Scalable Graph Analysis and Clustering on Commodity Hardware.

by

Disa Mhembere

A dissertation submitted to The Johns Hopkins University in conformity with the requirements for the degree of Doctor of Philosophy.

Baltimore, Maryland

May, 2019

Abstract

The abundance of large-scale datasets both in industry and academia today has lead to a need for scalable data analysis frameworks and libraries. This assertion is exceedingly apparent in large-scale graph datasets. The vast majority of existing frameworks focus on distributing computation within a cluster, neglecting to fully utilize each individual node, leading to poor overall performance. This thesis is motivated by the prevalence of Non-Uniform Memory Access (NUMA) architectures within multicore machines and the advancements in the performance of external memory devices like SSDs. This thesis focusses on the development of machine learning frameworks, libraries, and application development principles to enable scalable data analysis, with minimal resource consumption. We develop novel optimizations that leverage fine-grain I/O and NUMA-awareness to advance the state-of-the-art within the areas of scalable graph analytics and machine learning.

We focus on minimality, scalability and memory parallelism when data reside either in (i) memory, (ii) semi-externally, or (iii) distributed memory. We target two core areas: (i) graph analytics and (ii) community detection (clustering). The

ABSTRACT

semi-external memory (SEM) paradigm is an attractive middle ground for limited resource consumption and near-in-memory performance on a single thick compute node. In recent years, its adoption has steadily risen in popularity with framework developers, despite having limited adoption from application developers. We address key questions surrounding the development of state-of-the-art applications within an SEM, vertex-centric graph framework. Our target is to lower the barrier for entry to SEM, vertex-centric application development. As such, we develop Graphyti, a library of highly optimized applications in Semi-External Memory (SEM) using the FlashGraph framework. We utilize this library to identify the core principles that underlie the development of state-of-the-art vertex-centric graph applications in SEM. We then address scaling the task of community detection through clustering given arbitrary hardware budgets. We develop the clusternor extensible clustering framework and library with facilities for optimized scale-out and scale-up computation. In summation, this thesis develops key SEM design principles for graph analytics, introduces novel algorithmic and systems-oriented optimizations for scalable algorithms that utilize a two-step Majorize-Minimization or Minorize-Maximization (MM) objective function optimization pattern. The optimizations we develop enable the applications and libraries provided to attain state-of-the-art performance in varying memory settings.

ABSTRACT

Primary Advisor: Prof. Randal Burns, PhD

Secondary Reader: Prof. Joshua T. Vogelstein, PhD

Tertiary Reader: Prof. Carey E. Priebe, PhD

Acknowledgments

I would like to thank my primary advisor Dr. Randal Burns for the countless hours spent working with me developing ideas and improving my knowledge, skills and thinking as a researcher. His advice, mentorship and guidance both professionally and personally continue to be invaluable. I am very grateful for his belief in me and his never ending support of my development throughout the years.

I would like to thank my unofficial secondary advisor Dr. Joshua Vogelstein for tutelage in graph theory, statistics and machine learning. His investment in my research efforts and critical thinking is greatly appreciated.

I would also like to thank Dr. Carey Priebe for the insightful discussions and lessons in graph analysis, and statistics. I am also very thankful for the opportunities he afforded me to work with his students, post-docs and knowledgeable collaborators.

I would like to thank my labmate and closest collaborator Dr. Da Zheng for the countless hours of idea sessions and discussions. Da's work provided the foundation for much of the content within this thesis. Thank you for your candidness, and willingness to assist in the development of the challenging problems addressed in

this document.

Next, I would like to thank my labmates and collaborators Kunal Lillaney, Greg Kiar, James Browne, Meghana Madhyastha, Will Gray Roncal, Eric Perlman, Brian Choi, Stephen Hamilton, Kalin Kanov, Jesse Patsolic, Tyler Tomita, and Runze Tang. Thank you for the positive working environment and friendship.

Finally, I would like the thank the individuals and agencies that funded this work. Specifically, the Paul V. Renoff foundation for their funding of my academic fellowship. Finally, I would like to thank DARPA for grant funding in the form of DARPA GRAPHS N66001-14-1-4028, DARPA SIMPLEX N66001-15-C-4041, and NSF ACI-1649880.

Dedication

This thesis is dedicated to those in my life who have fueled, inspired and encouraged me to strive for my goals and prosper. To you, my best friend, love and wife Pola, my loving parents, Zanele and Ngira, and my wonderful sister, Danai.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Introduction

Scalable machine learning is at the epicenter of academia and industry alike as datasets grow exponentially. Performing analysis on such data is challenging from the perspective of framework choice, programming paradigm and hardware selection. Furthermore, there is often interest in performing different types of analysis on data based upon the type of data available. Graphs naturally produced by social networks [1], transactional banking [2], telecommunications [3] and more, contain a wealth of information and require specialized processing frameworks and libraries for analysis. Furthermore, data derived from streaming services [4], connectomics [5–7], image/video processing [3] and more produce dense feature vector datasets for which different types of frameworks are necessary.

This thesis explores machine learning from the perspective of (i) graph analysis and (ii) clustering. The most popular programming model for scalable graph analysis is the vertex-centric paradigm. Furthermore, the semi-external memory paradigm for computing has been shown to provide comparable performance to in-memory on a single machine without the need to distribute computation to a cluster. Though its adoption among developers has been limited due to the

perceived difficulty in programming. We address questions regarding how to effectively and efficiently design graph applications in semi-external memory (SEM). We develop illustrative and generalizable optimizations for adoption by developers in addition to providing them as an open source package. We then tackle the task of clustering in-memory, in distributed memory and in SEM. We address questions of how to design and implement a scalable and extensible clustering environment.

Today's graphs far outsize the in-memory capacity of most standalone machines. As such, system developers move towards distributed and out-of-core solutions. SEM systems are an attractive alternative as they provide a reasonable trade-off between resource consumption and performance. Understanding how to programmatically achieve highly parallel, I/O minimal applications is critical to SEM adoption.

Furthermore, when considering dense structured, feature-vectorized datasets, community detection is of great importance. The decomposition of extremely large datasets into clusters of data points that are similar is a topic of great interest in industry and academia. Clustering multi-billion data points is essential to targeted ad-driven organizations such as Google [8]. Behavioromics [9] uses clustering to map neurons to distinct motor patterns. In genetics, clustering is used to infer relationships between genetically similar species [10, 11].

The challenge with developing SEM vertex-centric applications is that now vertices must explicitly request edge data from disk. Additionally, one must maintain at most $\mathcal{O}(n)$ in-memory state for an n vertex, m edge graph. Vertex-centrism necessitates algorithmic evolution; SEM adds another layer of complexity as developers must now also encode I/O and memory minimalism into applications. These components constitute barriers to entry for application development.

The greatest challenges facing clustering tool builders are *(i)* reducing the cost of the synchronization barrier between the MM steps, *(ii)* mitigating the latency of data movement through the memory hierarchy, and *(iii)* scaling to arbitrarily large datasets. In addition, fully asynchronous computation of both MM steps is mostly infeasible because global state updates are performed between each step. The resulting global barriers pose a major challenge to the performance and

scalability of parallel and distributed implementations. This is especially true for data that require large numbers of iterations to converge.

Popular scalable frameworks and libraries [12–14] have converged on scale-out, distributed processing in which data are partitioned among cluster nodes, often randomly, and global updates are transmitted at the speed of the interconnect. These frameworks are negatively affected by inefficient data allocation, management, and task scheduling protocols. These negative attributes are exacerbated for both graph algorithms and clustering. This design incurs heavy network traffic owing to data shuffling and centralized master-worker designs.

A current trend for hardware design scales up a single machine, integrating large memories and using solid-state storage devices (SSDs) to extend memory capacity. This conforms to the node design for supercomputers [15]. Recent findings [16,17], show that increasingly large graph analytics tasks can be done on a small fraction of the hardware, at less cost, as fast, and using less energy on a single shared-memory node. As such, we advocate for the SEM approach to graph library development.

Our findings on scalable clustering reveal similar structure as graph analytics though to a lesser degree. We discover that there is need for a fully inclusive hybridized design that encompasses fully in-memory, SEM and distributed capabilities. Doing so permits users to select both the application performance and scalability requirements. A core argument we develop is that most clustering frameworks neglect to optimize computation within single machines before distributing computations. This thesis demonstrates the massive performance improvements, and resource savings foregone by doing so.

We present Graphyti, a vertex-centric SEM graph library developed for use in Python. Through illustrative examples implemented within Graphyti, we present guiding principles and techniques which serve to lower the barrier of entry for the development of state-of-the-art, vertex-centric, SEM applications. We demonstrate the practical application of these principles through Graphyti and release it as an extensible library. We demonstrate that when optimized, SEM graph applications

can perform on par and even surpass the performance of popular distributed graph frameworks.

We then present clusternor, a scalable hybridized clustering framework and library. clusternor prefers scale-up computation on shared-memory multicore machines in order to eliminate network traffic and perform fine-grained synchronization. Once datasets outgrow in-memory computation and SEM, it is then appropriate to perform distributed computation. clusternor provides this capability while still optimizing per-machine computation. clusternor introduces a novel NUMA-aware data partitioning scheme and scheduler for MM algorithms that is applicable in all memory settings. Additionally, clusternor develops a hierarchical clustering model that eliminates recursion and maximizes cache line utility. Lastly, clusternor introduces an algorithmic advancement on Elkan's triangle inequality algorithm (TI) [18] for algorithms that contain k-means. We develop the Minimal Triangle Inequality (MTI) algorithm that scales to large-scale datasets.

1.2 Background

We describe the **architecture** building blocks upon which this thesis relies. We begin by describing SAFS (Section 1.2.3) the file system on which FlashGraph (Section 1.2.4) is built. FlashGraph is a fundamental component that we modify and utilize for the development of Graphyti (Chapter 2), and the Semi-External-Memory capabilities of clusternor (Chapter 4).

We then describe the **algorithmic** building blocks upon which this thesis relies. We describe the k-means [19] algorithm in Section 1.2.5. This core algorithm is the basis upon which we develop clusternor (Chapter 4). A core argument we develop is that k-means is fundamental to many highly utilized clustering algorithms today. As such, we develop the clusternor system based on this argument. Finally, we describe the triangle inequality with bounds [18] computation reduction/pruning technique for k-means in Section 1.2.5. This thesis develops an asymptotically more efficient variant of this algorithm.

1.2.1 Nomenclature

Throughout this thesis, the following conventions are assumed. Let $\mathbb{N}$ be the set of all natural numbers. Let $\mathbb{R}$ be the set of all real numbers. Let $\vec{v}$ be a d-dimensional vector in dataset $\vec{V}$ with cardinality, $|\vec{V}| = n$. Let j be the number of iterations of the algorithm we perform. Let $t \in \{0...j\}$ be the current iteration of the algorithm. Let $\vec{c}^t$ be a d-dimension vector representing the mean of a cluster (i.e., a centroid), at iteration t. Let $\vec{C}^t$ be the set of the k centroids at iteration t, with cardinality $|\vec{C}^t| = k$. In a given iteration, t, we can cluster any point, $\vec{v}$ into a cluster $\vec{c}^t$.

For some algorithms, we use Euclidean distance $\mathbf{d}$ as the dissimilarity metric between any $\vec{v}$ and $\vec{c}^t$, such that $\mathbf{d}(\vec{v}, \vec{c}^t) =$

$$\sqrt{(\vec{v}_1 - \vec{c}_1^t)^2 + (\vec{v}_2 - \vec{c}_2^t)^2 + ... + (\vec{v}_{d-1} - \vec{c}_{d-1}^t)^2 + (\vec{v}_d - \vec{c}_d^t)^2}.$$

Let $f(\vec{c}^t | t > 0) = \mathbf{d}(\vec{c}^t, \vec{c}^{t-1})$. Finally, let T be the number of threads of concurrent execution, P be the number of processing elements available (e.g. the number of cores in the machine), and N be the number of NUMA nodes.

1.2.2 Non-Uniform Memory Access Architectures

NUMA machines (Figure 1.1) constitute the majority of multi-socket machines within commodity grade servers today. NUMA machines are characterized by NUMA zones associated with each socket. A chip with CPUs on a socket within a NUMA zone has affinity to the memory within that particular zone. CPUs with affinity to a particular zone enjoy reduced latency and increased throughput for memory access compared to accessing memory banks within other NUMA zones. Memory accesses to non-local memory banks constitute Remote Memory Accesses (RMA). RMA is penalized because data must traverse through the NUMAlink interconnect before reaching the NUMA node-local memory bank, and finally caches, where data must then traverse the caching hierarchy before it is within registers accessible by the CPU. The latency and throughput degradation due to RMA is referred to as "NUMA effects". This work explores how frameworks can exploit NUMA effects to their advantage to improve application performance without sacrificing parallelism, load

balancing and scalability.

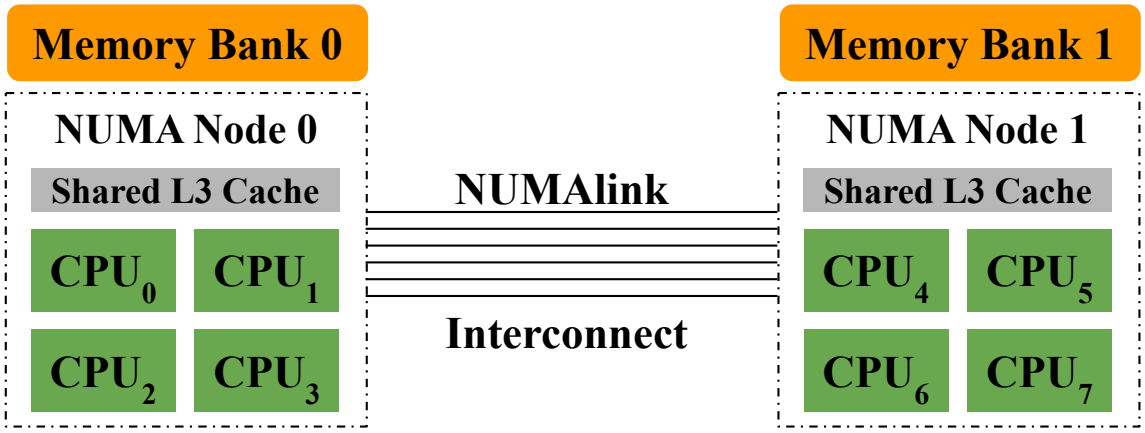


Figure 1.1: An example of a 2 NUMA node (2 Socket), 8 CPU machine.

1.2.3 Set-Associative File System (SAFS)

SAFS [20] is a file system developed to extract maximal file system IOPS from hardware arrays of solid state drives (SSDs) in NUMA machines running Linux. SAFS introduces a novel page caching design that overcomes the bottlenecks associated with traditional disk array solutions such as distributed file systems and Redundant Arrays of Independent Disks (RAID). SAFS demonstrates that inexpensive commodity hardware can produce performance comparable to that of customized alternatives that are orders of magnitude more expensive. Additionally, SAFS introduces optimizations to mitigate the bottlenecks introduced by:

- lock contention in device drivers and the operating system.
- the lack of support for multicore NUMA processors and SSDs.
- the design of page caches and device drivers in Linux.
- the several layers within the Linux block subsystem that must be traversed for disk-parallel I/O.

- the lack of support for a unified page cache view for I/O merging even for Linux Asynchronous I/O (AIO).

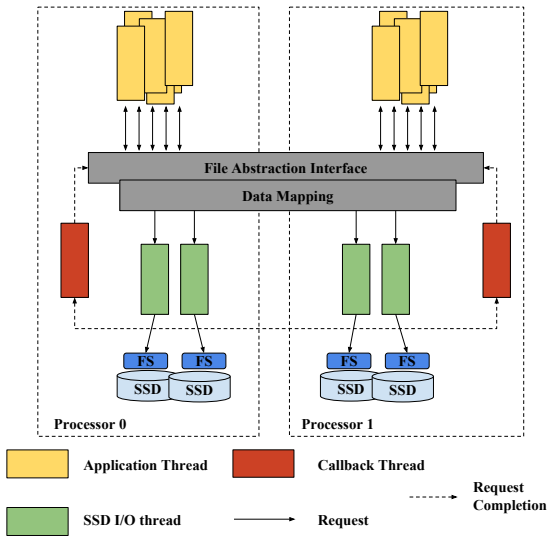


Figure 1.2: The architecture and design of the SAFS.

FlashGraph [17], discussed in Section (1.2.4) relies on SAFS for high speed parallel I/O, I/O merging and page caching to deliver state-of-the art performance for SEM applications. The SAFS page cache delivers roughly 2X more user-perceived IOPS for applications, improving its suitability for high-speed parallel random and sequential I/O.

1.2.4 FlashGraph

FlashGraph [17] is a SEM graph computation framework that places edge data on SSDs and allows user-defined vertex state to be held in memory. FlashGraph partitions a graph then exposes a vertex-centric programming interface that permits users to define functions written from the perspective of a single vertex, known as *vertex programs*. Parallelization is obtained from running multiple vertex programs concurrently. The vertex-centric interface was introduced by Google's Pregel engine [21] and became the most popular abstraction for graph parallelism [22–24].

The key differentiating component of FlashGraph is the addition of a semi-external memory

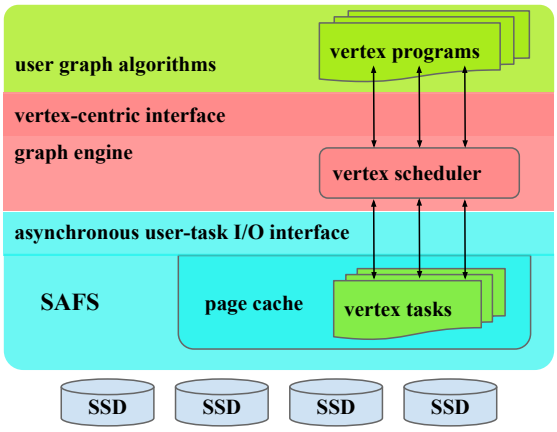


Figure 1.3: The architecture and design of FlashGraph.

interface to the vertex-centric paradigm. The implication is that users must now encode I/O into application development by specifically requesting egde data they require. FlashGraph then overlaps I/O with computation to mask latency in data movement through the memory hierarchy, delivering data to the page cache for consumption by applications. FlashGraph is also tolerant to in-memory failures, allowing recovery in SEM routines through lightweight check-pointing.

Both SAFS and FlashGraph work to merge I/O requests when requests are made for data located near one another on disk. This I/O merging amortizes the cost of accesses to SSDs. SAFS directly exposes FlashGraph to its page cache, allowing vertex programs to access it when a vertex's requested data is available. The key components in the architecture of FlashGraph are illustrated in Figure 1.3.

The programming interface for FlashGraph permits vertices to be in of four states:

- *inactive*: A vertex will not be processed by the graph engine in the upcoming iteration.
- *active*: A vertex will be processed by the graph engine in the upcoming iteration, at which point it may request data residing on disk.
- *running*: A vertex's requested data is within the page page of FlashGraph and can now be

utilized within in the user defined `run` method.

1.2.5 k-means

K-means is an intuitive and highly popular method of clustering n points in d-dimensions into k clusters. K-means maximizes within-cluster similarity and cross-cluster variance. The objective function being minimized by k-means is the residual sum of squares (RSS), i.e., the sum of squared distances of each sample/data point from it's nearest centroid:

$$RSS_k = \sum_{\vec{v} \in \vec{V}} |\vec{v} - \vec{\mu}|^2, \tag{1.1}$$

in which $\vec{\mu}$ is the nearest centroid to any data point, $\vec{v}$.

The most popular synchronous variant of k-means is Lloyd's algorithm [25]. Similar to Expectation Maximization [26], Lloyd's algorithm proceeds in two phases. However, k-means is a Majorize-Minimization or Minorize-Maximization (MM) algorithm as it performs hard clustering, in which each data point is assigned exactly one cluster. EM algorithms perform soft-clustering in which each data point is assigned a probability of cluster membership. Phase one of k-means computes the distance from each data point to each centroid (cluster mean). In phase two, we update the centroids to be the mean of their membership. This proceeds until the centroids no longer change from one iteration to the next. The algorithm locally minimizes within-cluster *distance*, for some distance metric that often is Euclidean distance (Section 1.2.1).

1.2.6 Triangle Inequality with Bounds

Elkan's algorithm for triangle inequality pruning with bounds (TI) [18] reduces potential distance computations between data points and centroids in k-means. TI relies on the fact that for any three points, $\vec{x}, \vec{y}, \vec{z}$:

$$\mathbf{d}(\vec{x}, \vec{z}) \leq \mathbf{d}(\vec{x}, \vec{y}) + \mathbf{d}(\vec{y}, \vec{z}). \tag{1.2}$$

Here we use Euclidean distance, but the algorithm guarantees correctness for any arbitrary distance metric for which the inequality holds true.

Elkan proves that if $\vec{x}$ is a point and $\vec{b}$ and $\vec{c}$ are centroids:

$$\text{If } \mathbf{d}(\vec{b},\vec{c}) \geq 2\mathbf{d}(\vec{x},\vec{b}), \text{ then } \mathbf{d}(\vec{x},\vec{c}) \geq \mathbf{d}(\vec{x},\vec{b}), \tag{1.3}$$

additionally,

$$\mathbf{d}(\vec{x},\vec{c}) \geq \max(0, \mathbf{d}(\vec{x},\vec{b}) - \mathbf{d}(\vec{b},\vec{c})) \tag{1.4}$$

This completes the theoretical framework enabling TI.

The basis of TI derives from the observation that many distance computations performed within k-means are redundant and can be obviated. TI achieves this by maintaining the following data structures:

- an $\mathcal{O}(n)$ data point to nearest centroid distance upper bound vector
- an $\mathcal{O}(k^2)$ centroid to centroid distance matrix
- an $\mathcal{O}(nk)$ data point to centroid lower bounds matrix

This method is extremely effective in pruning computation in real-world data, i.e. data with multiple natural clusters. It is proved that in the limit the number of distance computations when using TI is closer to n as compared with non-pruned k-means at nkj computations. The algorithm provides strong guarantees of algorithmic equivalence. TI guarantees that the centroids and cluster membership in each iteration will **exactly** match that of the non-pruned algorithm given identical initializations.

The single fundamental drawback of the algorithm for large-scale datasets is the increase in memory of $\mathcal{O}(nk)$ due to the data point to centroid lower bounds matrix. This storage requirement limits the setting in which TI is suitable to relatively small datasets.

1.3 Related Work

Today's popular graph libraries [27–29] are flexible, but lack multithreaded support and thus scalability. These represent the simplest case for application development because implementations are directly derived from algorithmic specifications. Optimizations within these libraries revolves around efficient data structure design. As such, optimizations developed here represent a small proportion of those applicable to the SEM setting. This is because developers can assume all vertices are in-memory at all times obviating I/O.

Graph frameworks like Turi [22], Giraph [30], and Mahout [13] scale through distributed processing in which datasets must fit in the aggregate memory of a cluster. Such frameworks use *vertex-centric* or *edge-centric* computation abstractions. Libraries developed within these frameworks are performance bottlenecked by network traffic. As such, optimizations focus on reducing network I/O, neglecting memory consumption and NUMA effects. Finally, such frameworks utilize process-level concurrency, obviating many of the shared-memory optimizations that are essential for SEM library acceleration.

Some out-of-core graph frameworks [31–33] focus on memory minimization, streaming datasets and thus provide scalability with minimal resources, but neglect performance. Application optimization here differs from SEM because entire datasets are streamed to memory in each algorithmic iteration. SEM instead permits $\mathcal{O}(n)$ data be held in-memory in addition to selective I/O. This leads to greater opportunity for I/O reduction and caching optimization.

Other out-of-core frameworks rely on heavy graph format preprocessing and non-commodity hardware like co-processors [34] and GPUs [35] to improve performance. Libraries developed in this space focus on minimizing device to host and host to device I/O. Furthermore, the architecture, memory hierarchy and processor density within co-processors vastly differ from that of CPUs. This leads to programming patterns that are distinct from those that accelerate SEM applications on CPUs.

SEM frameworks are the most performant of out-of-core solutions. The vast majority of which [17, 36–41], like FlashGraph [17] require only commodity hardware. The key difference is that within the SEM abstraction, vertices must now explicitly issue I/O requests for edge-related data. Once requests are fulfilled and data are in memory, *activated* vertices are processed. Graphyti is built on FlashGraph and exhibits application optimizations that are unique to SEM. The optimizations are derived from core principles that we enumerate in Chapter 2 as a blueprint for developers on which to build their own SEM applications.

Mahout [13] provides a machine learning library that combines canopy (pre-)clustering [42] alongside MM-style algorithms to cluster large-scale datasets. Mahout relies on Hadoop! an open source implementation of MapReduce [43] for parallelism and scalability. Map/reduce allows for effortless scalability and parallelism, but little flexibility in how to achieve either. As such, Mahout is subject to load imbalance in the second MM phase as this is generally an operation that can utilize fewer processors than are available for computation. This results in skew in one of the two MM phases.

MLlib is a machine learning library for Spark [44]. Spark imposes a functional paradigm to parallelism allowing for deferred computation through the use of transformations that form a lineage. The lineage is then evaluated and automatically parallelized. MLlib's performance is highly coupled with Spark's ability to efficiently parallelize computation using the generic data abstraction of the resilient distributed datasets (RDD) [45]. The in-memory data organization of RDDs does not currently account for NUMA architectures, but many of the NUMA optimizations that we develop could be applied to RDDs.

Popular machine learning libraries, such as Scikit-learn [46], ClusterR [47], and mlpack [48], support a variety of clustering algorithms. These frameworks perform computation on a single machine, often serially, without the capability to distribute computation in the cloud or perform computation on data larger than size of the machine's memory. clusternor presents a lower-level API that allows users to distribute and scale many algorithms. Once implemented, Python and R

bindings allow an algorithm to be called directly from user code.

Other works [49, 50] focus on developing serialized clustering approximations. Sophia-ML uses a mini-batch application that uses sampling to reduce the cost of Lloyd's k-means algorithm (also referred to as batched k-means) and stochastic gradient descent k-means [50]. Sophia-ML's target application is online, real-time applications. We demonstrate that clusternor can handle larger batch sizes than possible with Sophia-ML as we develop a parallel, and thus more scalable and performant mini-batch algorithm. Shindler et al [49] developed a fast approximation that addresses scalability by streaming data from disk sequentially, limiting the amount of memory necessary to iterate. This shares some similarity with the SEM capability of clusternor, but is designed for a single processor, whereas we optimize for both memory reduction and parallelism.

Euclidean distance (Section 1.2.1) defines a metric space and is commonly used in MM-style algorithms, like k-means, for computing the difference between feature-vectors. Given k clusters and a dataset $\vec{V} \in \mathbb{R}^{n \times d}$, k-means assigns a cluster , $c_i, i \in \{1...k\}$ to each data point v_i. Elkan proposes the use of the triangle inequality (TI) with bounds [18], to reduce the number of distance computations in k-means to fewer than $\mathcal{O}(kn)$ per iteration. TI determines when the distance of data point, v_i, that is assigned to a cluster, c_i, is far enough from any other cluster, $c_x, x \in \{1..k\} - i$, so that no distance computation is required between v_i and c_x. This method is extremely effective in pruning computation in real-world data, i.e. data with multiple natural clusters. The method relies on a sparse lower bound matrix of size $\mathcal{O}(nk)$. Yinyang k-means [51] develop a competitor pruning technique to TI that maintains a lower-bound matrix of size $\mathcal{O}(nt)$, in which t is a parameter and $t = k/10$ is generally optimal. Yingyang k-means outperforms TI by reducing the cost of maintenance of their lower-bound matrix. Both Yinyang k-means and TI suffer from scalability limitations because the lower-bound matrix increases in-memory state asymptotically.

Chapter 2

Graphyti

2.1 Introduction

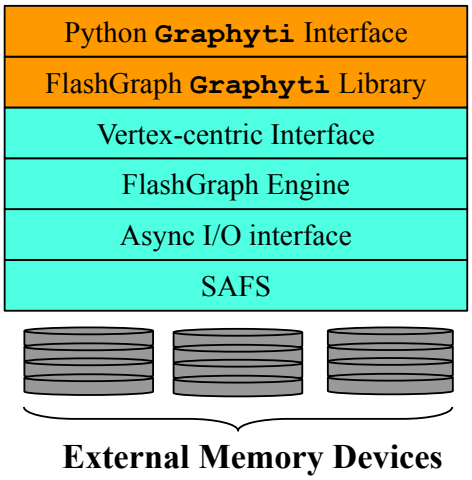
Graph datasets exceed the in-memory capacity of most standalone machines. Traditionally, graph frameworks have overcome memory limitations through scale-out, distributing computing. However, with hardware advancements in multi-core NUMA machines, and fast external memory storage devices like NVMe SSDs, framework developers strongly embraced the semi-external memory (SEM) [52] paradigm for graph analytics.

A large number of frameworks [17,36–40] aimed at reducing overhead from distributed data have adopted single-node scale-up computation. FlashGraph [17] was the first to adopt the SEM model for vertex-centric processing. In SEM $\mathcal{O}(m)$ data resides on disk and $\mathcal{O}(n)$ data in memory, for a graph with n vertices and m edges.

For developers, this adds complexity because they must explicitly encode I/O within applications. We present key principles that are critical for application developers to adopt in order to achieve state-of-the-art performance, while minimizing I/O and memory for algorithms in SEM. We present Graphyti, an extensible parallel SEM graph library built on FlashGraph and available in Python via `pip` as `graphyti`. In SEM Graphyti achieves 80% of the performance of in-memory

```
class vertex {
  // entry point (runs in memory)
  void run(engine&);
  // per vertex computation
  void run_on_vertex(engine&,vertex&);
  // process a message
  void run_on_message(engine&, msg&);
  void run_on_iteration_end(engine&);
};
```

(a) FlashGraph Programming Interface.



(b) Graphyti Architecture.

Figure 2.1: The programming interface of FlashGraph and architecture of Graphyti.

execution and retains the performance of FlashGraph, which outperforms distributed engines, such as PowerGraph [53] and Galois [24].

2.2 Architecture

Graphyti provides python bindings and a C++ library that runs on the FlashGraph engine. FlashGraph builds upon the SAFS userspace file system [20] that performs asynchronous parallel I/O from external memory devices. SAFS is distributed and installed transparently with FlashGraph. Figure 2.1 shows the C++ FlashGraph programming interface and architecture.

2.3 Principles

We present six representative algorithms that demonstrate the principles that are critical to realize state-of-the-art performance for SEM vertex-centric applications. The patterns in these algorithms serve as a blueprint for the developers of other SEM algorithms. Each subsection (2.3.1

– 2.3.6) describes an algorithm followed by the vertex-centric, SEM optimizations.

We conduct validation experiments on either the directed or undirected version of the Twitter [54] graph dataset which contains 42 Million vertices and 1.5 Billion edges of size 14 GB. All experiments require no more than 4 GB of memory of which 2 GB is used for FlashGraph's configurable page cache.

2.3.1 PageRank

The PageRank [55] algorithm identifies vertices of high importance in a graph. The algorithm assigns a higher rank to vertices referenced by other high ranking vertices as follows:

$$R(u) = c \sum_{v \in B_u} \frac{R(v)}{N_v}, \tag{2.1}$$

in which $R(x)$ is the PageRank of vertex x, B_x is the set of all inward pointing neighbors of vertex x, c is a normalization factor, and N_x is the number of outward pointing neighbors of vertex x. Traditionally, developers adopt the following algorithm for vertex-centric interfaces:

1. gather in-edge neighbor PageRanks.
2. compute a vertex's updated PageRank.
3. if the updated PageRank value surpasses a predefined threshold, multicast out-bound neighbors informing them to activate.

We refer to this as the `PR-pull` algorithm and it is utilized by both Google's Pregel [21] and Apple's Turi [22]. In the *pull* model vertices extract information from their neighbors.

When developing the application for SEM we must prioritize I/O minimization. We instead adopt a *push* (`PR-push`) model as follows:

1. compute a vertex's PageRank.

2. if a vertex's current PageRank exceeds a predefined threshold, multicast its PageRank to its out-bound neighbors.

Limit superfluous data reads: The key insight is that `PR-pull` often activates vertices and requests data for neighbors whose PageRank has already converged. `PR-push` instead computes a delta then sends messages only activating the minimal subset of vertices necessary, though possibly many times in a single iteration.

Vertex activation, processing and the superfluous I/O reads degrade the performance of `PR-pull`. Even though `PR-push` and `PR-pull` share the same upper bound of messaging complexity ($\mathcal{O}(m^2)$), on average `PR-push` sends fewer messages, reducing I/O and improving performance. Figure 2.2 demonstrates a reduction of I/O by a factor of 1.8, and improvement in runtime of 2.2. Furthermore, `PR-push` reduces I/O read requests by a factor of nearly 5. Finally, a reduction in messages leads to reduced burden on FlashGraph to load balance message queues for worker threads.

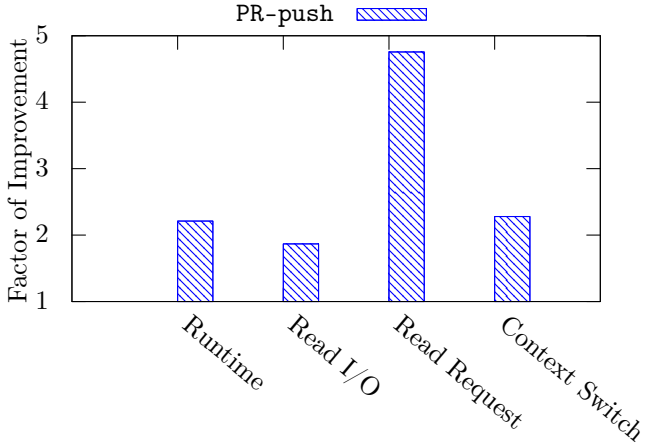


Figure 2.2: Runtime, Read I/O, I/O requests, and thread context switches of `PR-push` when compared with `PR-pull`.

2.3.2 Coreness Decomposition

Coreness decomposition extracts a maximal subgraph in which each vertex has at least degree k_{max}. The algorithm proceeds by iteratively deleting vertices (and adjacent edges), beginning

with those with degree 0 until k_{max}. Deleted vertices notify neighboring vertices to reduce their degree until only vertices with a coreness of $\geq k_{max}$ remain. The optimizations we employ to improve the performance of coreness highlight the following core principles:

Minimize messaging: Graphyti's `coreness` adopts a hybrid messaging discipline inspired by guided schedulers. Almost all vertices will need to modify their degree and inform neighbors of their deletion in early iterations in natural graphs. During this phase, multicast messages are most efficient. As the graph becomes sparser, multicast messages incur higher overhead because many neighboring vertices with lower coreness values are already deleted. At this juncture, point-to-point messages greatly reduces messaging overhead, improving runtime as shown in Figure 2.3. To facilitate this, the coreness application maintains a distribution over all remaining vertices to determine when each one should should switch to point-to-point messaging. We empirically determine that once a vertex has 10% of its original degree, point-to-point messaging improves the time necessary to process a single vertex by an order of magnitude.

Algorithmically prune computation: At the completion of a coreness iteration, k_i, in which $k_i < k_{max}$, as stated, the algorithm would proceed to k_{i+1}, k_{i+2} and so forth. Graphyti prunes unnecessary k_i values by observing the next possible core value is at least $k_{\min(deg(\alpha))} \forall \alpha \in A$, in which deg the degree of a vertex, $\alpha \subset V$ and V is the set of all vertices in the graph. This optimization alone improves performance by an order of magnitude (Figure 2.3).

2.3.3 Graph Diameter

Graph diameter is defined for connected graphs as the maximum of the all pairs shortest paths in a graph. Exact graph diameter is of computation complexity $\mathcal{O}(n^3)$ and is thus computationally challenging for any framework. As such, Graphyti computes an estimated diameter using a series of breadth-first searches from *pseudo-peripheral* vertices i.e., ones as close to the extremities of the graph as possible. Diameter estimation optimization highlights the following guiding principle:

Decouple algorithm development from framework constructs: Though simple,

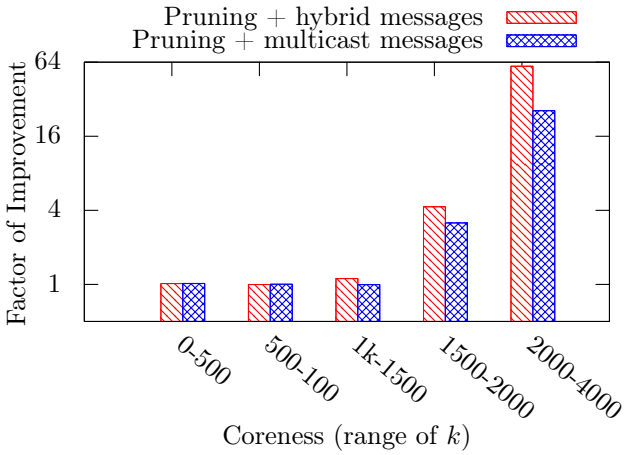


Figure 2.3: The relative factor of runtime performance improvement of Graphyti's `coreness` compared to an unoptimized implementation i.e., one with only point-to-point messages and no pruning. Pruning + hybrid messaging is 2.3X faster than pruning alone and 60X faster than unoptimized.

this algorithm provides developers with the opportunity to design a more efficient vertex-centric application. A straightforward way to perform this is to repeat the following until all reachable vertices are visited:

1. select a peripheral *source* vertex.
2. perform a parallel BFS from the selected vertex.
3. update neighboring vertex distances to one greater than their nearest neighbor in parallel.

We refer to this as *uni-source* BFS. This can be performed multiple times with different source vertices in order to attempt to find larger diameters. Though parallel, this algorithm limits the potential amount of work each vertex performs in a single BFS iteration limiting CPU cache data reuse, leading to more data stalls, and increasing the relative overhead of synchronization barriers at each BSP step. This is because uni-source BFS alone is computationally inexpensive, leading to no edge data reuse when brought into memory. This results in increased data stalls as the application becomes heavily I/O bound.

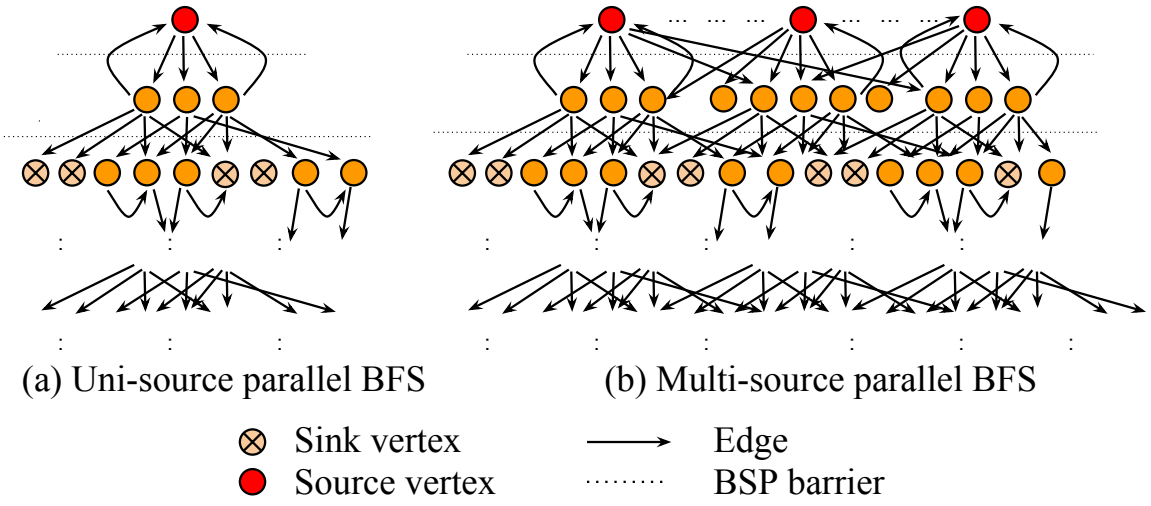
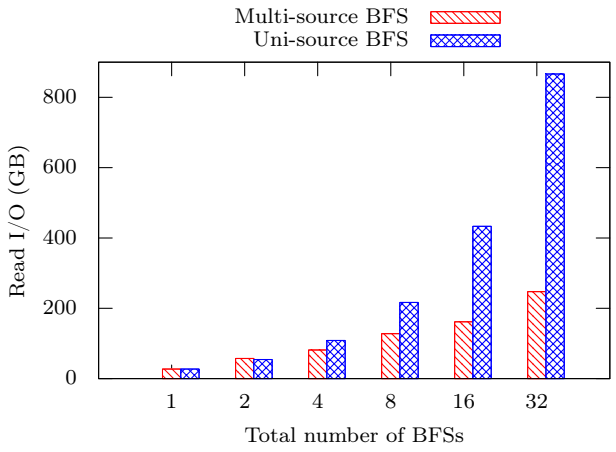


Figure 2.4: Uni-source BFS (left) is susceptible to terminal paths due to sink vertices and loops. Multi-source BFS (right) increases page cache hits by improving data reuse because multiple paths activate the same vertices in each BFS frontier.

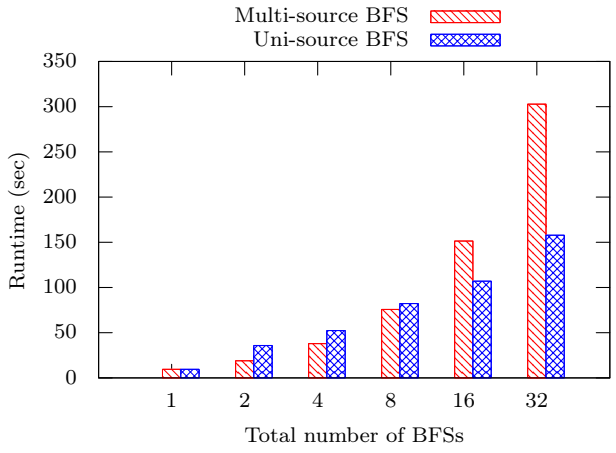
Graphyti rethinks the computation to minimize the overhead of each BSP step, by performing concurrent parallel breadth-first searches (Figure 2.4). We refer to this as *multi-source* BFS. This strategy mitigates the effect of vertices with already discovered neighbors and sink vertices, both of which result in the termination of a particular path. Additionally, this reduces cache thrashing, because requested data that are now in-memory have greater opportunity for reuse. Finally, this strategy lowers the overhead of global barriers by performing significantly more work within each iteration when compared with uni-source BFS. In multi-source BFS each vertex holds a bitmap indicating which BFS path(s) it is on and updates state appropriately. Figure 2.5b demonstrates the performance improvements and I/O reduction induced by these optimizations.

2.3.4 Betweenness Centrality (BC)

Betweenness centrality measures the importance of a vertex in a network by computing the number of shortest paths in which a vertex participates. The most efficient algorithm to compute betweenness centrality [56] is an iterative algorithm with computation complexity $O(nm + n^2 \log n)$,



(a) Quantity of data read from SSDs when performing parallel uni-source BFS compared with that of parallel multi-source BFS used in `diameter`.



(b) Runtime performance of parallel uni-source BFS compared with that of parallel multi-source BFS used in `diameter`.

Figure 2.5: I/O and Runtime performance of parallel multi-source BFS used in Graphtyi's `diameter` application, compared to performing parallel uni-source BFS.

for weighted graphs.

Betweenness centrality has three phases per iteration, (i) breadth-first search (BFS) from a source vertex (ii) backward propagation (BP), and (ii) an accumulation phase (ACC). We derive the following principles:

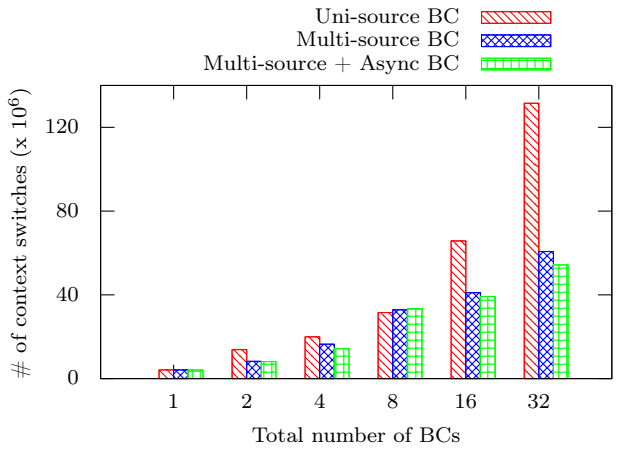
Develop applications asynchronously: Graphyti adopts a multi-source betweenness centrality strategy, similar to that of the graph diameter application. The existence of 3 phases, however, provides the opportunity for further application optimization. The observation is that developers can further improve parallel efficiency by eliminating phase synchrony for the multiple sources. Vertex activation messages now contain metadata for both the current path(s) and the current phase(s).

Graphyti's `betweenness` application separates algorithmic design from the innate BSP paradigm within all vertex-centric frameworks. Asynchronous design improves runtime by over 10% when compared with just multi-source and 40% when compared to uni-source at 32 sources. Figure 2.6. Furthermore, multi-source asynchronous betweenness centrality reduces the amount of data brought to disk by a factor of 4 when 32 concurrent searches are performed.

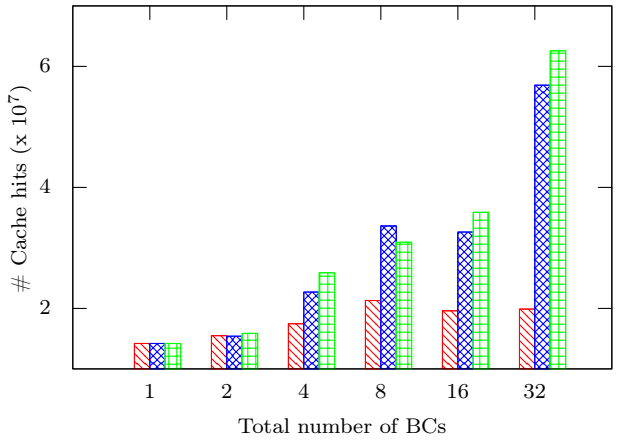
Utilize functional constructs: Vertex-centric frameworks provide abstractions over threads that are accessible to developers. Each partition thread in FlashGraph is a mechanism to represent contention-free structures. As such, associative operations such as functional reductions (e.g., `max`, `min`, `sum` etc.) are naturally supported without resource contention. The BFS phase, computes a global per-source-vertex `max`. The ACC phase, computes a global per-source-vertex `add`. Both phases utilize this optimization.

2.3.5 Triangle Counting

Triangle counting is a topological structure discovery algorithm concerned with determining the number of pairs of vertices that share a common neighbor. When performed in SEM the complexity is $\mathcal{O}(n^3)$. In SEM, the fundamental task is the comparison of neighboring vertex adjacency lists in order to determine the intersection, which constitutes the discovery of triangles. Therefore, a vertex requests neighbor adjacency lists, when each list hits the page cache the vertex performs the intersection computation. We discount alternative implementations in which the state of a vertex can exceed the size of its own edge list and that of one other neighbor because they would violate



(a) Multi-source and multi-source + async increase task sizes for vertices when active, reducing context switching.

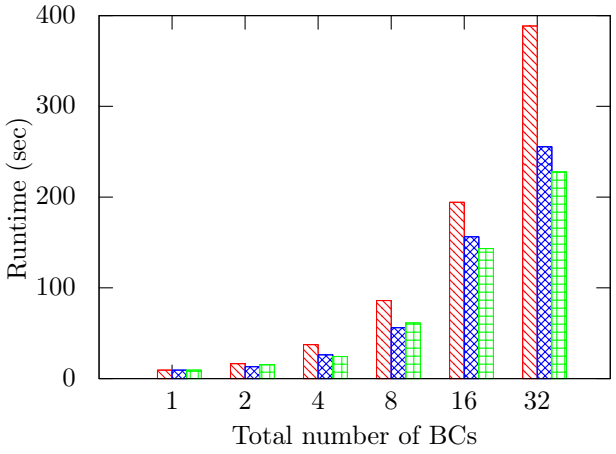


(b) Multi-source and multi-source + async increase the ratio of cache hits per accessed page.

the SEM limited memory usage guarantee.

Optimize in-memory operations: Once data has been brought into memory it is essential to not only reuse cached data, but perform in-memory optimizations. The following is done to accelerate the intersection search operation:

- Store a running vertex's adjacency list in sorted order. This enables the use of both binary search and sequential scans when appropriate.
- Store the adjacency list of a vertex with degree higher than a certain threshold in a hash table



(c) Multi-source BC and multi-source + asnyc BC outperform multiple uni-source BC runs.

Figure 2.6: Multi-source asynchronous betweenness centrality reduces context switching, improves cache utility and lowers runtime compared to multiple uni-source and multi-source BC alone.

to improve lookup performance.

- Perform a *restarted* binary search in the event an element is not found. A restarted binary search looks for the next item using the end point of the previous search. This is possible because edge lists are stored in sorted order.

- Order the adjacency list enumeration appropriately. This choice will lead to either forward or reverse traversal of edge lists being more efficient. In our case, reverse iteration leads to an improvement of $1.7X$ in search. This is because the discovery of triangles is performed by higher degree vertices leading to fewer requests for edge lists of lower degree vertices.

Figure 2.7 displays the improvement we obtain from each of the in-memory optimizations. After all optimizations are applied Graphyti's triangle counting application performs on average two orders of magnitude faster.

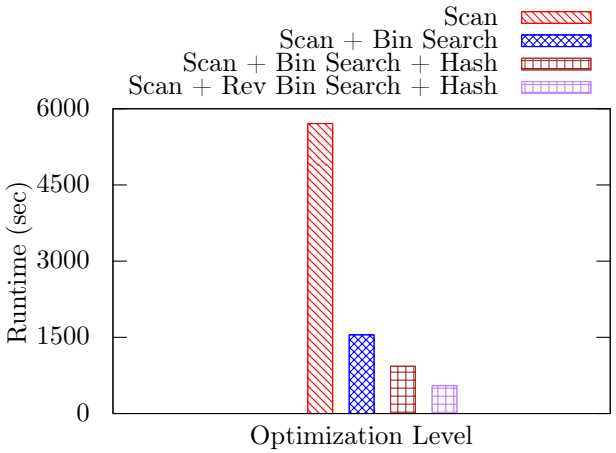


Figure 2.7: Incremental optimizations applied Graphyti's Triangle counting application. The applications is two orders of magnitude faster than a scan adjacency list intersection implementation.

2.3.6 Louvain Modularity

Louvain modularity [57] is an agglomerative community detection algorithm that aims to maximize the density of edges within communities and minimize those outside. Modularity for any pair of communities i and j is computed as follows:

$$Q = \frac{1}{2m}\sum_{ij}(A_{ij} - \frac{k_i k_j}{2m})\delta(c_i, c_j), \tag{2.2}$$

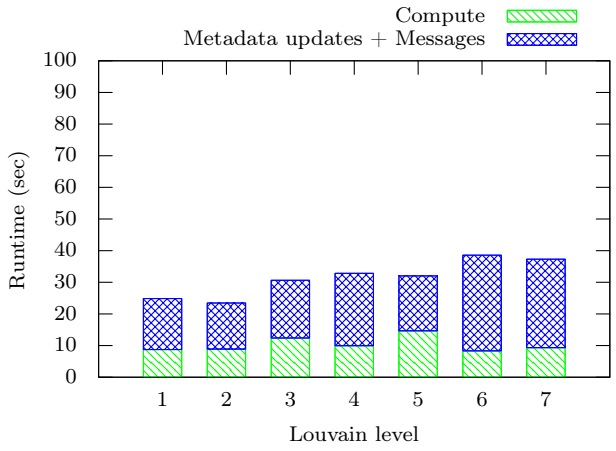
in which m is the sum of all graph edge weights, A_{ij} is the edge weight between v_i and v_j, k_i and k_j are the weighted sum of edges between v_i and v_j, δ is a function that differentiates one community from the next.

We adopt the most popular two phase, greedy algorithm [57] because exact solutions are computationally infeasible for very large networks. A vertex changes community to another that contains the maximum **positive** modularity among neighboring communities. For a vertex i, moving to community, C, the change in modularity is as follows:

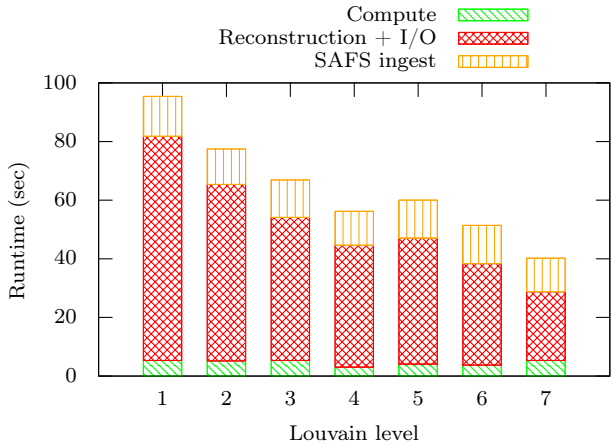
$$\Delta Q = \left[\frac{\zeta + k_{in}}{2m} - \left(\frac{\lambda + k_i}{2m}\right)^2\right] - \left[\frac{\zeta}{2m} - \left(\frac{\lambda}{2m}\right)^2 - \left(\frac{k_i}{2m}\right)^2\right], \tag{2.3}$$

in which ζ is the weight of links inside C, k_{in} is the sum of the weights of edges from i to C, and λ

is the sum of weights of edges within C.



(a) The breakdown of runtime for Graphyti's `louvain`.



(b) The breakdown of performance of Louvain when computed through progressive materializations of communities.

Figure 2.8: Graphyti's optimized `louvain` routine runs 2 X faster than a traditional one with physical graph modifications.

This algorithm poses challenges for SEM frameworks because graph modification is typically extremely expensive, and often not supported. To overcome this, we encourage developers to adopt the following principles:

Avoid graph structure modification: Edge data are on disk, thus modifying the graph

is prohibitively expensive. Furthermore, write amplification within SSDs causes their degradation and premature failure if writes are performed frequently. With SEM applications, modification can easily surpass the algorithmic runtime due to disk write throughput typically being orders of magnitude slower than memory throughput. We demonstrate this in Figure 2.8b. Accordingly, we circumvent modification through (i) lazy deletion and (ii) vertex nomination of a *community representative* vertex. We achieve this by maintaining a partitioned bitmap with lookups for deleted vertices in addition to an index for vertex-to-community lookups. This ensures all messages are appropriately routed to the correct vertex without involving the graph engine or requiring messages to be forwarded.

Figure 2.8b represents the "best-case scenario" for an SEM implementation that physically modifies the graph. We maintain a RAMDisk in fast DDR4 to hold the new physical state of the graph prior to striping edge data across disks during the *SAFS ingest* procedure. Despite this, we observe Graphyti's `louvain` will still perform twice as fast (Figure 2.8a). We trade-off graph structure modification with metadata updates and messaging. Naturally, as the algorithm progresses to deeper levels, more vertices merge, resulting in fewer clusters. This reduces the cost of traditional graph modification, while conversely increasing the overhead of messaging and metadata maintenance for Graphyti's `louvain`. Accordingly, Graphyti's `louvain` design capitalizes most during early levels to attain its performance gains.

2.4 Software

Graphyti is an open source library available through Python's `pip` package manager under the name `graphyti`. To extend the library, developers can visit https://github.com/flashxio/Graphyti. Furthermore, we provide Docker integration for developers to reduce the barrier to entry.

2.5 Conclusions

We present key principles identified as critical for state-of-the-art application performance for vertex-centric semi-external-memory graph algorithms. Through illustrative examples within Graphyti we demonstrate the positive performance effects of adoption of these principles. At the core of the applications and principles are novel advancements in fine-grain I/O management for graph analytics.

The optimizations we develop are NUMA sensitive. Because FlashGraph provides NUMA sensitive partitioning and vertex scheduling for graphs, we develop our optimizations to capitalize on higher I/O throughput rates and lower latency from NUMA local access. As such, optimizations we develop that improve cache reuse, eliminate cache thrashing and enable sequential access leading to compounded gains in improvement in performance due to circumventing negative NUMA effects. The themes of NUMA sensitivity, efficient caching, scheduling and computation pruning are recurrent within this thesis and are revisited from the perspective of divisive community detection within Chapters 3 and 4.

This work advances the knowledge of SEM application developers while providing a scalable, open source, extensible tool. Graphyti's final contribution is the improvement of accessibility of SEM graph applications to users by providing a high level Python interface. Throughout this thesis we continually provide scalable utilitarian tools and packages that leverage the fine-grain NUMA sensitive I/O optimizations similar to the `graphyti` package.

Chapter 3

knor: k-means Algorithmic and Computation Advancements for Multicore NUMA Machines

3.1 Introduction

K-means is one of the most influential and utilized unsupervised machine learning algorithms. Its computation limits the performance and scalability of many statistical analysis and machine learning tasks. With the popularity of deep neural networks soaring, k-means remains relevant as a critical component within unsupervised deep learning [58, 59]. K-means enables the fast computation of (approximate) nearest neighbor search [60], representation learning [61], computer vision [62]i, dimensionality reduction and manifold learning [63]. Finally, k-means has two desirable properties that generalize the optimization strategies we develop to other relevant algorithms:

1. k-means forms the basis upon which many popular clustering algorithms [50, 64–69] are built.

2. The Majorize-Minimization or Minorize-Maximization (MM) two-step patten is common to many popular machine learning algorithms [26, 70–72].

We rethink and optimize k-means in terms of modern NUMA architectures. We develop a novel parallelization scheme that delays synchronization barriers, and minimizes superfluous computations while maintaining practicality. We detail algorithmic contributions and later demonstrate their capacity to enable state-of-the art performance for k-means in all memory settings. We then demonstrate that when combined with framework optimizations, our k-means application outperforms distributed commercial products like H_2O, Apple's Turi (formerly GraphLab) and Spark's MLlib, by more than an order of magnitude for datasets of 10^7 to 10^9 points.

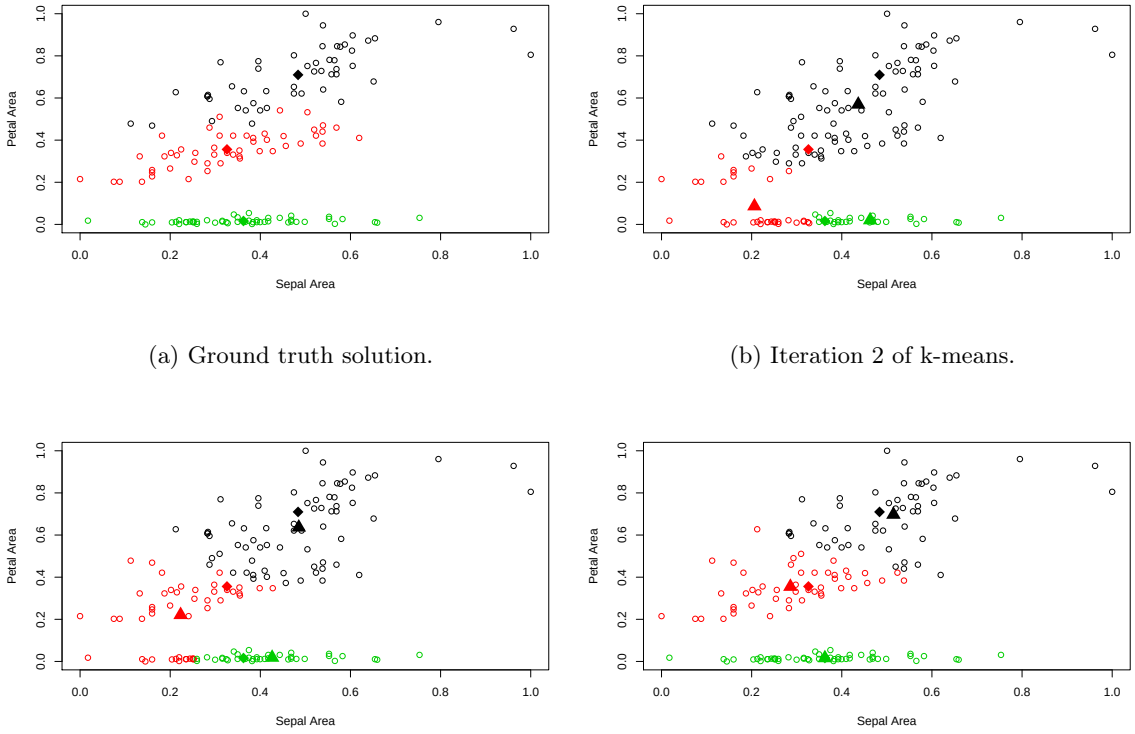
3.2 Algorithmic advancements

We develop algorithmic optimizations applicable to k-means and by extension other algorithms embodying the MM computation paradigm. To achieve state-of-the-art performance on multi-core NUMA machines, we maximize parallel processing (Section 3.2.1) and minimize superfluous computations while maintaining practical storage bounds (Section 3.2.2).

3.2.1 Barrier Minimization

We minimize synchronization barriers for algorithms in which (all or parts of) the two M-steps can be performed simultaneously. We maintain per-thread data structures and compute partial-aggregations that are finalized in a parallel reduction operation at the end of the computation. All algorithms that utilize k-means have this property. Our implementation modifies the most popular synchronous algorithm for k-means, Lloyd's algorithm [25]. The result is a parallelized, barrier-minimized and NUMA-aware algorithm we refer to as "||Lloyd's".

||Lloyd's reduces factors limiting parallelism in a naïve parallel Lloyd's algorithm. Traditionally, Lloyd's operates in two-phases each separated by a global barrier as follows:



(a) Ground truth solution.

(b) Iteration 2 of k-means.

(c) Iteration 4 of k-means.

(d) Iteration 8 of k-means.

Figure 3.1: The k-means algorithm on the normalized petal and sepal areas of the 3 classes of flowers in the *iris* dataset. Each class contains 50 samples. K-means converges with over 90% accuracy within 8 iterations.

1. Phase I: Compute the nearest centroid, $\vec{\mu}^t$ to each data point, $\vec{v}$, at iteration t.

2. Global barrier.

3. Phase II: Update each centroid, for the next iteration, $\vec{c}^{t+1}$ to be the mean value of all points nearest to it in Phase I.

4. Global barrier.

5. Repeat until converged.

Naïve Lloyd's uses two major data structures; A read-only global centroids structure, $\vec{c}^t$, and a shared global centroids for the next iteration, $\vec{c}^{t+1}$. Parallelism in Phase II is limited to k threads because $\vec{c}^{t+1}$ is shared. As such, Phase II is plagued with substantial locking overhead because of the high likelihood of data points concurrently attempting to update the the same nearest centroid. Consequently, as n gets larger with respect to k this interference worsens, further degrading performance.

||Lloyd's retains the read-only global centroid structure $\vec{c}^t$, but provides each thread with its own local copy of the next iteration's centroids. Thus we create T copies of $\vec{c}^{t+1}$. Doing so means ||Lloyd's merges Phase I and II into a super-phase and delays the barrier (Step 3 above). The super-phase concurrently computes the nearest centroid to each point and updates a local version of the centroids to be used in the following iteration. These local centroids can then be merged in parallel through a reduction operation at the end of the iteration. ||Lloyd's trades-off increased parallelism for a slightly higher memory consumption by a factor of $\mathcal{O}(T)$ over Lloyd's. This algorithm design naturally leads to lock-free routines that require fewer synchronization barriers as we show in Algorithm 1.

3.2.2 Minimal Triangle Inequality (MTI) Pruning

We relax the constraints of Elkan's Algorithm for triangle inequality pruning (TI) [18] by removing the the need for the lower bound matrix of size $\mathcal{O}(nk)$. Omitting the lower bound matrix means we forego the opportunity to prune certain computations. We accept this tradeoff in order to limit memory consumption. Section 3.2.2 empirically demonstrates on real-world data that: (1) MTI pruning efficacy is comparable to that of TI and (2) as the number of clusters, k, increases, the performance of MTI approaches that of TI while using a fraction of TI's memory. MTI prunes an

Algorithm 1 || Lloyd's algorithm

1: **procedure** ||MEANS($\vec{V}$, $\vec{C^t}$, k)
2: $pt\vec{C}^t$ ▷ Per-thread centroids
3: $clusterAs\vec{s}ignment^t$ ▷ Shared, no conflict
4: tid ▷ Current thread ID
5: **parfor** $\vec{v} \in \vec{V}$ **do**
6: $dist = \infty$
7: $\vec{\mu}^t$ = INVALID
8: **for** $\vec{c}^t \in \vec{C}^t$ **do**
9: **if** $\mathbf{d}(\vec{v}, \vec{c^t}) < dist$ **then**
10: $dist = \mathbf{d}(\vec{v}, \vec{c^t})$
11: $\vec{\mu}^t = \vec{c}^t$
12: **end if**
13: **end for**
14: $pt\vec{C}^t[tid][\vec{\mu}^t]$ += $\vec{v}$
15: **end parfor**
16: clusterMeans = MERGEPTSTRUCTS($pt\vec{C}^t$)
17: **end procedure**

18: **procedure** MERGEPTSTRUCTS($\vec{vectors}$)
19: **while** $|\vec{vectors}| > 1$ **do**
20: PAR_MERGE($\vec{vectors}$) ▷ $\mathcal{O}(Tlogn)$
21: **end while**
22: **return** vectors[0]
23: **end procedure**

average of 84% of distance computations pruned by TI, with an average reduction in performance of only 15%.The drastic memory reduction achieved by MTI far outweighs the minor performance loss. MTI makes pruning tractable for datasets that were previously intractable using TI in which the lower bound matrix quickly consumes more memory than the data, specifically when $k > d$. With $\mathcal{O}(n)$ memory, we implement three of the five [18] pruning clauses in an iteration of k-means using MTI. Let $u^t = \mathbf{d}(\vec{v}, \vec{\mu}^t) + f(\vec{\mu}^t)$, be the upper bound of the distance of a sample, $\vec{v}$, in iteration t from its assigned cluster $\vec{\mu}^t$. Finally, we define U to be an update function such that $U(u^t)$ fully tightens the upper bound of u^t.

Clause 1: if $u^t \leq \min \mathbf{d}(\vec{\mu}^t, \vec{c}^t \, \forall \, \vec{c}^t \in \vec{C}^t)$, then $\vec{v}$ remains in the same cluster for the current iteration. For semi-external memory, this is extremely significant because no I/O request is made for data.

Clause 2: if $u^t \leq \mathbf{d}(\vec{\mu}^t, \vec{c}^t \, \forall \, \vec{c}^t \in \vec{C}^t)$, then the distance computation between data point $\vec{v}$ and centroid $\vec{c}^t$ is pruned.

Clause 3: if $U(u^t) \leq \mathbf{d}(\vec{\mu}^t, \vec{c}^t \, \forall \, \vec{c}^t \in \vec{C}^t)$, then the distance computation between data point $\vec{v}$ and centroid $\vec{c}^t$ is pruned.

MTI vs. TI pruning

We empirically determine the efficacy of our Minimal Triangle Inequality algorithm in comparison to Elkan's Triangle Inequality with bounds algorithm on the k-means application. Figure 3.2 presents our findings on Friendster-32, a real-world dataset derived from a natural graph that follows a power-law distribution in connectivity. This dataset is representative of many real-world datasets studied today.

Figure 3.2 demonstrates that MTI is comparable to TI in computation pruning capacity. MTI is within 15% of the pruning ability of TI. Furthermore, Figure 3.2 shows that as the number of clusters increase, MTI performance rapidly approaches that of TI. Finally, Figure 3.2 highlights

MTI's constant memory consumption with respect to the number of clusters. We contrast this with TI in which memory consumption grows proportionally with the number of clusters, k, making it infeasible for many practical applications. Finally, the cost of storage and index lookups for TI adversely affects its runtime, especially as k increases, making it unsuitable for large-scale applications.

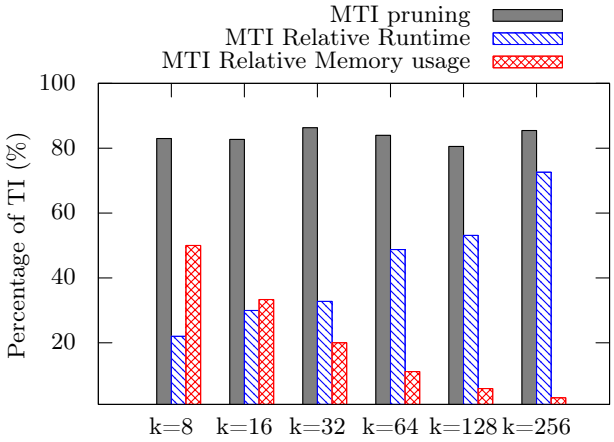


Figure 3.2: Comparison of the pruning efficacy, memory consumption and runtime performance of MTI vs. TI on the Friendster-32 dataset using k-means.

3.3 Software

The advancements to k-means developed in Chapter 3 are further generalized in Chapter in 4. We begin by developing the k-means application and publicly release it as a standalone application for in-memory, semi-external memory, and distributed processing of k-means. We are an open source project available at https://github.com/flashxio/knor. The in-memory capabilities are provided transparently to users on the Python package manager `pip` and the R programming language manager `CRAN` under then name `knor` (k-means NUMA Optimized Routines).

3.4 Conclusion

We rethinking Lloyd's algorithm for modern multiprocessor NUMA architectures through memory partitioned, conflict free data structures and the delay and minimization of critical regions. We formulate a minimal triangle inequality pruning technique (MTI) that is a relaxation of the Elkan's triangle inequality with bounds algorithm. MTI's pruning capacity is the first **practical** approach at large-scale computation pruning for k-means. In addition to inheriting all the theoretical guarantees of Elkan's triangle inequality, MTI also respects semi-external memory resource consumption bounds. This property enables the acceleration of the knor library and clusternor framework, described in Chapter 4.

Chapter 4

clusternor: A NUMA-Optimized Clustering Framework

4.1 Introduction

We build upon the algorithmic advancements described in (Chapter 3) and generalize the computation model to all MM clustering algorithms through clusternor. We rethink the parallelization of clustering for modern non-uniform memory architectures (NUMA) to maximize independent, asynchronous computation. We defer barriers, reduce remote memory accesses, and maximize cache reuse.

Clustering algorithms are iterative and have complex data access patterns that result in many small random memory accesses. We recognize the performance of parallel implementations suffer from synchronous barriers for each iteration and skewed workloads. To address these shortcomings, we present the *Clustering NUMA Optimized Routines* (clusternor) extensible parallel framework that provides algorithmic building blocks. The system is generic, we demonstrate nine modern clustering algorithms that have simple implementations. clusternor includes (i) in-memory, (ii) semi-

external memory, and (iii) distributed memory execution, enabling computation for varying memory and hardware budgets. clusternor provides a uniform programming interface with facilities for hierarchical, non-hierarchical, and linear algebraic classes of algorithms.

4.2 Applications

To motivate design decisions we select nine popular clustering algorithms that exhibit hierarchical, non-hierarchical and linear algebraic formulations for evaluation. We then implement these algorithms to demonstrate the utility, extensibility and performance of clusternor. Finally, we provide them as an open source library. We describe the algorithms below.

4.2.1 k-means

A detailed description exists in Section 1.2.5. An iterative partitioning algorithm in which data, $\vec{V}$, are assigned to one of k clusters based on the Euclidean distance, $\mathbf{d}$, from each of the cluster means $\vec{c}^t \in \vec{C}^t$. A serial implementation requires memory of $\mathcal{O}(nd + kd)$. The computation complexity of k-means both serially and parallelized within clusternor remains $\mathcal{O}(knd)$. The asymptotic memory consumption of k-means within clusternor is $\mathcal{O}(nd + Tkd + n + k^2)$. The term T arises from the per-thread centroids we maintain. Likewise, the $\mathcal{O}(n + k^2)$ terms allow us to maintain a centroid-to-centroid distance matrix and a point-to-centroid upper bound distance vector of size $\mathcal{O}(n)$ that we use for computation pruning as described in Section 3.2.2. For SEM, the computation complexity remains unchanged, but the asymptotic memory consumption drops to $\mathcal{O}(n + Tkd)$. k-means minimizes the residual sum of squares objective function for each data point, $\vec{v}$:

$$RSS_k = \sum_{\vec{v} \in \vec{V}} |\vec{v} - \vec{\mu}|^2, \tag{4.1}$$

the selection of the nearest centroid to a data point is computed as the minimum euclidean distance, $\mathbf{d}$, of all clusters. This is computed as follows:

$$min \sum_{\vec{v} \in \vec{V}} ||\mathbf{d}(\vec{v}, \vec{c}^t)|| \tag{4.2}$$

We utilize k-means as the baseline algorithm for optimization. We empirically demonstrate that its optimization directly results in the optimization of other algorithms. As such, we utilize the majority of Section 4.8 to demonstrate the performance of k-means, before turning to other algorithms.

4.2.2 Spherical k-means (sk-means)

Spherical k-means (sk-means) [67] projects all data points, $\vec{V}$, to the unit sphere prior to performing the k-means algorithm. Unlike k-means, spherical k-means uses the cosine distance function, $\mathbf{d}_{cos} = \frac{\vec{V} \cdot \vec{C}^t}{||\vec{V}||||\vec{C}^t||}$, to determine data point to centroid proximity.

4.2.3 k-means++

We develop a standalone k-means++ [64] stochastic clustering algorithm that performs multiple runs, r, of the k-means++ algorithm then selects the best run. The best run corresponds to the run that produces the minimum RSS. The k-means++ algorithm shares both the memory and computational complexity of k-means, but k-means++ chooses each new centroid $\vec{c}^t$ from the dataset through a weighted random selection such that:

$$\vec{C} \leftarrow \frac{D(\vec{v})^2}{\sum_{\vec{v} \in \vec{V}} D(\vec{v})^2}, \tag{4.3}$$

in which $D(\vec{v})$ is the minimum distance of a datapoint to the clusters already chosen.

4.2.4 Mini-batch k-means (mbk-means)

Lloyd's algorithm is often referred to as batched k-means because all data points are evaluated in every iteration. Mini-batch k-means (mbk-means) [50] incorporates random sampling into

each iteration of k-means thus reducing the memory cost of each iteration by a factor of B, the batch size, to $\mathcal{O}(\frac{nkd}{B})$ per iteration. Furthermore a parameter $\eta = \frac{1}{\vec{C}^t}$ is computed per centroid to determine the learning rate and convergence. Batching does not affect the memory requirements of k-means when run in-memory. In the SEM setting, the memory requirement is $\mathcal{O}(\frac{knd}{B})$, reducing by a factor of B. Finally, the update function is as follows:

$$\vec{C}^t \leftarrow (1-\eta)C^{t-1} + \eta\vec{V} \tag{4.4}$$

4.2.5 Fuzzy C-means (fc-means)

Fuzzy C-means (fcm) [71] is an iterative 'soft' clustering algorithm in which data points can belong to multiple clusters by computing a degree of association with each centroid. A fuzziness index, z, is a hyper-parameter used to control the degree of fuzziness. Similar to k-means, the computation complexity in the serial case is $\mathcal{O}(knd)$ per iteration, thus has the same asymptotic complexity of $\mathcal{O}(nd+Tkd+n+k^2)$ when parallelized within the framework. Fuzzy C-means computes $J \in \mathbb{R}^{n\mathbf{x}k}$:

$$J = \sum_{i=1}^{|N|}\sum_{k=1}^{|C|} u_{ik}^z ||\vec{v}_i - \vec{c}_j||^2, 1 \leq z < \inf, \tag{4.5}$$

in which u_{ik} is the degree of membership if $\vec{v}_i$ in cluster k.

4.2.6 k-medoids

K-medoids is a clustering algorithm that uses data point feature-vectors as cluster representatives (medoids), instead of centroids like k-means. In each iteration, each cluster determines whether to choose another cluster member as the medoid. This is commonly referred to as the *swap* step and is NP-hard, with complexity $\mathcal{O}(n^2d)$. This is followed by an MM step to determine cluster assignment for each data point given the updated medoids, resulting in a complexity of $\mathcal{O}(n-k)^2$.

We reduce the computation cost by implementing a sampled variant called (CLARA) [72] that is more practical, but still has a high asymptotic complexity of $\mathcal{O}(k^3 + nk)$.

4.2.7 Hierarchical k-means (H-means)

We implement a divisive version of k-means using clusternor's hiearchical interface. All data points begin in the same cluster and are partitioned recursively into two splits of their original cluster in each iteration until convergence is reached. The computation complexity is $\mathcal{O}(\frac{nd+Tkd+n+4}{B})$, in which the factor 4 is derived from the fact that we perform k-means with $k = 2$ centroids for each partition/cluster.

4.2.8 X-means

X-means [68] is a form of divisive hierarchical clustering in which the number of clusters is not provided a priori. Instead, X-means determines whether or not a cluster should be split using Bayesian Information Criterion (BIC) [73]. Computationally, it differs from H-means (Section 4.2.7) by an additional $\mathcal{O}(kn)$ step in which a decision is taken on whether or not to split after cluster membership is accumulated. We build X-means on clusternor's hiearchical interface.

4.2.9 Gaussian Means (G-means)

G-means is built on clusternor's hiearchical interface and is identical to X-means in its computation complexity and in that it does not require the number of clusters k as an argument. G-means mostly varies from X-means in that it uses the Anderson-Darling statistic [74] as the test to decide splits. The Anderson-Darling statistic performs roughly four times more computations than BIC, despite having the same asymptotic complexity.

4.3 In-memory design

We prioritize practical performance when we implement in-memory optimizations. We make design tradeoffs to balance the opposing forces of minimizing memory usage and maximizing CPU cycles spent on parallel computing.

Prioritize data locality for NUMA: As discussed in Section 1.2.2, NUMA, architectures are characterized by groups of processors that have affinity to a local memory bank via a shared local bus. Other non-local memory banks must be accessed through a globally shared NUMAlink interconnect. The result is low latency accesses with high throughput to local memory banks, and higher latency and lower throughput for remote memory accesses to non-local memory.

To minimize remote memory accesses, we bind every thread to a single NUMA node, equally partition the dataset across NUMA nodes, and sequentially allocate data structures to the local NUMA node's memory. Every thread works independently. Threads only communicate or share data to aggregate per-thread state as required by the algorithm. Figure 4.1 shows the data allocation and access scheme we employ. We bind threads to NUMA nodes rather than specific CPU cores because the latter is too restrictive to the OS scheduler. CPU thread-binding may cause performance degradation if the number of worker threads exceeds the number of physical cores.

Customized scheduling and work stealing: clusternor customizes scheduling for algorithm-specific computation patterns. For example, Fuzzy C-means 4.2.5 assigns equal work to each thread at all times, meaning it would not benefit from dynamic scheduling and load balancing via work stealing. As such, Fuzzy C-means invokes static scheduling. Conversely, k-means when utilizing MTI pruning would result in heavy skew without dynamic scheduling and thread-level work stealing.

For dynamic scheduling, we develop a NUMA-aware partitioned priority task queue (Figure 4.6) to feed worker threads, prioritizing tasks that maximize local memory access and, consequently, limit remote memory accesses. The task queue enables idle threads to *steal* work from threads bound to the same NUMA node first, minimizing remote memory accesses. The queue is partitioned into T

NUMA Node$_0$	CPU 0	Thread 0	data[0] … data[α]
	CPU 1	Thread (α)+1	data[βα] … data[(β+1)α]
	:	:	:
	Core γ-1	Thread (N-1)*T/N	data[(β-1)γα] … data[γβα]
:	:	:	:
:	:	:	:
NUMA Node$_1$	Core P-γ	Thread β-1	data[(β-1)α] … data[βα]
	Core P-(γ+1)	Thread 2β-1	data[(2β-1)α] … data[2βα]
	:	:	:
	Core P-1	Thread T-1	data[(T-1)α] … data[Tα]

Figure 4.1: The memory allocation and thread assignment scheme we utilize in memory on a single machine or in the distributed setting. $\alpha = n/T$ is the amount of data per thread, $\beta = T/N$ is the number of threads per NUMA node, and $\gamma = P/N$ is the number of physical processors per NUMA node. Distributing memory across NUMA nodes maximizes memory throughput while binding threads to NUMA nodes reduces remote memory accesses.

parts, each with a lock required for access. We allow a thread to cycle through the task queue once looking for high priority tasks before settling on another, possibly lower priority task. This tradeoff avoids starvation and ensures threads are idle for negligible periods of time. The result is good load balancing in addition to optimized memory access patterns.

Avoid interference and defer barriers: Whenever possible, per-thread data structures maintain mutable state. This avoids write-conflicts and obviates locking. Per-thread data are merged using a parallel reduction operator, much like funnel-sort [75], when algorithms reach the end of an iteration or the whole computation. For instance, in k-means, per-thread local centroids contain running totals of their membership until an iteration ends when they are finalized through a reduction.

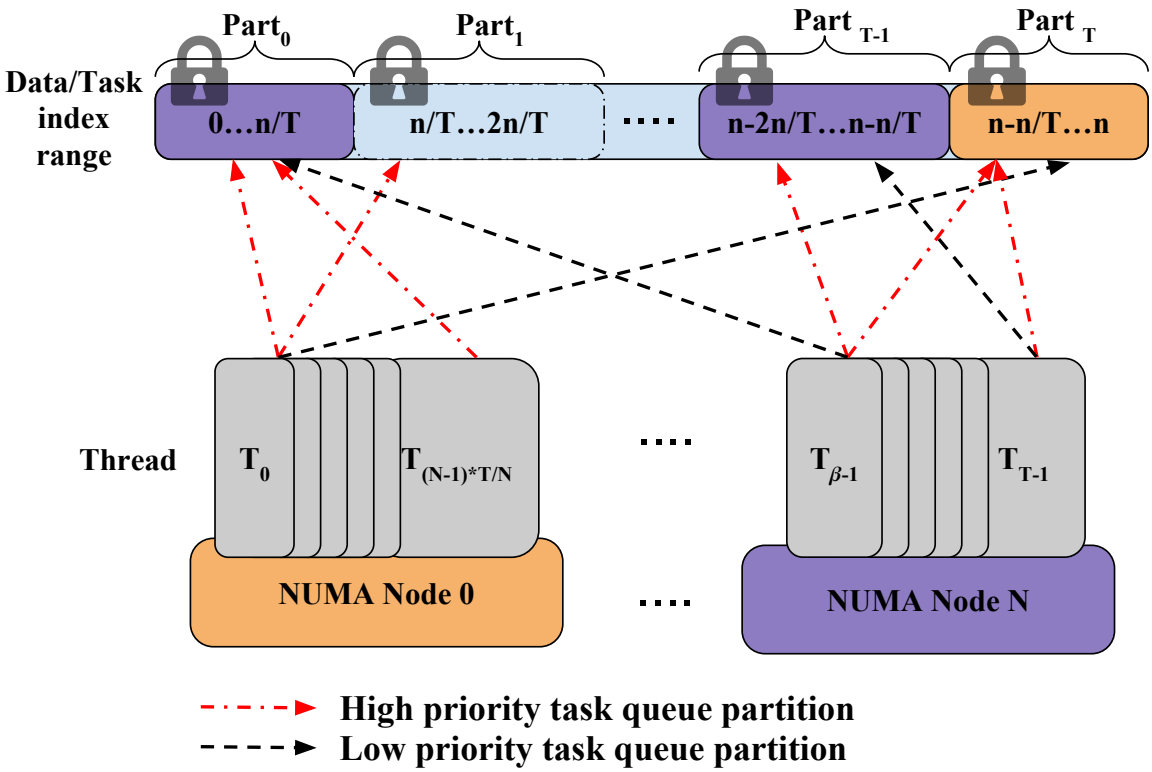


Figure 4.2: The NUMA-aware partitioned task scheduler. The scheduler minimizes task queue lock contention and remote memory accesses by prioritizing tasks with data in the local NUMA memory bank.

Effective data layout for CPU cache exploitation and cache blocking: Both per-thread and global data structures are placed in contiguously allocated chunks of memory. Contiguous data organization and sequential access patterns improve processor prefetching and cache line utilization. Furthermore, we optimize access to both input and output data structures to improve performance. In the case of a dot product operation (Figure 4.3), we access input data sequentially from local NUMA memory and write the output structure using a cache blocked scheme for higher throughput reads and writes. The size of the block is determined based on L1 and L2 cache specifications reported by the processor on a machine. We utilize this optimization in Fuzzy C-means.

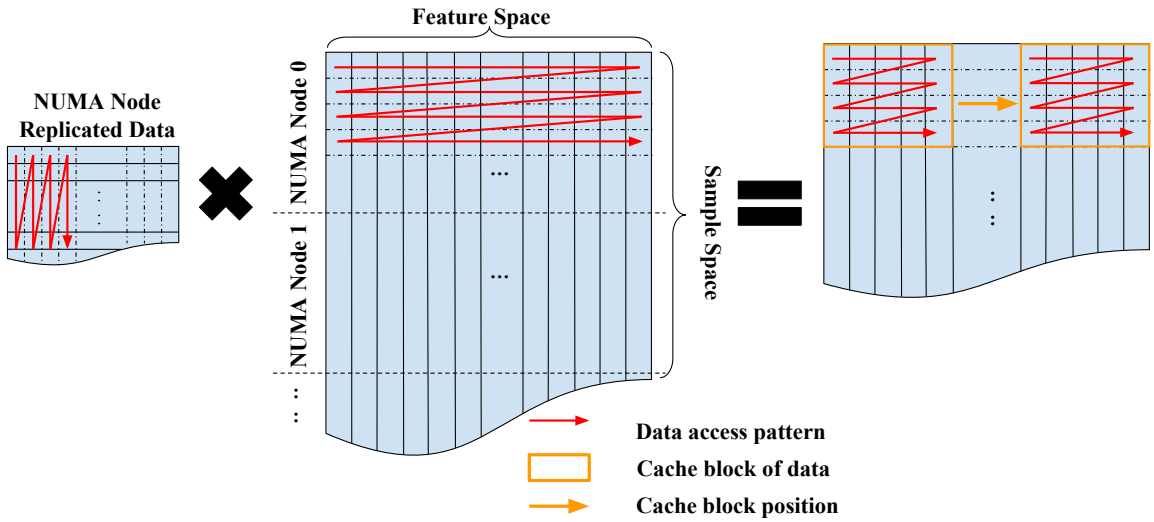
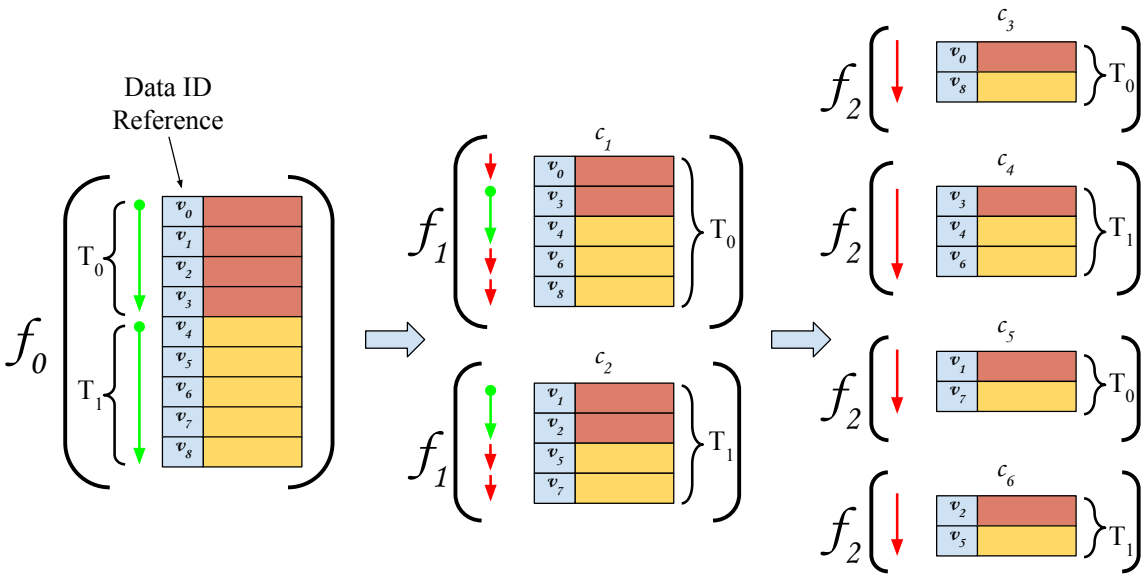


Figure 4.3: Data access patterns support NUMA locality, utilize prefetched data well and optimize cache reuse through a cache blocking scheme.

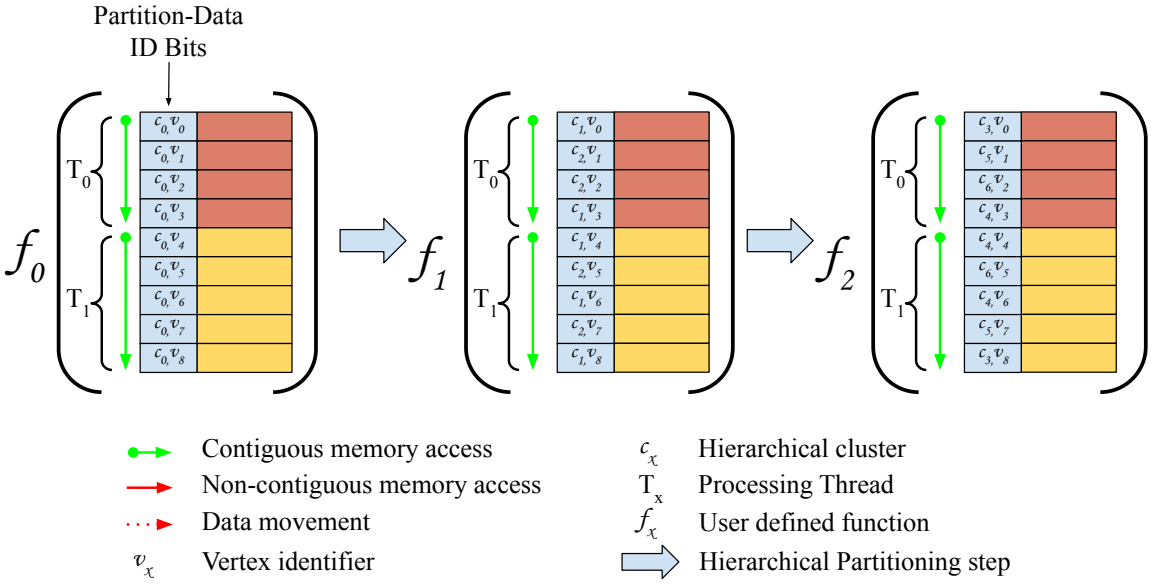
4.4 Hierarchical design

clusternor rethinks computation and data access patterns for traditionally recursive algorithms for the multicore NUMA setting. clusternor supports hierarchical clustering in which applications are written iteratively rather than recursively. Naïve implementations assign a thread to each cluster and shuffle data between levels of the hierarchy (Figure 4.4a). This incurs a great deal of remote memory access and non-contiguous I/O for each thread. clusternor avoids these pitfalls by not shuffling data. Instead, threads are assigned to contiguous regions of memory. Figure 4.4b shows the computation hierarchy in a simple two thread computation. This results in entirely local and sequential data access, which enhances prefetching.

Data movement is eliminated at the cost of an increase in managed state during clustering. We maintain a data-point to *partition-identifier* structure. The structure maps each data point to a specific partition that contains cluster labels that are eventually assumed by the data point. This design eliminates recursive calls, stack creation overhead during recursion, data movement and random data accesses.



(a) Naïve recursive parallel hierarchical clustering exhibits poor data locality, and non-contiguous data access patterns.



(b) clusternor transparently provides NUMA-local, sequential, and contiguous data access patterns.

Figure 4.4: A naïve hierarchical implementation with unfavorable data access patterns compared to clusternor. clusternor enforces sequential data access, naturally load-balances and maximizes use of cache data.

4.5 Semi-external Memory Design

We design a highly-optimized, semi-external memory module that targets **scale-up** computing on multi-core NUMA machines, rather than distributed computing. With SEM, we scale to problem instances that exceed the memory size of the machine and typically find that single-node systems are much faster than distributed systems that use an order of magnitude more hardware. We realize single-node scalability by placing data on SSDs and performing asynchronous I/O requests for data as necessary while overlapping computation. The SEM model allows us to reduce the asymptotic memory bounds. A SEM routine uses $\mathcal{O}(n)$ memory for a dataset, $\vec{V} \in \mathbb{R}^{n \mathtt{x} d}$ that when processed completely in memory would require $\mathcal{O}(nd)$ memory.

4.5.1 FlashGraph Modifications

Our implementation modifies the FlashGraph system to support matrix-like computations. FlashGraph's primitive data type is the `page_vertex` that is interpreted as a vertex with an index to the edge list of the `page_vertex` on SSDs. We define a *row* of data to be equivalent to a d-dimension data point, $\vec{v}_i$. Each row is composed of a unique identifier, *row-ID*, and d-dimension data vector, *row-data*. We add a `page_row` data type to FlashGraph and modify the asynchronous I/O layer to support floating point row-data reads rather than the numeric identifiers for graph edge lists. The `page_row` type computes its row-ID and row-data location on disk meaning only user-defined state is stored in-memory. The `page_row` reduces the memory necessary to use FlashGraph by $\mathcal{O}(n)$ because it does not store an index to data on SSDs unlike a `page_vertex`. This allows our SEM applications to scale to larger datasets than possible before on a single machine.

4.5.2 I/O minimization

I/O bounds the performance of most well-optimized SEM applications. Accordingly, we reduce the number of data-rows that need to be brought into memory each iteration. In the case of

k-means, only Clause 1 of MTI (Section 3.2.2) facilitates the skipping of all distance computations for a data point. Likewise for mini-batch k-means and k-medoids that subsample the data, we need not read all data points from disk in every iteration. We observe the same phenomenon when data points have converged in a cluster for H-means, G-means and X-means as well. In these cases, we do not issue I/O requests but still retrieve significantly more data than necessary from SSDs because pruning occurs near-randomly and sampling pseudo-randomly. Reducing the filesystem *page size*, i.e. minimum read size from SSDs alleviates this to an extent, but a small page size can lead to a higher number of I/O requests, offsetting any gains achieved from reduced fragmentation. We utilize a minimum read size of 4KB. Even with this small value, we receive much more data from disk than we request. To address this, we develop an optionally lazily-updated partitioned *row cache* that drastically reduces the amount of data brought into memory.

Partitioned Row Cache (RC)

We add a layer to the memory hierarchy for SEM applications by designing an optionally lazily-updated row cache (Figure 4.5). The row cache improves performance by reducing I/O and minimizing I/O request merging and page caching overhead in FlashGraph. A row is *active* when it performs an I/O request in the current iteration for row-data on disk. The row cache pins active rows to memory at the granularity of a row, rather than a page, improving its effectiveness in reducing I/O compared to a page cache.

We partition the row cache into as many partitions as FlashGraph creates for the underlying matrix, generally equal to the number of threads of execution. Each partition is updated locally in a lock-free caching structure. This vastly reduces the cache maintenance overhead, keeping the RC lightweight. The size of the cache is user-defined, but 1GB is sufficient to significantly improve the performance of billion-point datasets.

The row cache operates in one of two modes based upon the data access properties of the algorithm:

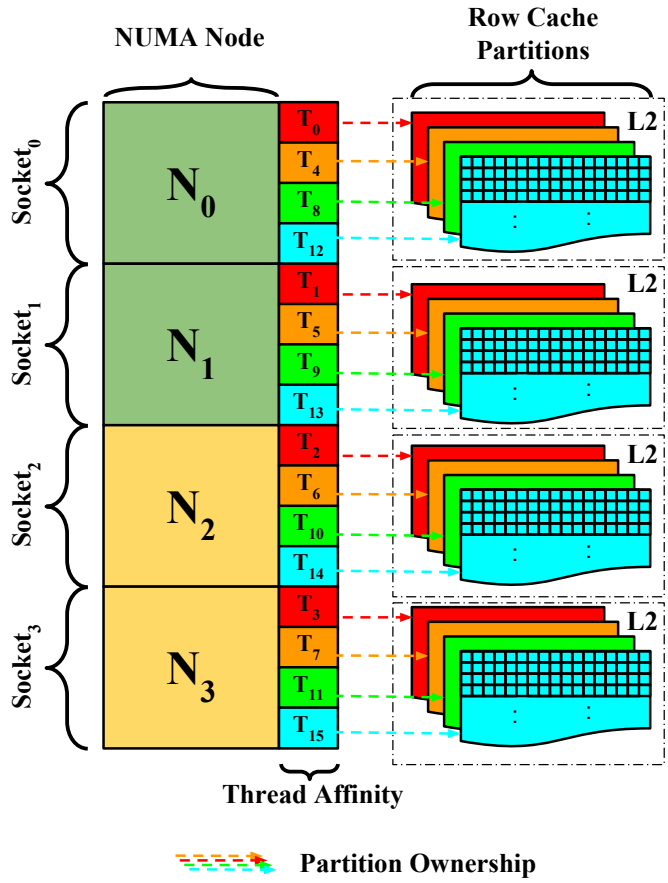


Figure 4.5: The structure of the row cache for SEM applications in a four socket, four NUMA node machine utilizing 16 threads. Partitioning the row cache eliminates the need for locking during cache population. The aggregate size of all row cache partitions resides within the NUMA-node shared L2 cache.

Lazy update mode: the row cache lazily updates on specified iterations based on a user defined *cache update interval* (I_{cache}). The cache updates/refreshes at iteration I_{cache} then the update frequency increases quadratically such that the next row cache update is performed after $2I_{cache}$, then $4I_{cache}$ iterations and so forth. This means that row-data in the row cache remains static for several iterations before the row cache is flushed then repopulated. This tracks the row activation patterns of algorithms like k-means, mb-kmeans, sk-means, and divisive hierarchical clustering. In early iterations, the cache provides little benefit, because row activations are random. As the

algorithm progresses, the same data points tend to stay active for many consecutive iterations. As such, much of the cache remains static for longer periods of time. We set I_{cache} to 5 for all experiments. The choice trades-off cache freshness for reduced cache maintenance. We demonstrate the efficacy of this design in Figure 4.10.

Active update mode: the row cache can also function as a traditional Least Recently Used (LRU) cache. This mode simply stores the more recently requested rows and evicts those that are less popular. Intuitively, this mode has higher maintenance overhead, but is more general for cases in which data access patterns are less predictable.

4.6 Distributed Design

We scale to the distributed setting through the Message Passing Interface (MPI). We employ modular design principles and build our distributed functionality as a layer above our parallel in-memory framework. Each machine maintains a decentralized *driver* (MPI) process that launches *worker* threads that retain the NUMA performance optimizations across its multiple processors. We partition a data set once per machine in the cluster, then again within a single machine. Global data is each specified as duplicated or can be requested from the initiating process.

We do not address load balancing between machines in the cluster. We recognize that in some cases it may be beneficial to dynamically dispatch tasks, but we argue that this would negatively affect the performance enhancing NUMA polices. We further argue that the gains in performance of our data partitioning scheme (Figure 4.1) outweigh the effects of skew in this setting. We validate these assertions empirically in Section 4.8.8

4.7 Application Programming Interface (API)

clusternor provides a C++ API on which users may define their own algorithms. There are two core components:

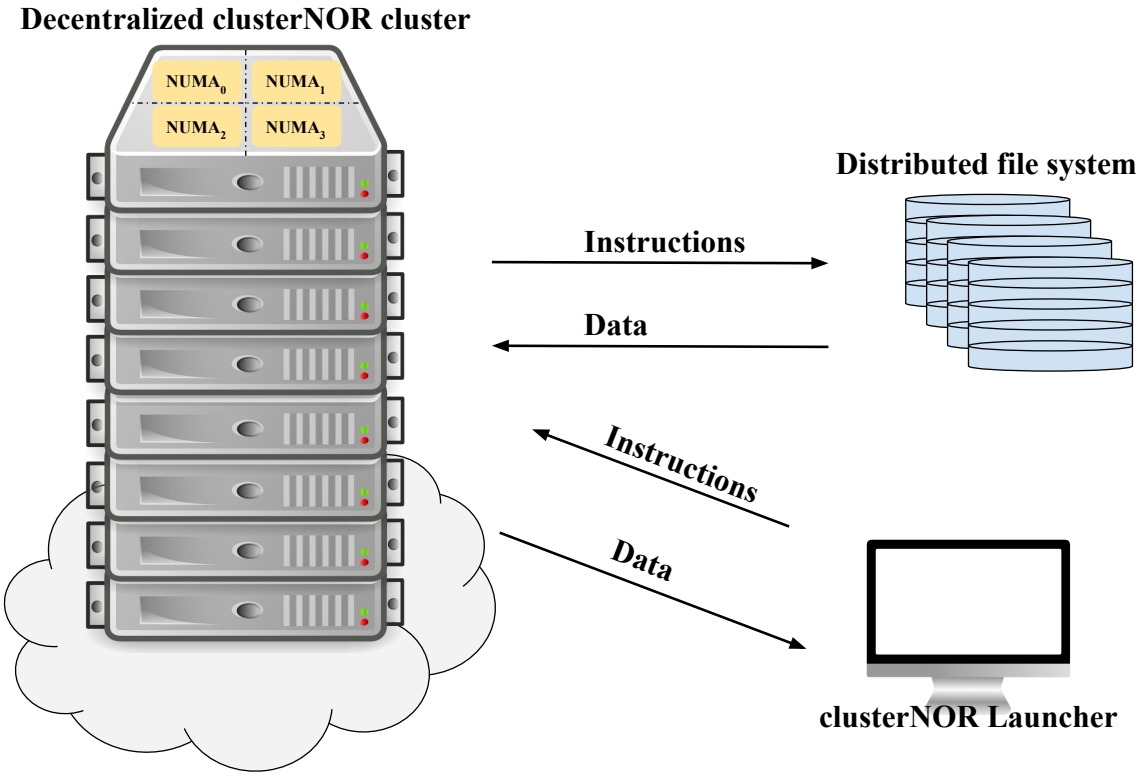


Figure 4.6: The decentralized distributed design of clusternor utilizes per-machine NUMA optimizations developed for single-node computations leading to state-of-the-art performance.

- the base iterative interface, `base`.
- the hierarchical iterative interface, `hclust`.

, in addition to two API extensions:

- the Semi-External Memory interface, `sem`.
- the distributed memory interface, `dist`.

4.7.1 `base`

The `base` interface provides developers with abstract methods that can be overridden to implement a variety of algorithms, such as k-means, mini-batch k-means, fuzzy C-means, and k-mediods (Sections 4.2.1, 4.2.4, 4.2.5, and 4.2.6).

- `run()`: Defines algorithmic specific steps for a particular application. This generally follows the serial algorithm.
- `MMStep()`: Used when both MM steps can be performed simultaneously. and reduces the effect of the barrier between the two steps.
- `M1Step()`: Used when the Majorize or Minorize step must be performed independently from the Minimization or Maximization step.
- `M2Step()`: Used in conjunction with `M1Step` as the Minimization or Maximization step of the algorithm.

4.7.2 `hclust`

The `hclust` interface extends `base` and is used to develop algorithms in which clustering is performed in a hierarchical fashion, such as H-means, X-means, and G-means (Sections 4.2.7, 4.2.8, and 4.2.9). For performance reasons, this interface is iterative rather than recursive. We discuss this design decision and its merits in Section 4.4. `hclust` provides the following additional abstract methods for user definition:

- `SplitStep()`: Used to determine when a cluster should split.
- `HclustUpdate()`: Used to update the hierarchical global state from one iteration to the next.

4.7.3 `sem`

The SEM interface builds upon `base` and `hclust` and incorporates a modified FlashGraph [17] API that we extend to support matrices and iterative clustering algorithms. The interface provides an abstraction over an asynchronous I/O model in which data are requested from disk and computation is overlapped with I/O transparently to users:

- `request(ids[])`: Issues I/O requests to the underlying storage media for the feature-vectors associated with the entries in `ids[]`.

4.7.4 `dist`

The distributed interface builds upon `base` and `hclust` creating infrastructure to support distributed processing. As is common with distributed memory, there also exist *optional* primitives for data synchronization, scattering and gathering, if necessary. Mandatory methods pertain to organizing state before and after computation and are abstractions above MPI calls:

- `OnComputeStart()`: Pass state or configuration details to processes when an algorithm begins.
- `OnComputeEnd()`: Extract state or organize algorithmic metadata upon completion of an algorithm.

4.7.5 Code Example

We provide a high-level implementation of the G-means algorithm written within clusternor to run in parallel on a standalone server. The simple C++ interface provides an abstraction that encapsulates parallelism, NUMA-awareness and cache friendliness. This code can be extended to SEM and distributed memory by simply inheriting from and implementing the required methods from `sem` and `dist`.

```
using namespace clusterNOR;


class gmeans : public hclust {
  void MMstep() {
    for (auto& sample : samples()) { // Data iterator
      auto best = min(Euclidean(sample, clusters()));
      JoinCluster(sample, best);
    }
  }


  void SplitStep() override {
    for (auto& sample : samples())
      if (ClusterIsActive(sample))
        AndersonDarlingStatistic(sample);
  }


  void run() override {
    while (nclust() < kmax()) {
      initialize(); // Starting conditions
      MMstep();
      SplitStep();
      Sync(); // Split clusters
      if (SteadyState())
        break; // Splits impossible
    }
  }
```

4.8 Experimental Evaluation

We begin the evaluation of clusternor by benchmarking the performance and efficacy of our optimizations for the k-means application alone. k-means is a core algorithm for the framework and a building block upon which other applications like mini-batch k-means, H-means, X-means and G-means are built. For brevity we refer to the *k-means NUMA Optimized Routine* as knor. Finally, we complete our evaluation by benchmarking all applications described in Section 4.9.

We evaluate knor optimizations and benchmark against other state-of-the-art frameworks. In Section 4.8.3 we evaluate the performance of the knor baseline single threaded implementation to ensure all speedup experiments are relative to a state-of-the-art baseline performance. Sections 4.8.4 and 4.8.5 evaluate the effect of specific optimizations on our in-memory and semi-external memory tools respectively. Section 4.8.6 evaluates the performance of k-means both in-memory and in the SEM setting relative to other popular state-of-the-art frameworks from the perspective of time and resource consumption. Section 4.8.8 specifically performs comparison between knord and MLlib in a cluster.

We evaluate knor optimizations on the Friendster top-8 and top-32 eigenvector datasets, because the Friendster dataset represents real-world machine learning data. The Friendster dataset is derived from a graph that follows a power law distribution of edges. As such, the resulting eigenvectors contain natural clusters with well defined centroids, which makes MTI pruning effective, because many data points fall into strongly rooted clusters and do not change membership. These trends hold true for other large-scale datasets, albeit to a lesser extent on uniformly random generated data (Section 4.8.6). The datasets we use for performance and scalability evaluation are shown in Table 4.2. Additionally, a summary of knor routine memory bounds is shown in Table 4.1.

We use the following notation throughout the evaluation:

- **knori**: k-means, in-memory, on a standalone machine.
- **knori-**: knori, with MTI pruning *disabled*.
- **knors**: k-means, in SEM mode, on a standalone machine with attached SSDs.
- **knors-**: knors, with MTI pruning *disabled*.
- **knors--**: knors, with both MTI pruning and the row cache (RC) *disabled*.
- **knord**: k-means, in a distributed cluster of machines, completely in-memory and in the cloud.
- **knord-**: knord with MTI pruning *disabled*.

- **MLlib-EC**2: MLlib's k-means, on Amazon EC2 instances [76].
- **MPI**: a pure MPI [77] distributed implementation of ||Lloyd's (Section 3.2.1) with MTI pruning.
- **MPI-**: a pure MPI distributed implementation of ||Lloyd's with MTI pruning *disabled*.

Table 4.1: Asymptotic memory complexity of knor routines.

Module / Routine	Memory complexity
Naïve Lloyd's	$\mathcal{O}(nd + kd)$
knors-, knors--	$\mathcal{O}(n + Tkd)$
knors	$\mathcal{O}(2n + Tkd + k^2)$
knori-, knord-	$\mathcal{O}(nd + Tkd)$
knori, knord	$\mathcal{O}(nd + Tkd + n + k^2)$

Table 4.2: The datasets under evaluation in this study.

Data Matrix	n	d	**Size**
Friendster-8 [78] eigenvectors	66M	8	4GB
Friendster-32 [78] eigenvectors	66M	32	16GB
Rand-Multivariate (RM_{856M})	856M	16	103GB
Rand-Multivariate (RM_{1B})	1.1B	32	251GB
Rand-Univariate (RU_{2B})	2.1B	64	1.1TB

For completeness we note versions of all frameworks and libraries we use for comparison in this study; Spark v2.0.1 for MLlib, H_2O v3.7, Turi v2.1, R v3.3.1, MATLAB R2016b, BLAS v3.7.0, Scikit-learn v0.18, MLpack v2.1.0.

4.8.1 Single Node Evaluation Hardware

We perform single node experiments on a NUMA server with four Intel Xeon E7-4860 processors clocked at 2.6 GHz and 1TB of DDR3-1600 memory. Each processor has 12 cores. The machine has three LSI SAS 9300-8e host bus adapters (HBA) connected to a SuperMicro storage chassis, in which 24 OCZ Intrepid 3000 SSDs are installed. The machine runs Linux kernel v4.4.0-124. The C++ code is compiled using mpicxx.mpich2 version 4.8.4 with the -O3 flag.

4.8.2 Cluster Evaluation Hardware

We perform distributed memory experiments on Amazon EC2 compute optimized instances of type c4.8xlarge with 60GB of DDR3-1600 memory, running Linux kernel v3.13.0-91. Each machine has 36 vCPUS, corresponding to 18 physical Intel Xeon E5-2666 v3 processors, clocking 2.9 GHz, sitting on 2 independent sockets. We allow no more that 18 independent MPI processes or equivalently 18 Spark workers to exist on any single machine. We constrain the cluster to a single availability zone, subnet and placement group, maximizing cluster-wide data locality and minimizing network latency on the 10 Gigabit interconnect. We measure all experiments from the point when all data is in RAM on all machines. For MLlib we ensure that the Spark engine is configured to use the maximum available memory and does not perform any checkpointing or I/O during computation.

4.8.3 Baseline Single-thread Performance

knori, even with MTI pruning *disabled*, performs on par with state-of-the-art implementations of Lloyd's algorithm. This is true for implementations that utilize generalized matrix multiplication (GEMM) techniques and vectorized operations, such as MATLAB [79] and BLAS [80]. We find the same to be true of popular statistics packages and frameworks such as MLpack [48], Scikit-learn [46] and R [81] all of which use highly optimized C/C++ code, although some use scripting language wrappers. Table 4.3 shows performance at 1 thread. Table 4.3 provides credence to our speedup results because our baseline single threaded performance tops other state-of-the-art serial

routines.

Table 4.3: Serial performance of popular, optimized k-means routines, all using Lloyd's algorithm, on the Friendster-8 dataset. For fairness all implementations perform all distance computations. The **Language** column refers to the underlying language of implementation and not any user-facing higher level wrapper.

Implementation	Type	Language	Time/iter (sec)
knori-	**Iterative**	**C++**	**7.49**
MATLAB	GEMM	C++	20.68
BLAS	GEMM	C++	20.7
R	Iterative	C	8.63
Scikit-learn	Iterative	Cython	12.84
MLpack	Iterative	C++	13.09

4.8.4 In-memory Optimization Evaluation

We show NUMA-node thread binding, maintaining NUMA memory locality, and NUMA-aware task scheduling is highly effective in improving performance. We achieve near-linear speedup (Figure 4.7). Because the machine has 48 physical cores, speedup degrades slightly at 64 cores; additional speedup beyond 48 cores comes from simultaneous multithreading (hyperthreading). The NUMA-aware implementation is nearly 6x faster at 64 threads compared to a routine containing no NUMA optimizations, henceforth referred to as *NUMA-oblivious*. The NUMA-oblivious routine relies on the OS to determine memory allocation, thread scheduling, and load balancing policies.

We further show that although both the NUMA-oblivious and NUMA-aware implementation speedup sub-linearly, the NUMA-oblivious routine has a lower linear constant when compared with a NUMA-aware implementation (Figure 4.7).

Increased parallelism amplifies the performance degradation of the NUMA-oblivious implementation. We identify the following as the greatest contributors:

- the NUMA-oblivious allocation policies of traditional memory allocators, such as `malloc`, place

data in a contiguous chunk within a single NUMA memory bank whenever possible. This leads to a large number of threads performing remote memory accesses as the number of threads increase;

- a dynamic NUMA-oblivious task scheduler may give tasks to threads that cause worker threads to perform many more remote memory accesses than necessary compared to a NUMA-aware scheduler.

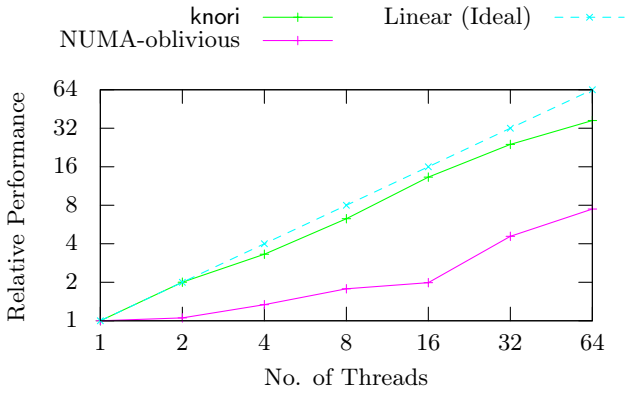


Figure 4.7: Speedup of knori (which is NUMA-aware) vs. a NUMA-oblivious routine on the Friendster top-8 eigenvector dataset, with $k = 10$.

We demonstrate the effectiveness of a NUMA-aware partitioned task scheduler for pruned computations via knori (Figure 4.8). We define a *task* as a block of data points in contiguous memory given to a thread for computation. We set a minimum *task size*, i.e. the number of data points in the block, to 8192. We empirically determine that this task size is small enough to not artificially introduce skew in billion-point datasets while simultaneously providing enough work to amortize the cost of locking at the task scheduler. We compare against a static and a first in, first out (FIFO) task scheduler. The static scheduler preassigns n/T rows to each worker thread. The FIFO scheduler first assigns threads to tasks that are local to the thread's partition of data, then allows threads to steal tasks from straggler threads whose data resides on *any* NUMA node.

We observe that as k increases, so does the potential for skew. When $k = 10$, the NUMA-

aware scheduler performs negligibly worse than both FIFO and static scheduling, but as k, increases the NUMA-aware scheduler improves performance—by more than 40% when $k = 100$. We observe similar trends in other datasets; we omit these redundant results.

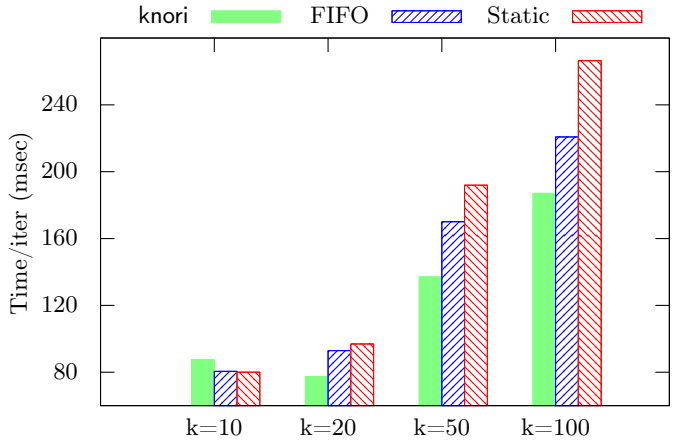
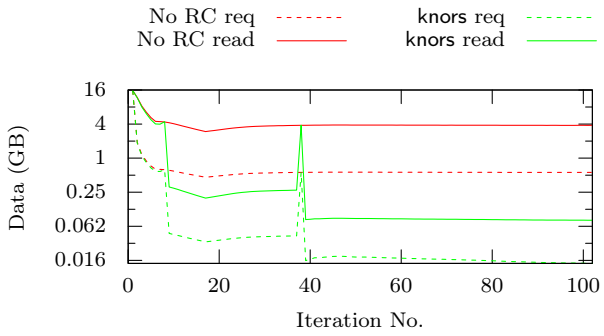


Figure 4.8: Performance of the partitioned NUMA-aware scheduler (clusternor default) vs. FIFO and static scheduling for knori on the Friendster-8 dataset.

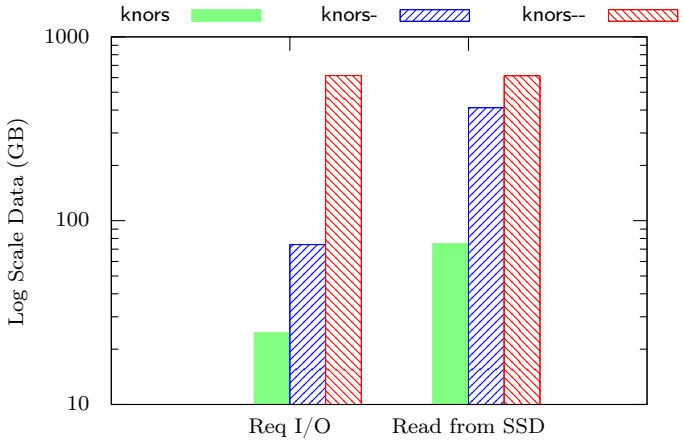
4.8.5 Semi-External Memory Evaluation

We evaluate knors optimizations, performance and scalability. We set a small *page cache* size for FlashGraph (4KB) to minimize the amount of superfluous data read from disk due to data fragmentation. Additionally, we disable checkpoint failure recovery during performance evaluation for both our routines and those of our competitors.

We drastically reduce the amount of data read from SSDs by utilizing the row cache. Figure 4.9a shows that as the number of iterations increase, the row cache's ability to reduce I/O and improve speed also increases because most rows that are active are pinned in memory. Figure 4.9b contrasts the total amount of data that an implementation requests from SSDs with the amount of data SAFS actually reads and transports into memory. When knors *disables* both MTI pruning and the row cache i.e., knors--, every request issued for row-data is either served by FlashGraph's



(a) knors data requested (req) from SSDs vs. data read (read) from SSDs each iteration when the row cache (RC) is *enabled* or *disabled*. MTI pruning allows fewer data points to be requested from SSDs, but the file system must still read an entire block from SSDs in which some data may not be useful. As a result, there is a discrepancy between the quantity of data requested and the quantity read.



(b) Total data requested (req) vs. data read from SSDs when (i) both MTI and RC are *disabled* (knors--), (ii) Only MTI is *enabled* (knors-), (iii) both MTI and RC are *enabled* (knors). Without pruning, all data are requested and read.

Figure 4.9: The effect of the row cache and MTI on I/O for the Friendster top-32 eigenvectors dataset. Row cache size = 512MB, page cache size = 1GB, $k = 10$.

page cache or read from SSDs. When knors *enables* MTI pruning, but *disables* the row cache i.e., knors-, we read an order of magnitude more data from SSDs than when we *enable* the row cache. Figure 4.9 demonstrates that a page cache is **not** sufficient for k-means and that caching at the

granularity of row-data is necessary to achieve significant reductions in I/O and improvements in performance for real-world datasets. Additionally, this observation is applicable to all computation pruning and sub-sampling applications where selective I/O is possible.

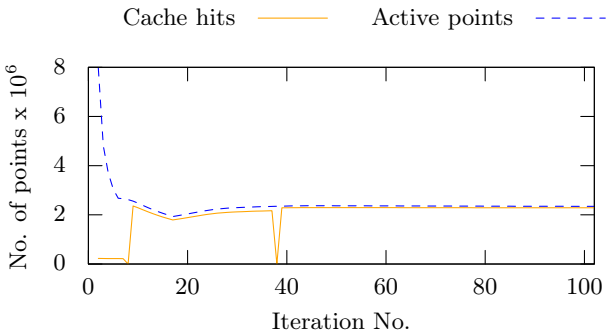


Figure 4.10: Row cache hits per iteration contrasted with the maximum achievable number of hits on the Friendster top-32 eigenvectors dataset.

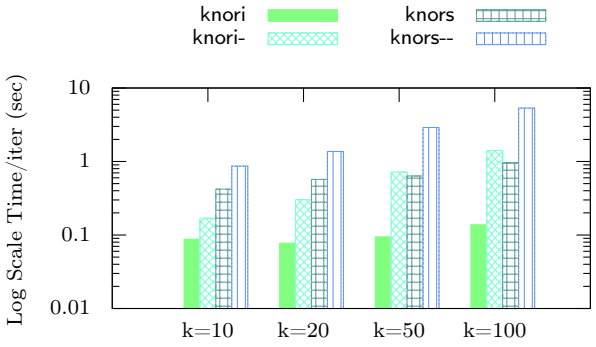
clusternor's lazy row update mode reduces I/O significantly for this application. Figure 4.10 justifies our design decision for a lazily updated row cache. As the algorithm progresses, we obtain nearly a 100% cache hit rate, meaning that knors operates at in-memory speeds for the vast majority of iterations.

MTI Performance Characteristics

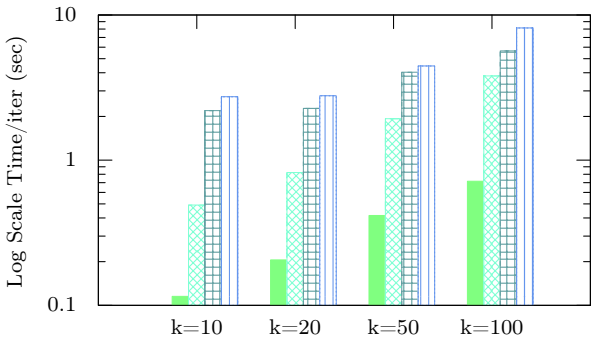
Figures 4.11a and 4.11b highlight the performance improvement of knor modules with MTI *enabled* over MTI *disabled* counterparts. We show that MTI provides a few factors of improvement in time when enbabled. Figure 4.11c highlights that MTI increases the memory load by negligible amounts compared to non-pruning modules. We conclude that MTI (unlike TI) is a viable optimization for large-scale datasets.

4.8.6 knor vs. Other Frameworks

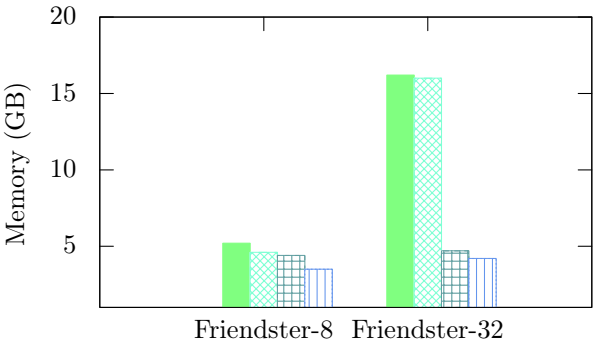
We evaluate the performance of knor in comparison with other frameworks on the datasets in Table 4.2. We show that knori achieves greater than an order of magnitude improvement over



(a) Runtime performance of k-means on the Friendster-8 dataset.



(b) Runtime performance of k-means on the Friendster-32 dataset.



(c) Memory comparison of fully optimized knor routines (knori, knors) compared to more vanilla knor routines (knori-, knors--).

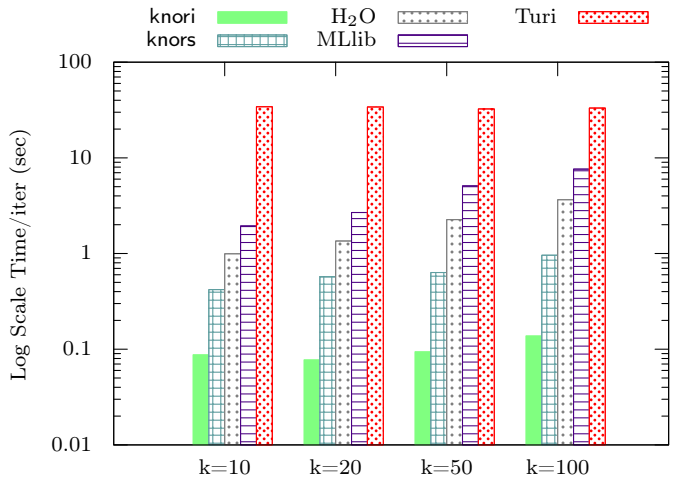
Figure 4.11: Performance and memory usage comparison of knor modules on matrices from the Friendster graph top-8 and top-32 eigenvectors.

other state-of-the-art frameworks. Finally, we demonstrate knors outperforms other state-of-the-art frameworks by several factors.

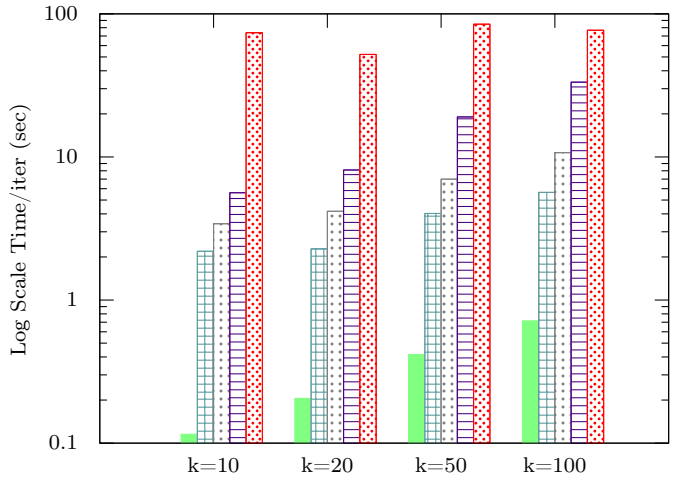
Both our in-memory and semi-external memory modules incur little memory overhead when compared with other frameworks. Figure 4.12c shows memory consumption. We note that MLlib requires the placement of temporary Spark block manager files. Because the block manager cannot be disabled, we provide an in-memory RAM-disk so as to not influence MLlib's performance negatively. We configure MLlib, H_2O and Turi to use the minimum amount of memory necessary to achieve their highest performance. We acknowledge that a reduction in memory for these frameworks is possible, but would degrade computation time and lead to unfair comparisons. All measurements are an average of 10 runs. We drop all caches between runs.

We demonstrate that knori is no less than an order of magnitude faster than all competitor frameworks (Figure 4.12). knori is often hundreds of times faster than Turi. Furthermore, knors is consistently twice as fast as competitor in-memory frameworks. We further demonstrate performance improvements over competitor frameworks on algorithmically identical implementations by *disabling* MTI. knori- is nearly 10x faster than competitor solutions, whereas knors- is comparable and often faster than competitor in-memory solutions. We attribute our performance gains over other frameworks when MTI is *disabled* to our parallelization scheme for Lloyd's (Algorithm 1). Lastly, Figure 4.11 demonstrates a consistent 30% improvement in knors when we utilize the row cache. This is evidence that the design of our lazily updated row cache provides a performance boost.

Finally, comparing knori- and knors-- to MLlib, H_2O and Turi (Figures 4.11 and 4.12) reveals knor to be several times faster and to use significantly less memory. This is relevant because knori- and knors-- are algorithmically identical to k-means within MLlib, Turi and H_2O.



(a) Runtime performance of k-means on the Friendster-8 dataset.

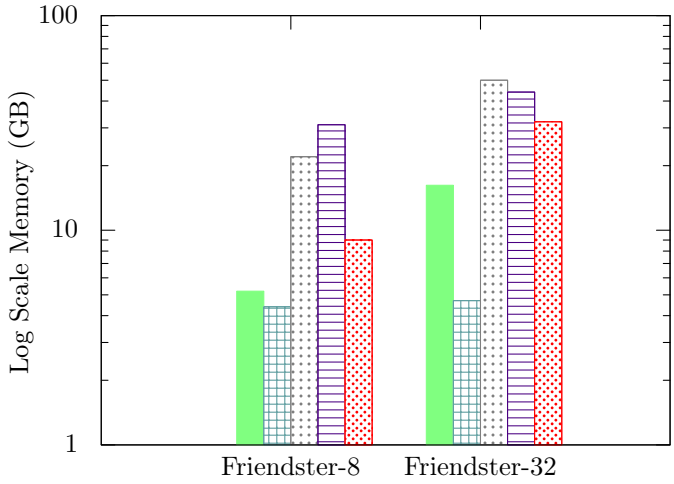


(b) Runtime performance of k-means on the Friendster-32 dataset.

4.8.7 Single-node Scalability Evaluation

To demonstrate scalability, we compare the performance of k-means on synthetic datasets drawn from random distributions that contain hundreds of millions to billions of data points. Uniformly random data are typically the worst case scenario for the convergence of k-means, because many data points tend to be near several centroids.

Both in-memory and SEM modules outperform popular frameworks on 100GB+ datasets. We achieve 7-20x improvement when in-memory and 3-6x improvement in SEM when compared to



(c) Peak memory consumption on the Friendster datasets, with $k = 10$. Row cache size = 512MB, page cache size = 1GB.

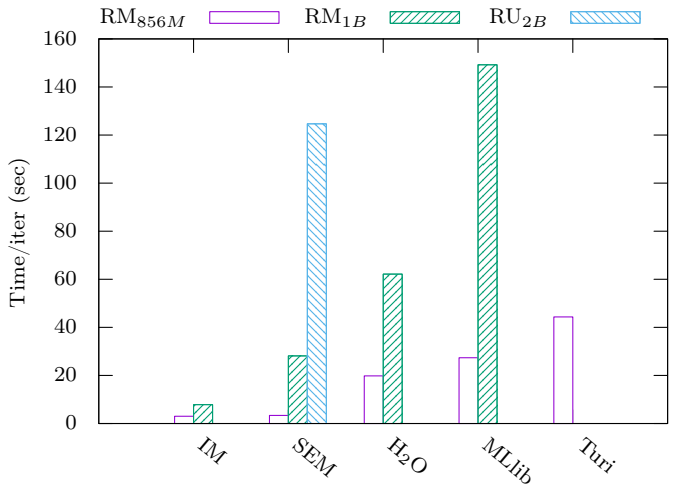
Figure 4.12: knor routines outperform competitor solutions in runtime performance and memory consumption.

MLlib, H_2O and Turi. As data increases in size, the performance difference between knori and knors narrows because there is now enough data to mask I/O latency and to turn knors from an being I/O bound to being computation bound. We observe knors is only 3-4x slower than its in-memory counterpart in such cases.

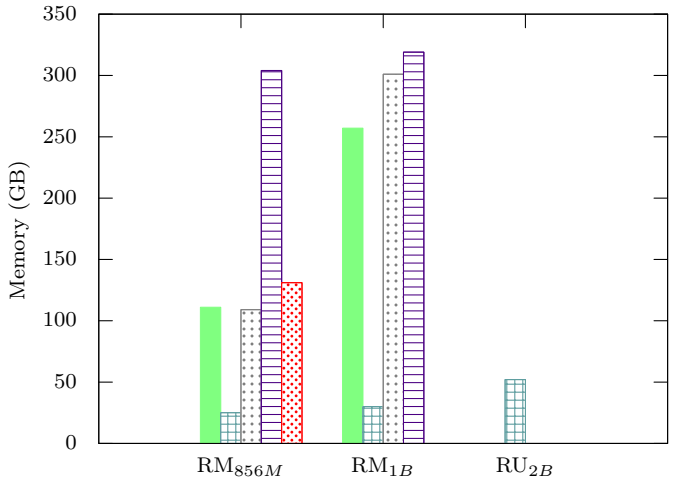
Memory capacity limits the scalability of k-means and semi-external memory allows algorithms to scale well beyond the limits of physical memory. The 1B point matrix (RM_{1B}) is the largest that fits in 1TB of memory on our machine. At 2B points (RU_{2B}), semi-external memory algorithms continue to execute proportionally and all other algorithms fail.

4.8.8 Distributed Comparison vs. Other Frameworks

We analyze the performance of knord and knord- on Amazon's EC2 cloud in comparison to that of (i) MLlib (**MLlib-EC2**), (ii) a pure MPI implementation of our ||Lloyd's algorithm with MTI pruning (**MPI**), and (iii) a pure MPI implementation of ||Lloyd's algorithm with pruning *disabled*



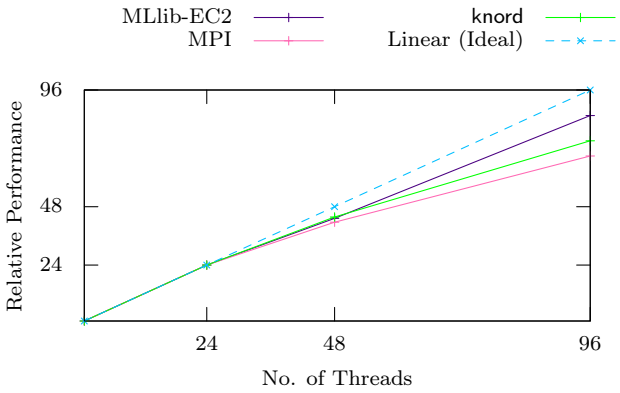
(a) Per iteration runtime of each routine.



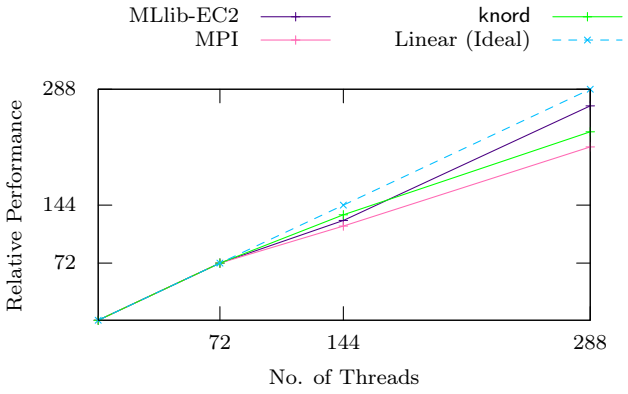
(b) Memory consumption of each routine.

Figure 4.13: Performance comparison on RM$_{856M}$ and RM$_{1B}$ datasets. Turi is unable to run on RM$_{1B}$ on our machine and only SEM routines are able to run on RU$_{2B}$ on our machine. Page cache size = 4GB, Row cache size = 2GB.

(**MPI-**). Note that H$_2$O has no distributed memory implementation and Turi discontinued their distributed memory interface prior to our experiments.



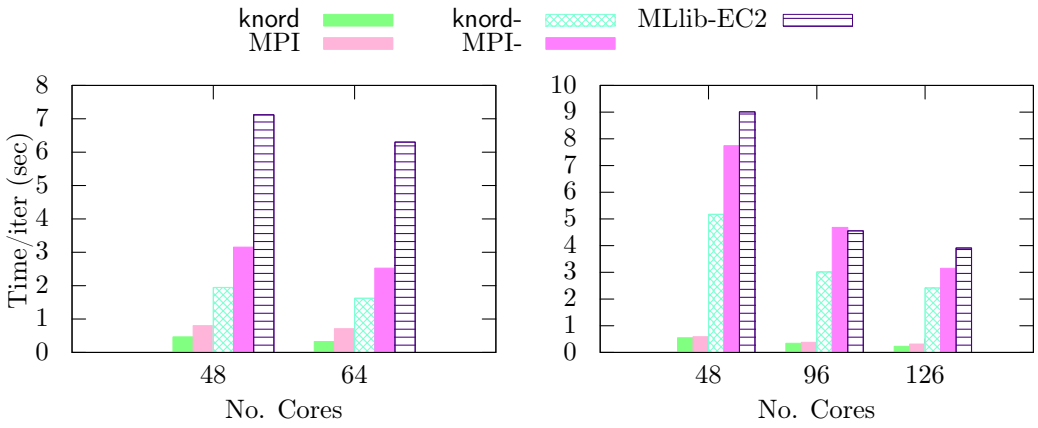
(a) Distributed speedup comparison on the Friendster-32 dataset.



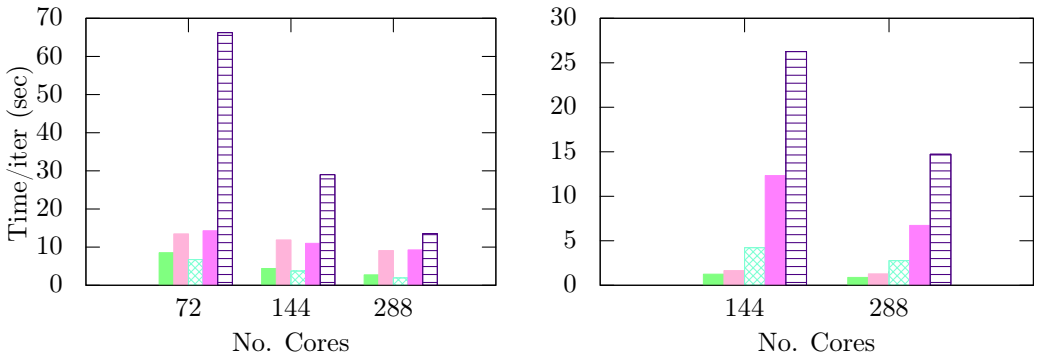
(b) Distributed speedup comparison on the RM_{1B} dataset.

Figure 4.14: Speedup experiments are normalized to each implementation's serial performance. Each machine has 18 physical cores with 1 thread per core.

Figures 4.14 and 4.15 reveal several fundamental and important results. Figure 4.14 shows that knord scales well to very large numbers of machines, performing within a constant factor of linear performance. This is a necessity today as many organizations push big-data computation to the cloud. Figure 4.15 shows that in a cluster, knord, even with TI *disabled*, outperforms MLlib by a factor of 5 or more. This means we can often use fractions of the hardware required by MLlib to perform equivalent tasks. Figure 4.15 demonstrates that knord also benefits from our in-memory NUMA optimizations as we outperform a NUMA-oblivious MPI routine by 20-50%, depending on the dataset. Finally, Figure 4.15 shows that MTI remains a low-overhead, effective method to reduce



(a) Friendster8 (left) and Friendster32 (right) datasets per iteration runtime for $k = 100$.



(b) RM_{856M} (left) and RM_{1B} (right) datasets per iteration runtime for $k = 10$.

Figure 4.15: Distributed performance comparison of knord, MPI and MLlib on Amazon's EC2 cloud. Each machine has 18 physical cores with 1 thread per core.

computation even in the distributed setting.

Semi-External Memory in the Cloud

We continue knor evaluation by measuring the performance of knors on a single 32 core i3.16xlarge machine with 8 SSDs on Amazon EC2 compared to knord, MLlib and an optimized MPI routine running in a cluster. We run knors with 48 threads, with extra parallelism coming from symmetric multiprocessing. We run all other implementations with the same number of processes/threads as physical cores.

Figure 4.16 highlights that knors often outperforms MLlib even when MLLib runs in a cluster that contains more physical CPU cores. knors has comparable performance to both MPI and knord, leading to our assertion that the SEM scale-up model should be considered prior to moving to the distributed setting.

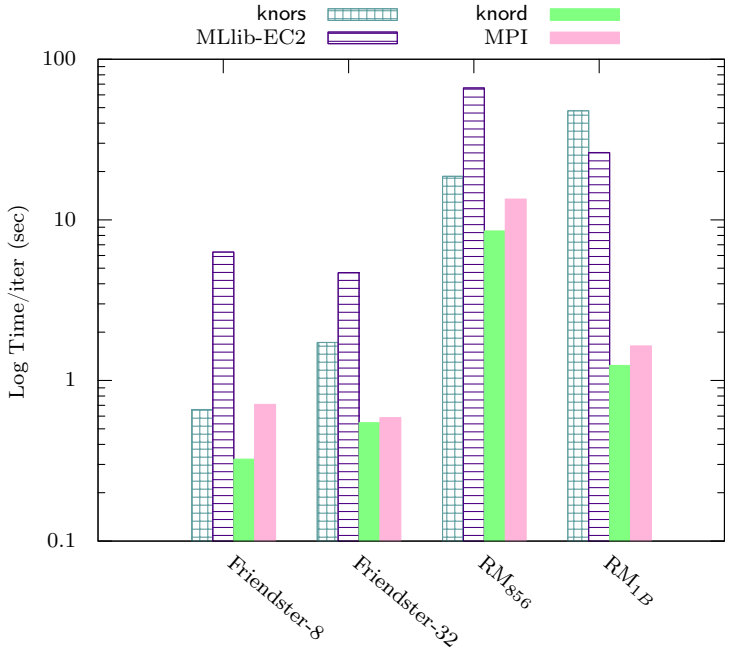


Figure 4.16: Performance comparison of knors to distributed packages. knors uses one i3.16xlarge machine with 32 physical cores. knord, MLlib-EC2 and MPI use 3 c4.8xlarge with a total of 48 physical cores for all datasets other than RU_{1B} where they use 8 c4.8xlarge with a total of 128 physical cores.

4.9 Application Evaluation

We benchmark the performance of the nine applications developed using clusternor (Section 4.2). We present results for in-memory execution. The relative performance in other settings, SEM and distributed memory, track in-memory results closely. Figure 4.17 demonstrates that for applications with similar computational complexity as k-means, clusternor achieves comparable per-

formance to knor, which we have shown to be state-of-the-art. This is a strong indication that all other applications are comparable to state-of-the-art as well. At this time, to our knowledge, there exist no other open-source large-scale parallel clustering libraries with whom we can compare performance. As such the clusternor benchmark applications enable scientific experimentation with clustering algorithms at a scale previously unavailable.

Figure 4.17 demonstrates that applications with similar algorithmic complexity to k-means perform comparably to knor. This is a strong demonstration that clusternor optimizations are applicable to a wide range of MM algorithms. For mini-batch k-means (mbk-means), we set the batch size, B, to 20% of the dataset size. This is roughly twice the value used in experiments by Sculley [50] in his seminal work describing the algorithm. We highlight that even though mbk-means performs several factors fewer distance computations compared to batched k-means (e.g., knor), its computation time can be greater due to the algorithmically serial gradient step (Equation 4.4). Furthermore, we note that the computation time of fuzzy c-means can be up to an order of magnitude slower than that of k-means. This is due to fc-means performing a series of linear algebraic operations, some of which must be performed outside the confines of the parallel constructs provided by the framework. As such, the application's performance is bound by the computation of updates to the cluster *contribution matrix*, an $\mathcal{O}(kn)$ data structure containing the probability of a data point being in a cluster.

Hierarchical clustering algorithms also perform well in comparison to knor, despite requiring heavier logic between iterations. To benchmark H-means, X-means and G-means we perform 20 iterations of k-means between each divisive cluster-splitting step i.e., the `SplitStep`. We recognize that the computation cost of the hierarchical algorithms for one iteration is lower than that of k-means, but argue that performing the same number of iterations at each level of the hierarchy provides a comparable measure of computation. Furthermore, X-means requires the computation of BIC and G-means requires the computation of the Anderson-Darling statistic between `SplitStep`s. This increases the cost of hierarchical clustering over H-means (Figure 4.18), in which X-means and

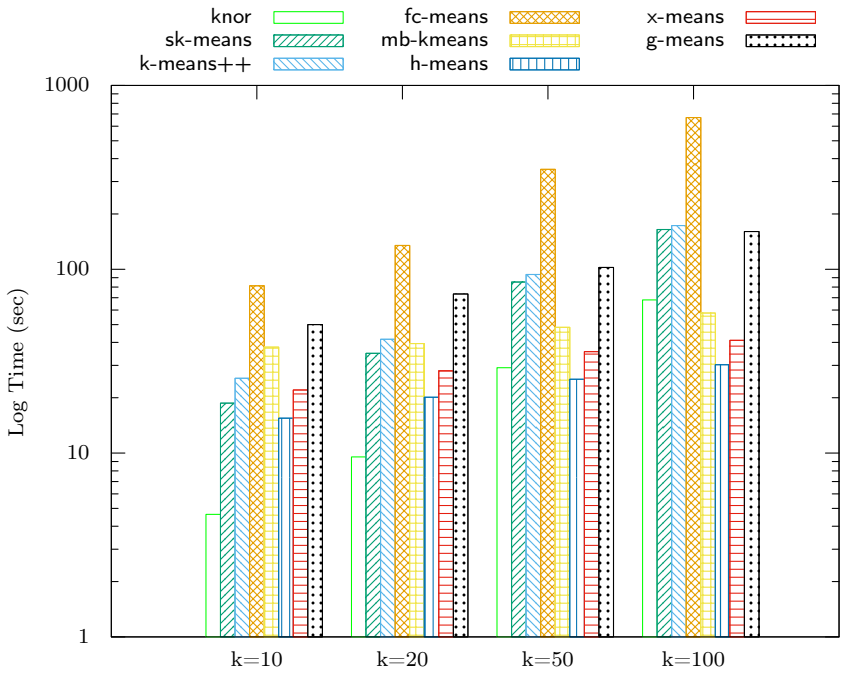


Figure 4.17: In-memory performance of clusternor benchmark applications on the Friendster-32 dataset. We fix the number of iterations to 20 for all applications and use a mini-batch size of 20% of the data size for mb-kmeans.

G-means perform at about 70% and 30% of the performance of H-means.

We present the result of the k-medoids experiment (Table 4.4) on a 250 thousand subsampling of the Friendster-32 dataset. We subsample because the complexity of k-medoids is significantly higher than that of all other applications making it infeasible for even our smallest dataset. Nevertheless, k-medoids demonstrates the programming flexibility of our framework. We observe that as the number of clusters, k, increases the computational overhead reduces. This is due to the size of each cluster generally decreasing as data points are spread across more clusters. clusternor ensures that the degree of parallelism achieved is independent of the number of clusters. The most intensive medoid *swap* procedure now requires less inter-cluster computation leading to reduced computation time. We vary the degree to which we subsample within the *swap* procedure from 20% up to 100% to highlight the observed phenomenon.

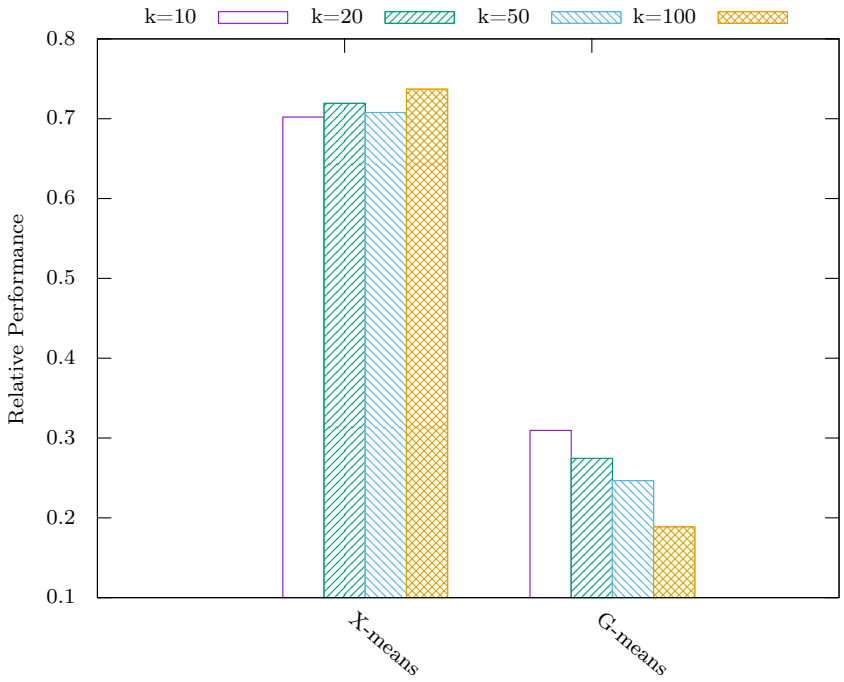


Figure 4.18: The relative performance of hierarchical algorithms in comparison to H-means, the baseline hierarchical cluster application on the Friendster-32 dataset

Table 4.4: The performance of k-medoids on a 250 thousand random sampling of the Friendster-32 dataset run for 20 iterations.

Sample %	$k = 10$	$k = 20$	$k = 50$	$k = 100$
20	455.95s	679.52s	262.42s	134.46s
50	2003.74s	1652.90s	717.19s	342.34s
100	2154.81s	2616.57s	1801.56s	761.98s

4.10 Discussion

clusternor demonstrates that there are large performance benefits associated with NUMA-targeted optimizations. Data locality optimizations, such as NUMA-node thread binding, NUMA-aware task scheduling, and NUMA-aware memory allocation schemes, provide several times speedup for MM algorithms. Many of the optimizations within clusternor are applicable to data processing

frameworks built for non-specialized commodity hardware.

For technical accomplishments, we accelerate k-means and its derived algorithms by over an order of magnitude by rethinking Lloyd's algorithm for modern multiprocessor NUMA architectures through the minimization of critical regions. Our modifications to Lloyd's are relevant to both in-memory, distributed memory and semi-external memory. Additionally, we formulate a minimal triangle inequality (MTI) pruning algorithm that further boosts the performance of k-means on real-world billion point datasets by over 100x when compared to some popular frameworks. MTI does so without significantly increasing memory consumption.

Finally, clusternor provides an extensible unified framework for in-memory, semi-external memory and distributed memory iterative algorithm development. The clusternor benchmark applications provide a scalable, state-of-the-art clustering library. Bindings to the open source library are accessible within 'CRAN', the R Programming Language [82] package manager, under the name *clusternor*. We are an open source project available at
https://github.com/flashxio/knor. Our flagship knor application, on which this work is based, receives hundreds of downloads monthly on both CRAN and `pip`, the Python package manager.

Chapter 5

Conclusion

This thesis investigates the effects of NUMA optimizations, fine-grained I/O management and effective caching policies on algorithms for large-scale data analysis. We address the need for libraries and frameworks that target large-scale data analysis from the perspective of graphs, and dense feature vector datasets. We scale graph and iterative machine learning algorithms through semi-external memory, and eventually, distributed memory, once the capacity of a single machine is exhausted.

This thesis advances the state-of-the art in multicore NUMA optimizations and semi-external memory computation for graphs and Majorize-Minimization algorithms. Core contributions within this body of work include:

- Identification of key principles for SEM vertex-centic graph application development.
- The development of an extensible high-level language, vertex-centric SEM graph library, Graphyti.
- Algorithmic and multi-core computation advancements of pruning techniques for the k-means algorithm via the Minimal Triangle Inequality (MTI), and ||Lloyds.
- The development of an extensible framework for community detection that encompasses fully

in-memory, SEM, and distributed memory.

We glean several conclusions from this line of investigation. From knor and clusternor we conclude that large performance gains ranging from several factors to several orders of magnitude improvement are left on the table by neglecting NUMA optimizations on modern multicore machines. The clusternor framework is evidence that NUMA-centric, fine-grain, caching and I/O optimizations are effective tools enabling resource minimality while not sacrificing performance. We show that our k-means application developed within clusternor out-performs commercial grade products in all memory settings.

From Graphyti, we conclude that there are a few key principles that developers should follow in order to develop I/O minimal, performant, vertex-centric SEM applications. We demonstrate that by incorporating these principles into applications, they stand to gain several factors of improvement over unoptimized algorithms. These principles encode latent asynchrony into applications, minimize I/O and, reduce memory consumption, and optimize cache access patterns and reuse. The aforementioned principles are manifested within Graphyti, but are further generalized and incorporated into the design of the clusternor framework.

Through Graphyti, knor and clusternor we comprehensively investigate the runtime and memory improvements attainable through NUMA and I/O sensitive design of libraries and frameworks. We demonstrate the efficacy and suitability of the semi-external memory paradigm for large-scale graph analysis and machine learning. We do so by modifying the FlashGraph engine to support larger dense feature-vector datasets. Additionally, we rethinking the bulk-synchronous processing model within vertex-centric graph engines and incorporate asynchrony in multi-stage algorithms. Finally, we demonstrate the applicability of NUMA optimizations beyond a single machine into the distributed setting. We conclude that a combination of I/O optimizations and NUMA awareness are at the core of scaling graph analytics and iterative Majorize-Minimization algorithms in the future.

Bibliography

[1] J. Ugander, B. Karrer, L. Backstrom, and C. Marlow, "The anatomy of the facebook social graph," *arXiv preprint arXiv:1111.4503*, 2011.

[2] "1999 czech financial dataset - real anonymized transactions - dataset by lpetrocelli," Jun 2017. [Online]. Available: https://data.world/lpetrocelli/czech-financial-dataset-real-anonymized-transactions

[3] J. Leskovec and A. Krevl, "SNAP Datasets: Stanford large network dataset collection," http://snap.stanford.edu/data, Jun. 2014.

[4] J. Bennett and S. Lanning, "The netflix prize," in *Proceedings of KDD cup and workshop*, vol. 2007, 2007, p. 35.

[5] N. Binkiewicz, J. T. Vogelstein, and K. Rohe, "Covariate assisted spectral clustering," *arXiv preprint arXiv:1411.2158*, 2014.

[6] V. Lyzinski, D. L. Sussman, D. E. Fishkind, H. Pao, L. Chen, J. T. Vogelstein, Y. Park, and C. E. Priebe, "Spectral clustering for divide-and-conquer graph matching," *Parallel Computing*, 2015.

[7] V. Lyzinski, M. Tang, A. Athreya, Y. Park, and C. E. Priebe, "Community detection and classification in hierarchical stochastic blockmodels," *arXiv preprint arXiv:1503.02115*, 2015.

[8] A. S. Das, M. Datar, A. Garg, and S. Rajaram, "Google news personalization: scalable online collaborative filtering," in *Proceedings of the 16th international conference on World Wide Web*. ACM, 2007, pp. 271–280.

[9] J. T. Vogelstein, Y. Park, T. Ohyama, R. A. Kerr, J. W. Truman, C. E. Priebe, and M. Zlatic, "Discovery of brainwide neural-behavioral maps via multiscale unsupervised structure learning," *Science*, vol. 344, no. 6182, pp. 386–392, 2014.

[10] L. B. Jorde and S. P. Wooding, "Genetic variation, classification and'race'," *Nature genetics*, vol. 36, pp. S28–S33, 2004.

[11] N. Patterson, A. L. Price, and D. Reich, "Population structure and eigenanalysis," 2006.

[12] Y. Low, J. E. Gonzalez, A. Kyrola, D. Bickson, C. E. Guestrin, and J. Hellerstein, "Graphlab: A new framework for parallel machine learning," *arXiv preprint arXiv:1408.2041*, 2014.

[13] S. Owen, R. Anil, T. Dunning, and E. Friedman, *Mahout in action*. Manning Shelter Island, 2011.

[14] X. Meng, J. Bradley, B. Yavuz, E. Sparks, S. Venkataraman, D. Liu, J. Freeman, D. Tsai, M. Amde, S. Owen *et al.*, "Mllib: Machine learning in apache spark," *arXiv preprint arXiv:1505.06807*, 2015.

[15] J. Ang, R. F. Barrett, R. Benner, D. Burke, C. Chan, J. Cook, D. Donofrio, S. D. Hammond, K. S. Hemmert, S. Kelly *et al.*, "Abstract machine models and proxy architectures for exascale computing," in *Hardware-Software Co-Design for High Performance Computing (Co-HPC), 2014*. IEEE, 2014, pp. 25–32.

[16] F. McSherry, M. Isard, and D. G. Murray, "Scalability! but at what cost?" in *15th Workshop on Hot Topics in Operating Systems (HotOS XV)*, 2015.

[17] D. Zheng, D. Mhembere, R. Burns, J. Vogelstein, C. E. Priebe, and A. S. Szalay, "FlashGraph: Processing billion-node graphs on an array of commodity SSDs," in *13th USENIX Conference on File and Storage Technologies (FAST 15)*, 2015.

[18] C. Elkan, "Using the triangle inequality to accelerate k-means," in *ICML*, vol. 3, 2003, pp. 147–153.

[19] J. MacQueen *et al.*, "Some methods for classification and analysis of multivariate observations," in *Proceedings of the fifth Berkeley symposium on mathematical statistics and probability*, vol. 1, no. 14. Oakland, CA, USA., 1967, pp. 281–297.

[20] D. Zheng, R. Burns, and A. S. Szalay, "Toward millions of file system IOPS on low-cost, commodity hardware," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, 2013.

[21] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: A system for large-scale graph processing," in *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, 2010.

[22] Y. Low, J. E. Gonzalez, A. Kyrola, D. Bickson, C. E. Guestrin, and J. Hellerstein, "Graphlab: A new framework for parallel machine learning," *arXiv preprint arXiv:1408.2041*, 2014.

[23] D. Nguyen, A. Lenharth, and K. Pingali, "A lightweight infrastructure for graph analytics," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. ACM, 2013, pp. 456–471.

[24] M. Kulkarni, K. Pingali, B. Walter, G. Ramanarayanan, K. Bala, and L. P. Chew, "Optimistic parallelism requires abstractions," in *ACM SIGPLAN Notices*, vol. 42, no. 6. ACM, 2007, pp. 211–222.

[25] S. P. Lloyd, "Least squares quantization in pcm," *Information Theory, IEEE Transactions on*, vol. 28, no. 2, pp. 129–137, 1982.

[26] A. P. Dempster, N. M. Laird, and D. B. Rubin, "Maximum likelihood from incomplete data via the em algorithm," *Journal of the royal statistical society. Series B (methodological)*, pp. 1–38, 1977.

[27] G. Csardi, T. Nepusz *et al.*, "The igraph software package for complex network research," *InterJournal, Complex Systems*, vol. 1695, no. 5, pp. 1–9, 2006.

[28] A. Hagberg, D. Schult, P. Swart, D. Conway, L. Séguin-Charbonneau, C. Ellison, B. Edwards, and J. Torrents, "Networkx. high productivity software for complex networks," *Webová strá nka https://networkx. lanl. gov/wiki*, 2013.

[29] B. Dezső, A. Jüttner, and P. Kovács, "Lemon–an open source c++ graph template library," *Electronic Notes in Theoretical Computer Science*, vol. 264, no. 5, pp. 23–45, 2011.

[30] A. Ching, S. Edunov, M. Kabiljo, D. Logothetis, and S. Muthukrishnan, "One trillion edges: Graph processing at facebook-scale," *Proceedings of the VLDB Endowment*, vol. 8, no. 12, pp. 1804–1815, 2015.

[31] X. Zhu, W. Han, and W. Chen, "Gridgraph: Large-scale graph processing on a single machine using 2-level hierarchical partitioning," in *2015 {USENIX} Annual Technical Conference ({USENIX}{ATC} 15)*, 2015, pp. 375–386.

[32] A. Kyrola, G. Blelloch, and C. Guestrin, "Graphchi: Large-scale graph computation on just a {PC}," in *Presented as part of the 10th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 12)*, 2012, pp. 31–46.

[33] A. Roy, I. Mihailovic, and W. Zwaenepoel, "X-stream: Edge-centric graph processing using streaming partitions," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, 2013.

[34] S. Maass, C. Min, S. Kashyap, W. Kang, M. Kumar, and T. Kim, "Mosaic: Processing a

trillion-edge graph on a single machine," in *Proceedings of the Twelfth European Conference on Computer Systems*. ACM, 2017, pp. 527–543.

[35] P. Kipfer, M. Segal, and R. Westermann, "Uberflow: a gpu-based particle engine," in *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*. ACM, 2004, pp. 115–122.

[36] H. Liu and H. H. Huang, "Graphene: Fine-grained {IO} management for graph computing," in *15th {USENIX} Conference on File and Storage Technologies ({FAST} 17)*, 2017, pp. 285–300.

[37] P. Kumar and H. H. Huang, "G-store: high-performance graph store for trillion-edge processing," in *SC'16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2016, pp. 830–841.

[38] Z. Ai, M. Zhang, Y. Wu, X. Qian, K. Chen, and W. Zheng, "Squeezing out all the value of loaded data: An out-of-core graph processing system with reduced disk i/o," in *2017 {USENIX} Annual Technical Conference ({USENIX}{ATC} 17)*, 2017, pp. 125–137.

[39] R. Pearce, M. Gokhale, and N. M. Amato, "Multithreaded asynchronous graph traversal for in-memory and semi-external memory," in *SC'10: Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2010, pp. 1–11.

[40] Y. Xing, Y. Feng, S. Yu, Z. Chen, F. Liu, and N. Xiao, "Hpgraph: A high parallel graph processing system based on flash array," in *2016 IEEE 18th International Conference on High Performance Computing and Communications; IEEE 14th International Conference on Smart City; IEEE 2nd International Conference on Data Science and Systems (HPCC/SmartCity/DSS)*. IEEE, 2016, pp. 497–505.

[41] P. Sun, Y. Wen, T. N. B. Duong, and X. Xiao, "Graphmp: An efficient semi-external-memory

big graph processing system on a single machine," in *2017 IEEE 23rd International Conference on Parallel and Distributed Systems (ICPADS)*. IEEE, 2017, pp. 276–283.

[42] A. McCallum, K. Nigam, and L. H. Ungar, "Efficient clustering of high-dimensional data sets with application to reference matching," in *Proceedings of the sixth ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, 2000, pp. 169–178.

[43] J. Dean and S. Ghemawat, "MapReduce: Simplified data processing on large clusters," in *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, 2004.

[44] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster computing with working sets." *HotCloud*, vol. 10, pp. 10–10, 2010.

[45] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," in *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*. USENIX Association, 2012, pp. 2–2.

[46] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, "Scikit-learn: Machine learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

[47] L. Mouselimis, *ClusterR: Gaussian Mixture Models, K-Means, Mini-Batch-Kmeans, K-Medoids and Affinity Propagation Clustering*, 2018, r package version 1.1.7. [Online]. Available: https://CRAN.R-project.org/package=ClusterR

[48] R. R. Curtin, J. R. Cline, N. P. Slagle, W. B. March, P. Ram, N. A. Mehta, and A. G. Gray, "Mlpack: A scalable c++ machine learning library," *Journal of Machine Learning Research*, vol. 14, no. Mar, pp. 801–805, 2013.

[49] M. Shindler, A. Wong, and A. W. Meyerson, "Fast and accurate k-means for large datasets," in *Advances in neural information processing systems*, 2011, pp. 2375–2383.

[50] D. Sculley, "Web-scale k-means clustering," in *Proceedings of the 19th international conference on World wide web*. ACM, 2010, pp. 1177–1178.

[51] Y. Ding, Y. Zhao, X. Shen, M. Musuvathi, and T. Mytkowicz, "Yinyang k-means: A drop-in replacement of the classic k-means with consistent speedup," in *Proceedings of the 32nd International Conference on Machine Learning (ICML-15)*, 2015, pp. 579–587.

[52] J. Abello, A. L. Buchsbaum, and J. R. Westbrook, "A functional approach to external graph algorithms," in *Algorithmica*. Springer-Verlag, 1998, pp. 332–343.

[53] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin, "Powergraph: Distributed graph-parallel computation on natural graphs," in *Presented as part of the 10th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 12)*, 2012, pp. 17–30.

[54] H. Kwak, C. Lee, H. Park, and S. Moon, "What is twitter, a social network or a news media?" in *Proceedings of the 19th International Conference on World Wide Web*, 2010.

[55] L. Page, S. Brin, R. Motwani, and T. Winograd, "The pagerank citation ranking: Bringing order to the web." Stanford InfoLab, Tech. Rep., 1999.

[56] U. Brandes, "A faster algorithm for betweenness centrality," *Journal of mathematical sociology*, vol. 25, no. 2, pp. 163–177, 2001.

[57] V. D. Blondel, J.-L. Guillaume, R. Lambiotte, and E. Lefebvre, "Fast unfolding of communities in large networks," *Journal of statistical mechanics: theory and experiment*, vol. 2008, no. 10, p. P10008, 2008.

[58] J. Wu, Y. Wang, Z. Wu, Z. Wang, A. Veeraraghavan, and Y. Lin, "Deep k-means: Re-training and parameter sharing with harder cluster assignments for compressing deep convolutions," *arXiv preprint arXiv:1806.09228*, 2018.

[59] M. Caron, P. Bojanowski, A. Joulin, and M. Douze, "Deep clustering for unsupervised learning of visual features," in *Proceedings of the European Conference on Computer Vision (ECCV)*, 2018, pp. 132–149.

[60] M. Muja and D. G. Lowe, "Scalable nearest neighbor algorithms for high dimensional data," *IEEE transactions on pattern analysis and machine intelligence*, vol. 36, no. 11, pp. 2227–2240, 2014.

[61] A. Coates and A. Y. Ng, "Learning feature representations with k-means," in *Neural networks: Tricks of the trade*. Springer, 2012, pp. 561–580.

[62] A. Coates, A. Ng, and H. Lee, "An analysis of single-layer networks in unsupervised feature learning," in *Proceedings of the fourteenth international conference on artificial intelligence and statistics*, 2011, pp. 215–223.

[63] B. Yang, X. Fu, N. D. Sidiropoulos, and M. Hong, "Towards k-means-friendly spaces: Simultaneous deep learning and clustering," in *Proceedings of the 34th International Conference on Machine Learning-Volume 70*. JMLR. org, 2017, pp. 3861–3870.

[64] D. Arthur and S. Vassilvitskii, "k-means++: The advantages of careful seeding," in *Proceedings of the eighteenth annual ACM-SIAM symposium on Discrete algorithms*. Society for Industrial and Applied Mathematics, 2007, pp. 1027–1035.

[65] B. Bahmani, B. Moseley, A. Vattani, R. Kumar, and S. Vassilvitskii, "Scalable k-means++," *Proceedings of the VLDB Endowment*, vol. 5, no. 7, pp. 622–633, 2012.

[66] J. Yoder and C. E. Priebe, "Semi-supervised k-means++," *arXiv preprint arXiv:1602.00360*, 2016.

[67] I. S. Dhillon and D. S. Modha, "Concept decompositions for large sparse text data using clustering," *Machine learning*, vol. 42, no. 1-2, pp. 143–175, 2001.

[68] D. Pelleg, A. W. Moore *et al.*, "X-means: Extending k-means with efficient estimation of the number of clusters." in *Icml*, vol. 1, 2000, pp. 727–734.

[69] G. Hamerly and C. Elkan, "Learning the k in k-means," in *Advances in neural information processing systems*, 2004, pp. 281–288.

[70] C. F. Gauss, *Theory of the motion of the heavenly bodies moving about the sun in conic sections: a translation of Carl Frdr. Gauss" Theoria motus": With an appendix. By Ch. H. Davis.* Little, Brown and Comp., 1857.

[71] J. C. Bezdek, R. Ehrlich, and W. Full, "Fcm: The fuzzy c-means clustering algorithm," *Computers & Geosciences*, vol. 10, no. 2-3, pp. 191–203, 1984.

[72] L. Kaufman and P. J. Rousseeuw, "Clustering large applications (program clara)," *Finding groups in data: an introduction to cluster analysis*, pp. 126–146, 2008.

[73] G. Schwarz *et al.*, "Estimating the dimension of a model," *The annals of statistics*, vol. 6, no. 2, pp. 461–464, 1978.

[74] T. W. Anderson, T. W. Anderson, T. W. Anderson, T. W. Anderson, and E.-U. Mathématicien, *An introduction to multivariate statistical analysis.* Wiley New York, 1958, vol. 2.

[75] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran, "Cache-oblivious algorithms," in *Foundations of Computer Science, 1999. 40th Annual Symposium on.* IEEE, 1999, pp. 285–297.

[76] A. Inc. Amazon web services. [Online]. Available: https://aws.amazon.com

[77] M. P. Forum, "Mpi: A message-passing interface standard," Knoxville, TN, USA, Tech. Rep., 1994.

[78] "Frienster graph," https://archive.org/download/friendster-dataset-201107, Accessed 4/18/2014.

[79] MATLAB, *version 7.10.0 (R2010a)*. Natick, Massachusetts: The MathWorks Inc., 2010.

[80] C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh, "Basic linear algebra subprograms for fortran usage," *ACM Transactions on Mathematical Software (TOMS)*, vol. 5, no. 3, pp. 308–323, 1979.

[81] R Core Team, *R: A Language and Environment for Statistical Computing*, R Foundation for Statistical Computing, Vienna, Austria, 2015. [Online]. Available: https://www.R-project.org/

[82] ——, *R: A Language and Environment for Statistical Computing*, R Foundation for Statistical Computing, Vienna, Austria. [Online]. Available: https://www.R-project.org

Vita

Disa Mhembere received his Bachelors Degree from Morgan State University in 2010. He received both his masters in engineering management (2013) and masters in computer science (2015) from Johns Hopkins University. His research interests include scalable framework and library development for scientific computing and machine learning. His Ph.D. research focus was on the development of graph analytics systems and clustering frameworks.