random number type T; this object is used to generate random numbers in $[0, \max(T)]$. For each random number within the desired interval, we apply type-safe modulo operations [8].

Sampling Within the Step Function. The sampling function for choosing the next vertex to walk to is called $\Theta(W \cdot L)$ times. For high speed it is thus important to carefully optimize the sampling algorithm used in the `step function`.

An obvious method to sample from $N^+(u)$ (when visiting $u \in V$) is called *inverse transform sampling* [16], whose sequential implementation is visualized in Fig. 3a. First, with $\alpha(v)$ being the edge transition probability of $v \in N^+(u)$ and $\widehat{\alpha} := \sum_{v \in N^+(u)} \alpha(v)$ the corresponding sum, we draw a random number $r := rand[0, \widehat{\alpha}]$. Second, we accumulate $acc := acc + \alpha(v)$ over the neighbors until $acc \geq r$. Hence, in total, for sequential inverse transform sampling, we draw one random number and iterate twice over $N^+(u)$.



(a) seq. inverse transform sampling (b) rejection sampling

Fig. 3. Sampling algorithm examples based on the node2vec sampling step of Fig. 1. (a) Sequential inverse transform sampling with color reference for the edge transition probabilities $\alpha(13), \alpha(11)$ and $\alpha(1)$. (b) Rejection sampling where a random throw has a chance of $\frac{1}{3}$ to be rejected by hitting the invalid area.

Rejection sampling [25], in turn, is based on the idea of representing the transition probabilities on a rectangle of size $\deg^+(u) \times \alpha_{\max}$, where $\alpha_{\max} := \max_{v \in N^+(u)} \alpha(v)$. To identify the next vertex, rejection sampling first computes $\alpha_{\max}$, followed by a random integer draw $r_i = rand_{\mathbb{N}}[0, \deg^+(u))$ to choose the x-coordinate. Then, the method requires an additional random floating point number $r_f = rand_{\mathbb{R}}[0, \alpha_{\max}]$ to *throw a dart* on the y-coordinate, see Fig. 3b. If the invalid area was hit, we repeat the sampling step. Otherwise, we identify the neighbor v corresponding to the area that was hit. In total, for rejection sampling we draw two random numbers and iterate once over $N^+(u)$, i. e., one additional random number but one less iteration over $N^+(u)$.

A third option would be to optimize for outliers in the rejection sampling algorithm, thereby reducing the total number of rejections [25, p.529]. Yet, in our preliminary experiments, rejection sampling was the by far fastest method. Thus, in the experiments of Sect. 4, ScaleRunner uses rejection sampling only.

RW Scheduler and Walk Function. The `walk function` takes the `step function` as well as the DHB object as parameters and computes a single walk. Each walk represented by the `walk function` is then dispatched to a thread pool by the *RW Scheduler*. Here, we make use of an OpenMP task-based system where each walk represents a task [18]. A task is scheduled to be executed by any thread available. Each of the $\hat{p}$ compute nodes must compute $\frac{W}{\hat{p}}$ RWs. Setting the granularity of each task to one random walk mitigates inherent problems of load imbalances due to different computational efforts in different RWs.

Binary Output. Finally, ScaleRunner writes the produced path data to a single file using collective MPI file I/O routines on all compute nodes. Using such collective I/O improves the speed; its disadvantage is the requirement to provide a pointer to a consecutive memory segment containing the path data. We always allocate an array of size $W \cdot L$, even if a walk is shorter than L. When packing data instead, each thread would need its own dynamically allocated memory segment. Such dynamic allocation would introduce two issues: (i) it would be slower than a single allocation before scheduling walks, and (ii) we would need to collect the thread-private data and map the memory segments to the layout required for collective MPI file output, also slowing down the output routine.

The described path output method can easily be exchanged or modified. Postprocessing applications can also use the *Paths* data structure directly.

4 Experiments

4.1 Setup

HW/SW Environment. All experiments[3] were run on a cluster with 16 compute nodes (= # processes), each equipped with 2× 12-Core Intel Xeon X6126

[3] SclaeRunner Experiments: https://github.com/hu-macsy/ScaleRunner-Exp.

Table 2. List of graph instances used in our experiments: original names, our introduced synonyms, assigned category and its color, common graph metrics.

Name	Synonym	Origin	Category	Color	n	m	$\deg^+_{\max}$	$\overline{\deg^+}$	$\rho(G)$
web-NotreDame	NotreDame	SNAP	web		326K	1.47M	3.44K	4.6	1.385e-05
rt-retweet-crawl	retweet	NR	social		1.11M	2.28M	2.54K	2.1	1.841e-06
cage14	cage14	NR	biological		1.51M	2.41M	38	1.7	1.065e-06
europe_osm	europe_osm	NR	road		50.9M	5.44M	13	0.2	2.100e-09
Amazon0601	Amzn0601	SNAP	collaboration		403K	3.39M	10	8.4	2.082e-05
human_gene2	human_gene2	NR	biological		14.3K	8.04M	4.59K	561	3.912e-02
web-Google	Google	SNAP	web		916K	5.11M	456	5.6	6.079e-06
roadNet-CA	roadNet	SNAP	road		1.97M	5.53M	12	2.9	1.424e-06
rec-amazon-ratings	amzn-ratings	NR	collaboration		2.15M	2.15M	1	1	4.660e-07
rec-epinions-user-ratings	epns-user	NR	collaboration		756K	13.7M	162K	18.1	2.393e-05
inf-road-usa	road-usa	NR	road		23.9M	28.9M	8	1.3	5.031e-08
bn-human-Jung2015_M87125334	hmn-Jung-MX5334	NR	biological		1.83M	40.3M	2.31K	22.1	1.206e-05
soc-LiveJournal1	LiveJournal	SNAP	social		4.85M	68.5M	20.3K	14.2	2.914e-06
soc-orkut	orkut	NR	social		3M	106M	3.14K	35.5	1.184e-05
bn-human-Jung2015_M87126525	hmn-Jung-MX6525	NR	biological		1.83M	146M	6.98K	80	4.376e-05
web-uk-2005-all	web-uk-2005-all	NR	web		39.5M	921M	5.21K	23.4	5.917e-07
soc-twitter-mpi-sws	twitter-mpi	NR	social		41.7M	1.47B	3M	35.3	8.464e-07

(HT) CPUs, and 192 GB RAM. The compute nodes are inter-connected by a 100 Gb Infiniband Omnipath network. C++ code was compiled using GCC v12.3 and MPICH v4.2.0. For setting up and running all experiments, we use the framework SimexPal [1]. Our competitor KnightKing could not succeed for some instances due to program crashes during RWs, rendering compute nodes unreachable. An extensive and time-costly search for the issue was unsuccessful. All instances marked as *failed* in the following have failed repeatedly.

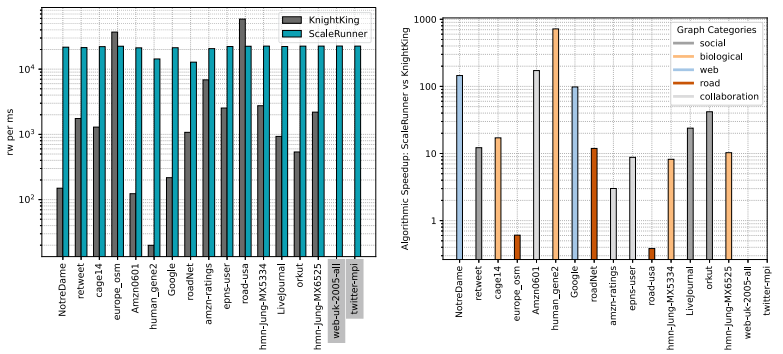
In order to run systematic and reliable experiments, we adapted[4] the KnightKingI/O code to use collective MPI routines. For the comparisons, we report throughput in RWs per millisecond (ms) as well as the algorithmic speedup (throughput of ScaleRunner divided by throughput of KnightKing) for each graph instance excluding graph or path I/O times.

Data Sets. Information on the origin of all graphs used in our experiments including several basic metrics is shown in Table 2. If a graph data set contains timestamped edges, the edge data with the higher timestamp is used. We preprocess the graph data by converting every graph instance to the open-source binary format of the library GDSB. This process does not modify the data but allows for faster binary read operations. Moreover, if a graph is undirected, we interpret it as bidirected. Weights are represented in single-precision. Parameterized random walk algorithms of second order embed community and centrality measures well when computing $\mu \ll n$ paths [10,17]. Our choice of computing n random walks in each experiment is therefore sufficient.

[4] kklib "KnightKing as a library": https://github.com/hu-macsy/kklib.

4.2 Evaluation

First Order: CRW. First, we investigate a a Classic Random Walk (CRW), which is an important fundamental use case for measuring the performance of our tool. We compute n paths of length 80 on 16 compute nodes. A similar setting was proposed by the authors of KnightKing. As can be seen in Fig. 4, for all instances ScaleRunner produces more than 10 000RWs per ms.

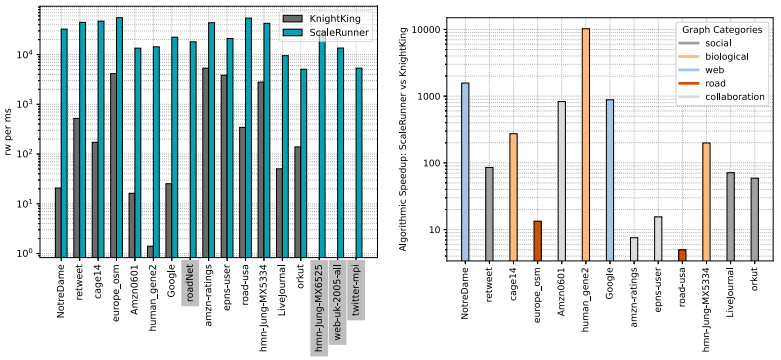


(a) throughput in $\frac{rw}{ms}$ (b) algorithmic speedup: ScaleRunner vs KnightKing.

Fig. 4. Experiment executing the first-order RW algorithm CRW. (a) Number of RWs per ms: $\frac{rw}{ms}$. Graph names in gray have failed for KnightKing. (b) Algorithmic speedup of ScaleRunner compared to KnightKing for all graph instances KnightKing could solve. Higher is better.

Our tool ScaleRunner generally outperforms KnightKing in terms of throughput, with the exception of two *Road Networks* ■: europe_osm and road-usa. In general, we expect that for *Road Networks* ■, the RWs in KnightKing cross compute node frontiers less frequently due to the graphs' high diameter. The graph data structure DHB used in ScaleRunner, in turn, is optimized for graphs with certain characteristics such as a skewed degree distribution. These characteristics are not prevalent in road networks, which have a low density and degrees closely concentrated around the (rather small) mean.

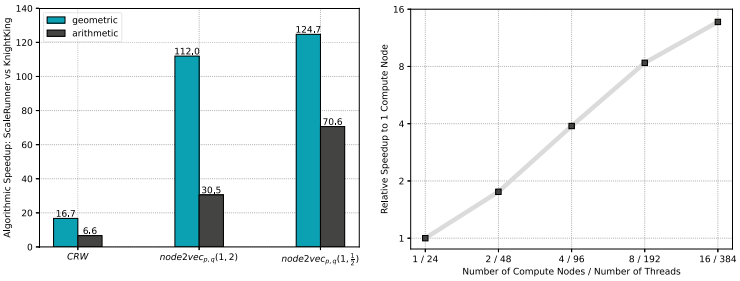
Second-Order RW: node2vec. Due to the necessary recomputation of the transition probabilities in second-order RWs, the node2vec algorithm is expected to reveal our engine's sensitivities to the different graph categories. We use the same parameters, experiment setup, and evaluation methods as in the first-order RW experiment above. For the node2vec hyperparameters (p, q), we present experiments using setting $(1, \frac{1}{2})$. The other setting $(1, 2)$ shows a similar trend, as can be observed from the aggregated algorithmic speedup plot in Fig. 6a. As can be seen in Fig. 5a, ScaleRunner is faster than KnightKing on all graphs now. In Fig. 5b we show the corresponding algorithmic speedups: for



(a) throughput in $\frac{rw}{ms}$. (b) algorithmic speedup: ScaleRunner vs KnightKing.

Fig. 5. Results for second-order RW algorithm node2vec with hyperparameter setting $p = 1$, $q = \frac{1}{2}$: (a) throughput (graph names in gray failed for KnightKing), (b) algorithmic speedup of ScaleRunner per graph. Higher is better.

Webgraphs ■ our tool significantly outperforms KnightKing by two orders of magnitude. For *Biological Networks* ■, ScaleRunner demonstrates an even higher advantage over KnightKing, achieving performance gains from two to four orders of magnitude. At the other end of the spectrum, we observe a smaller performance advantage for *Road Networks* ■ than for the other categories. Yet, notice that we still outperform KnightKing by a significant factor.



(a) (b)

Fig. 6. (a) Cumulative algorithmic speedups comparing our tool ScaleRunner with KnightKing for all successful experiments. **geometric:** average algorithmic speedup (geometric mean of all graph instances); **arithmetic:** relative running time using the sum of all RW experiment durations. (b) Strong CPU scaling results w. r. t. throughput for ScaleRunner.

Aggregated Algorithmic Speedups and Strong Scaling. In Fig. 6a we show the average algorithmic speedup compared to KnightKing running first- and second-order RW algorithms. For averaging these ratios, we use the geometric mean; it has the additional advantage of being insensitive to samples at the extremes. Regarding the algorithmic speedup, ScaleRunner is 16.7× on average faster than KnightKing for the CRW algorithm (first order). The second-order RW algorithm node2vec is accelerated even more: ScaleRunner is 112.0× ($q = 2$) to 124.7× ($q = \frac{1}{2}$) faster. We think that $node2vec_{p,q}(1, 2)$ has a lower algorithmic speedup than $node2vec_{p,q}(1, \frac{1}{2})$ because an RW is more likely to explore distant vertices if $q > p$; hence, this setting is disadvantageous for graph data caching, reducing the throughput at which an RW is computed. Also note that KnightKing fails on the three largest graphs, which leads to a distorted average, likely to our disadvantage given the general trend.

Finally, we showcase CPU strong scaling for ScaleRunner in Fig. 6b from 1 to 16 compute nodes (24 to 384 threads). The speedups are computed as ratios of the respective average throughput on all graph instances on 2 to 16 compute nodes compared to one – and show a (desirable) nearly-linear scaling behavior.

5 Conclusions and Future Work

The goal of this work was to improve the performance of executing first- and second-order random walk algorithms for multi-CPU systems. To this end, we implemented the C++ library ScaleRunner using MPI+OpenMP parallelism. ScaleRunner provides a simple, yet powerful API to implement first- and second-order random walks – with a focus on performance-optimized RW functionality for unweighted *and* weighted graphs. Our experimental results show significant throughput improvements over KnightKing. They also indicate that higher gains are possible for more complex RW algorithms.

Other than KnightKing, our tool replicates the full graph on each compute node. If extreme-scale graphs exceed this memory, we would have to distribute different parts of the graph with a graph partitioner. This is left for future work because typical downstream tasks would take extremely long on such large graphs. More importantly, we plan to execute large-scale experiments on a cluster with more than 16 compute nodes. So far, crashing compute nodes when running KnightKing were problematic for executions on clusters not managed by us.

Acknowledgments and Artifact Availability. This work is partially supported by German Research Foundation (DFG) grant GR 5745/1-1 (DyANE).

The artifact is available in the Zenodo repository [22].

Disclosure of Interests. The authors have no competing interests to declare that are relevant to the content of this article.

References

1. Angriman, E., et al.: Guidelines for experimental algorithmics: a case study in network analysis. Algorithms **12**(7) (2019)
2. Angriman, E., Predari, M., van der Grinten, A., Meyerhenke, H.: Approximation of the diagonal of a Laplacian's pseudoinverse for complex network analysis. In: 28th Annual European Symposium on Algorithms, ESA 2020. LIPIcs, vol. 173, pp. 6:1–6:24. Schloss Dagstuhl–Leibniz-Zentrum für Informatik (2020)
3. Cai, H., Zheng, V.W., Chang, K.C.C.: A comprehensive survey of graph embedding: problems, techniques, and applications. IEEE Trans. Knowl. Data Eng. **30**(9), 1616–1637 (2018)
4. Casella, G., Robert, C.P., Wells, M.T.: Generalized accept-reject sampling schemes. Lect. Notes Monogr. Ser. **45**, 342–347 (2004)
5. Chung, F.R.K.: Spectral Graph Theory. American Mathematical Society (1997)
6. Fang, P., et al.: Information-oriented random walks and pipeline optimization for distributed graph embedding. IEEE Trans. Knowl. Data Eng. **37**(1), 408–422 (2025)
7. Giakkoupis, G., Saribekyan, H., Sauerwald, T.: Spread of information and diseases via random walks in sparse graphs. In: 34th International Symposium on Distributed Computing, DISC 2020. LIPIcs, vol. 179, pp. 9:1–9:17. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2020)
8. Goualard, F.: Drawing random floating-point numbers from an interval. ACM Trans. Model. Comput. Simul. **32**(3) (2022)
9. van der Grinten, A., Predari, M., Willich, F.: A fast data structure for dynamic graphs based on hash-indexed adjacency blocks. In: 20th International Symposium on Experimental Algorithms (SEA 2022). LIPIcs, vol. 233, pp. 11:1–11:18. Schloss Dagstuhl – Leibniz-Zentrum für Informatik (2022)
10. Grover, A., Leskovec, J.: Node2vec: scalable feature learning for networks. In: Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, pp. 855–864. KDD '16, ACM, New York, NY, USA (2016)
11. Kyng, R., Meierhans, S., Probst, M.: Derandomizing directed random walks in almost-linear time. In: 63rd IEEE Annual Symposium on Foundations of Computer Science, FOCS 2022, Denver, CO, USA, pp. 407–418. IEEE (2022)
12. Li, H., Shao, Y., Du, J., Cui, B., Chen, L.: An i/o-efficient disk-based graph system for scalable second-order random walk of large graphs. Proc. VLDB Endow. **15**(8), 1619–1631 (2022)
13. Lovász, L.: Random walks on graphs: a survey, Technical report, Yale University, Department of Computer Science (1994)
14. Mei, J., et al.: Flowwalker: a memory-efficient and high-performance GPU-based dynamic graph random walk framework. arXiv preprint arXiv:2404.08364 (2024)
15. Michaels, R.: Fast, high-quality pseudo-random numbers for non-cryptographers in c++ (2022). https://www.youtube.com/watch?v=I5UY3yb0128
16. Pandey, S., Li, L., Hoisie, A., Li, X.S., Liu, H.: C-saw: a framework for graph sampling and random walk on GPUs. In: SC20: International Conference for High Performance Computing, Networking, Storage and Analysis, pp. 1–15 (2020)
17. Perozzi, B., Al-Rfou, R., Skiena, S.: Deepwalk: online learning of social representations. In: Proceedings of the 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, pp. 701–710. KDD '14, ACM, New York, NY, USA (2014)

18. Podobas, A., Karlsson, S.: Towards unifying OpenMP under the task-parallel paradigm. In: Maruyama, N., de Supinski, B.R., Wahib, M. (eds.) OpenMP: Memory, Devices, and Tasks, pp. 116–129. Springer, Cham (2016)
19. Sun, S., Chen, Y., Lu, S., He, B., Li, Y.: ThunderRW: an in-memory graph random walk engine. Proc. VLDB Endow. **14**(11), 1992–2005 (2021)
20. Wang, P., et al.: Skywalker: efficient alias-method-based graph sampling and random walk on GPUs. In: 30th International Conference on Parallel Architectures and Compilation Techniques (PACT), pp. 304–317 (2021)
21. Wang, R., Li, Y., Xie, H., Xu, Y., Lui, J.C.S.: Graphwalker: an i/o-efficient and resource-friendly graph analytic system for fast and scalable random walks. In: USENIX Annual Technical Conference (ATC) 2020. USENIX Association, USA (2020)
22. Willich, F., Meyerhenke, H.: Artifact of the paper: Scalerunner: a fast MPI-based random walk engine for multi-CPU systems (2025). https://doi.org/10.5281/zenodo.15593388
23. Xia, F., Liu, J., Nie, H., Fu, Y., Wan, L., Kong, X.: Random walks: a review of algorithms and applications. IEEE Trans. Emerg. Top. Comput. Intell. **4**(2), 95–107 (2020)
24. Yang, K., Ma, X., Thirumuruganathan, S., Chen, K., Wu, Y.: Random walks on huge graphs at cache efficiency. In: Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles, pp. 311–326. SOSP'21, ACM, New York, NY, USA (2021)
25. Yang, K., Zhang, M., Chen, K., Ma, X., Bai, Y., Jiang, Y.: Knightking: a fast distributed graph random walk engine. In: Proceedings of the 27th ACM Symposium on Operating Systems Principles, pp. 524–537. SOSP '19, ACM, New York, NY, USA (2019)
26. Yen, L., Fouss, F., Decaestecker, C., Francq, P., Saerens, M.: Graph nodes clustering based on the commute-time kernel. In: Advances in Knowledge Discovery and Data Mining, pp. 1037–1045. Springer, Berlin, Heidelberg (2007)