knor: A NUMA-Optimized In-Memory, Distributed and Semi-External-Memory k-means Library

Disa Mhembere
Dept. of Computer Science,
Johns Hopkins University
Baltimore, Maryland

Da Zheng
Dept. of Computer Science,
Johns Hopkins University
Baltimore, Maryland

Carey E. Priebe
Dept. of Applied Math and Statistics,
Johns Hopkins University
Baltimore, Maryland

Joshua T. Vogelstein
Institute for Computational Medicine,
Dept. of Biomedical Engineering,
Johns Hopkins University
Baltimore, Maryland

Randal Burns
Dept. of Computer Science,
Johns Hopkins University
Baltimore, Maryland

ABSTRACT

k-means is one of the most influential and utilized machine learning algorithms. Its computation limits the performance and scalability of many statistical analysis and machine learning tasks. We rethink and optimize k-means in terms of modern NUMA architectures to develop a novel parallelization scheme that delays and minimizes synchronization barriers. The *k-means NUMA Optimized Routine* (knor) library has (i) in-memory (knori), (ii) distributed memory (knord), and (ii) semi-external memory (knors) modules that radically improve the performance of k-means for varying memory and hardware budgets. knori boosts performance for single machine datasets by an order of magnitude or more. knors improves the scalability of k-means on a memory budget using SSDs. knors scales to billions of points on a single machine, using a fraction of the resources that distributed in-memory systems require. knord retains knori's performance characteristics, while scaling in-memory through distributed computation in the cloud. knor modifies Elkan's triangle inequality pruning algorithm such that we utilize it on billion-point datasets without the significant memory overhead of the original algorithm. We demonstrate knor outperforms distributed commercial products like H_2O, Turi (formerly Dato, GraphLab) and Spark's MLlib by more than an order of magnitude for datasets of 10^7 to 10^9 points.

KEYWORDS

NUMA, k-means, semi-external memory, cloud, clustering, parallel

ACM Reference format:
Disa Mhembere, Da Zheng, Carey E. Priebe, Joshua T. Vogelstein, and Randal Burns. 2017. knor: A NUMA-Optimized In-Memory, Distributed and Semi-External-Memory k-means Library. In *Proceedings of HPDC '17, Washington , DC, USA, June 26-30, 2017,* 12 pages.
DOI: http://dx.doi.org/10.1145/3078597.3078607

1 INTRODUCTION

Clustering data to maximize within-cluster similarity and cross-cluster variance is highly desirable for the analysis of big data. K-means is an intuitive and highly popular method of clustering n points in d-dimensions into k clusters. A very popular synchronous variant of k-means is Lloyd's algorithm [22]. Similar to Expectation Maximization [8], Lloyd's algorithm proceeds in two phases. In phase one, we compute 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 4). Despite k-means' popularity, state-of-the-art machine learning libraries [16, 23, 28] experience many challenges scaling performance well with respect to growing data sets. Furthermore, these libraries place an emphasis on scaling-out computation to the distributed setting, neglecting to fully utilize the resources within each machine.

The decomposition of extremely large datasets into clusters of data points that are similar is a topic of great interest in industry and academia. For example, clustering is the backbone upon which popular user recommendation systems at Netflix [3] are built. Furthermore, partitioning multi-billion data points is essential to targeted ad-driven organizations such as Google [6] and Facebook [38]. In addition, clustering is highly applicable to neuroscience and genetics research. Connectomics [4, 24, 25], uses clustering to group anatomical regions by structural, physiological, and functional similarity, for the purposes of inference. Behavioromics [39] uses clustering to map neurons to distinct motor patterns. In genetics, clustering is used to infer relationships between genetically similar species [20, 30].

The greatest challenges facing tool builders are *(i)* reducing the cost of the synchronization barrier between the first and second phase of k-means, *(ii)* mitigating the latency of data movement through the memory hierarchy, and *(iii)* scaling to arbitrarily large datasets, while maintaining performance. In addition, fully asynchronous computation for Lloyd's algorithm is infeasible because each iteration updates global state, membership and centroids. 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 frameworks [23, 28, 29] 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 with regards to k-means. This design incurs heavy network traffic owing to data shuffling and centralized master-worker designs. Furthermore, such frameworks struggle to capitalize on potential gains from the use of computation pruning techniques, such as Elkan's triangle inequality algorithm (TI) [11]. Pruning introduces skew in which few workers have the bulk of the computation. Skew degrades parallelism. While skew can be dealt with through dynamic scheduling, this incurs data movement and message passing overheads.

In contrast, our k-means prefers scale-up computation on shared-memory multicore machines in order to eliminate network traffic and perform fine-grained synchronization. A current trend for hardware design scales up a single machine, rather than scaling out to many networked machines, integrating large memories and using solid-state storage devices (SSDs) to extend memory capacity. This conforms to the node design for supercomputers [2]. Recent findings, from Frank McSherry [27, 35] and our prior work [45], show that the largest 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, rather than a distributed compute engine. Our findings reveal that clustering has the same structure. We perform k-means on a single or few machines to minimize network bottlenecks and find that even our single node performance (with SSDs) outperforms our competitor's distributed performance on many instances.

Our approach advances Lloyd's algorithm for modern, multicore NUMA architectures to achieve a high degree of parallelism by significantly merging the two phases in Lloyd's. We present knori, a fast in-memory, module that performs several orders of magnitude faster than other state-of-the-art systems for datasets that fit into main memory. We implement a practical modification to TI, that we call the minimal triangle inequality (MTI). TI incurs a memory increment of $O(nd)$, limiting its scalability. MTI requires an increase of only $O(n)$ memory, drastically improving its utility for large-scale datasets. In practice, MTI outperforms TI because it requires significantly less data structure maintenance, while still pruning computation comparably. knor clusters data an order of magnitude faster than competitors.

The knord distributed module builds directly on knori and runs across multiple machines using a decentralized driver. It runs on larger datasets that fit in the aggregate memory of multiple nodes.

knors is a semi-external memory implementation that scales the computation of a single node beyond memory bounds. Semi-external memory (SEM) k-means holds $O(n)$ data in memory while streaming $O(nd)$ data from disk for a dataset, $\vec{V} \in \mathbb{R}^{n \times d}$. This notion of SEM is analogous to the definition of SEM for graph algorithms [1, 31] in which vertex state is kept in memory and edge lists on disk. We build knors on a modified FlashGraph [45] framework to access asynchronous I/O and overlap I/O and computation. knors uses a fraction of the memory of popular frameworks and outperforms them by large factors using less hardware.

This work demonstrates that k-means on extremely large datasets can be run on increasingly smaller/fewer machines than possible before; creating reductions in monetary expense and power consumption. Furthermore, our routines are highly portable. We simply require the C++11 standard library be available, with thread-level parallelism is implemented using the POSIX thread (p-threads) library. Distributed routines rely on the Message Passing Library, MPI [12]. The I/O components for SEM routines are implemented using low-level Linux interfaces.

2 RELATED WORK

Zhao et al [43] developed a parallel k-means routine on Hadoop!, an open source implementation of MapReduce [7]. Their implementation has much in common with Mahout [29], a machine learning library for: Hadoop!. The Map-Reduce paradigm consists of a `Map` phase, a synchronization barrier in which data are shuffled to *reducers*, then a `Reduce` phase. For k-means one would perform distance computations in the `Map` phase and build centroids to be used in the following iteration in `Reduce` phase. The model allows for effortless scalability and parallelism, but little flexibility in how to achieve either. As such, the implementation is subject to skew within the reduce phase as data points assigned to the same centroid end up at a single *reducer*.

MLlib is a machine learning library for Spark [42]. Spark imposes a functional paradigm to parallelism allowing for delayed 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 resilient distributed datasets [41].

Other works focus on developing fast k-means approximations. Sophia-ML uses a mini-batch algorithm that uses sampling to reduce the cost of Lloyd's algorithm (also referred to as batched k-means) and stochastic gradient descent k-means [36]. Sophia-ML's target application is online, real-time applications, which differs from our goal of exact k-means on large scale data. Shindler et al [37] 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 knors, but is designed for a single processor, passing over the data just once and operating on medium-sized data. We avoid approximations; they see little widespread use owing to questions of cluster quality.

Elkan proposed the use of the triangle inequality (TI) with bounds [11], to reduce the number of distance computations to fewer than $O(kn)$ 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 $O(nk)$. Yinyang k-means [9] develop a competitor pruning technique to TI that maintains a lower-bound matrix of size $O(nt)$, where 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. We propose MTI for computation pruning. MTI costs a constant $O(n)$ more memory making it practical for use with big-data.

FlashGraph [45] 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. FlashGraph overlaps I/O with computation to mask latency in data movement through the memory hierarchy. FlashGraph is also tolerant to in-memory failures, allowing recovery in SEM routines through lightweight checkpointing.

FlashGraph is built on top of a userspace filesystem called SAFS [44]. SAFS provides a framework to perform high speed I/O from an array of SSDs. To facilitate this, both SAFS and FlashGraph work to merge I/O requests for multiple 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 creates and manages a *page cache* that pins frequently touched pages in memory. The page cache reduces the number of actual I/O requests made to disk. Section 6.1 discusses the modifications we make to FlashGraph to build knors on top of FlashGraph.

3 NOMENCLATURE

We define notation that we use throughout the manuscript. 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-dimension vector in the dataset $\vec{V}$ with cardinality, $|\vec{V}| = n$. Let j be the number of iterations of Lloyd's algorithm we perform in a single run of k-means. Let $t \in \{0...j\}$ be the current iteration within a run of k-means. 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 $\vec{c}^{\,t}$. We use Euclidean distance, denoted as $\mathbf{d}$, as the dissimilarity metric between any $\vec{v}$ and $\vec{c}^{\,t}$, such that $\mathbf{d}(\vec{v}, \vec{c}^{\,t}) = \sqrt{(\vec{v}_1 - \vec{c}^{\,t}_1)^2 + (\vec{v}_2 - \vec{c}^{\,t}_2)^2 + ... + (\vec{v}_{d-1} - \vec{c}^{\,t}_{d-1})^2 + (\vec{v}_d - \vec{c}^{\,t}_d)^2}$.

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.

4 PARALLEL LLOYD'S ALGORITHM

Underlying further optimizations is our parallel version of Lloyd's algorithm (||Lloyd's) that boosts the performance of knor and 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:

(1) Phase I: Compute the nearest centroid, $\vec{c\_nearest}^{\,t}$ 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 eliminates 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 $O(T)$ over Lloyd's. This algorithm design naturally leads to lock-free routines that require fewer synchronization barriers as we show in Algorithm 1.

Algorithm 1 || Lloyd's algorithm

```
 1: procedure ||MEANS($\vec{V}$, $\vec{C^t}$, $k$)
 2:     $\vec{ptC^t}$                                        ▷ Per-thread centroids
 3:     $\vec{clusterAssignment}^t$                           ▷ Shared, no conflict
 4:     tid                                                   ▷ Current thread ID
 5:     parfor $\vec{v} \in \vec{V}$ do
 6:         $dist = \infty$
 7:         $\vec{c\_nearest}^t$ = 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:                 $\vec{c\_nearest}^t = \vec{c}^{\,t}$
12:             end if
13:         end for
14:         $\vec{ptC^t}[tid][\vec{c\_nearest}^t]$ += $\vec{v}$
15:     end parfor
16:     clusterMeans = MERGEPTSTRUCTS($\vec{ptC^t}$)
17: end procedure

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

Minimal Triangle Inequality (MTI) Pruning

We simplify Elkan's Algorithm for triangle inequality pruning (TI) [11] by removing the the necessity for the lower bound matrix of size $O(nd)$. Omitting the lower bound matrix means we forego the opportunity to prune certain computations; we accept this tradeoff in order to limit main memory consumption and prioritize usability. Due to space limitations, we omit experiments exhibiting that in practice pruning benefits of maintaining a lower bound matrix are minimal. With $O(n)$, memory we implement three of the four [11] pruning clauses invoked for each data point in an iteration of a knor module with pruning *enabled*. Let $u^t = \mathbf{d}(\vec{v}, c\_\vec{nearest}^t) + f(c\_\vec{nearest}^t)$, be the upper bound of the distance of a sample, $\vec{v}$, in iteration t from its assigned cluster $c\_\vec{nearest}^t$. 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 $u^t \leq \min \mathbf{d}(c\_\vec{nearest}^t, \vec{c}^t \,\forall \vec{c}^t \in \vec{C}^t)$, then $\vec{v}$ remains in the same cluster for the current iteration. For knors, this is extremely significant because no I/O request is made for data.

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

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

5 IN-MEMORY DESIGN

We prioritize practical performance when we implement knori optimizations. We make design tradeoffs to balance the opposing forces of minimizing memory usage and maximizing CPU cycles spent on parallel computing. Section 5 chronicles the memory bounds that we achieve and optimizations that we apply.

5.1 Asymptotic Memory Consumption

knori with MTI *disabled*, which we call knori-, retains the computation complexity of a serial routine, i.e., $O(ndk)$, but has a memory bound of $O(nd + Tkd)$ as compared to the original $O(nd + kd)$. The factor of T arises from the per-thread centroids we maintain. knori (MTI *enabled*) uses additional memory $O(k^2 + n)$, which does not increase the asymptotic bound. The $O(k^2)$ comes from maintaining an upper/lower triangular centroid-to-centroid distance matrix and $O(n)$ comes from maintaining the upper bound of each data point's distance to any centroids. The $O(n)$ in practice adds between 6-10 Bytes per data point or $\leq 1GB$ when $n = 100$ million and d is unrestricted. We justify the tradeoff of slightly higher memory consumption for an improvement in performance in Section 8. A complete summary of knor routine memory bounds is shown in Table 1.

5.2 In-memory optimizations

The following design principles and optimizations improve the performance of Algorithm 1.

Prioritize data locality for NUMA: Non-uniform memory access (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 interconnect. The effect is low latency accesses with high throughput to local memory banks, and conversely 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 results. Figure 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.

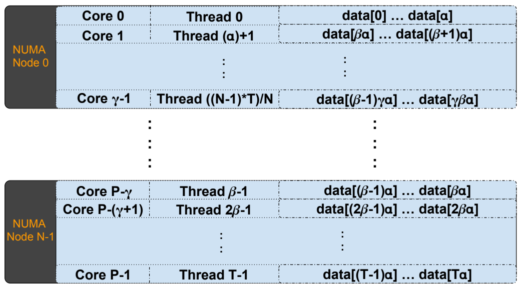


Figure 1: The memory allocation and thread assignment scheme we utilize for knori and knord on each machine. $\alpha = n/T$ is the amount of data per thread, $\beta = T/N$ is the number of threads per NUMA node, and $\gamma = P/N$ is the number of physical processors per NUMA node. Distributing memory across NUMA nodes maximizes memory throughput while binding threads to NUMA nodes reduces remote memory accesses.

Dynamic Scheduling and Work Stealing: To achieve optimal performance when MTI pruning is *disabled*, statically scheduling thread tasks to locally allocated data partitions is sufficient. When MTI is *enabled*, we see worker skew at the thread level. In response, we develop a NUMA-aware partitioned priority task queue, (Figure 2), to feed worker threads, prioritizing tasks that maximize local memory access and thus 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 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 when pruning in addition to optimized memory access patterns.

Avoid interference and delay the synchronization barrier: We employ per-thread local centroids and write-conflict free shared data structures to eliminate interference. Local centroids are unfinalized running totals of global centroids used in the following iteration for distance computations. Local centroids are concurrently updated. Finally, we require only a single global barrier prior to merging local clusters in a parallel funnelsort-like [14] reduction routine for use in the following iteration.

Effective data layout for CPU cache exploitation: Both per-thread and global data structures are contiguously allocated chunks of memory. Contiguous data organization and sequential access patterns when computing cluster-to-centroid distances maximizes both prefetching and CPU caching opportunities.

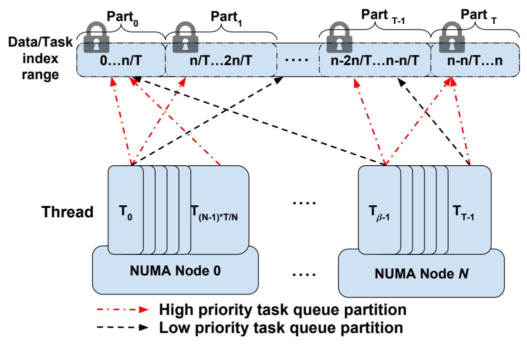


Figure 2: The NUMA-aware partitioned task scheduler we utilize for knori and knord on each machine. The scheduler minimizes task queue lock contention and remote memory accesses by prioritizing tasks with data in the local NUMA memory bank.

6 SEMI-EXTERNAL MEMORY DESIGN

We design a highly optimized semi-external memory module, knors, that targets the stand-alone server environment and when data are too large to reside fully in main memory. knors extends its in-memory counterpart from the perspective of scalability by placing data on SSDs and requesting data as necessary. Data requests are performed asynchronously allowing the overlap of I/O and computation. The SEM model allows us to reduce the asymptotic memory bounds so as to scale to larger datasets. A SEM routine uses $O(n)$ memory for a dataset, $\vec{V} \in \mathbb{R}^{nxd}$ that when processed completely in memory would require $O(nd)$ memory. For completeness, we summarize all knor memory asymptotic bounds in Table 1.

6.1 FlashGraph modifications

We modify FlashGraph 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 FlashGraph's I/O layer to support reading floating point row-data from SSD rather than the numeric data type associated with edge lists. The `page_row` type computes it 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 $O(n)$ because it does not store a row-data index to data on SSDs unlike a `page_vertex`. This allows knors to scale to larger datasets than possible before on a single machine.

6.2 Semi-external Memory Asymptotic Analysis

SEM implementations do not alter the computation bounds, whereas they do lower the memory bounds for k-means to $O(n + Tkd)$. This improves on the $O(nd + Tkd)$ bound of knori and the $O(nd + kd)$ bound of Lloyd's original algorithm. In practice, the disk I/O bound of knors is much lower than the worst case of $O(nd)$ obtained when MTI pruning is *disabled* (knors-), especially for data with natural clusters.

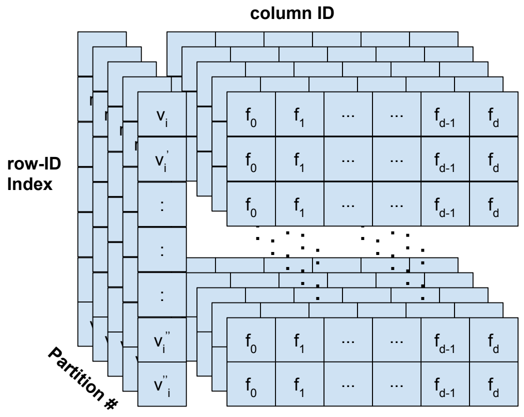


Figure 3: The structure of the row cache we utilize for knors. Partitioning the row cache eliminates the need for locking during cache population.

6.2.1 *I/O minimization.*

I/O bounds the performance of k-means in the SEM model. Accordingly, we reduce the number of data-rows that need to be brought into memory each iteration. Only Clause 1 of MTI (Section 4) facilitates the skipping of all distance computations for a data point. In this case we do not issue an I/O request for the data point's row-data. This results in a reduction in I/O, but because data are pruned in a near-random fashion, we retrieve significantly more data than necessary from SSDs due to fragmentation. 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 higher amounts of I/O requests, offsetting any gains achieved by the reduction in fragmentation. We utilize a minimum read size of $4KB$; even with this relatively small value we still receive significantly more data from disk than we request (Figure 6b). To address this, we develop a lazily-updated partitioned *row cache* described in Section 6.2.2, that drastically reduces the amount of data brought into memory as shown in Figure 6a.

6.2.2 *Partitioned Row Cache (RC).*

We add a layer to the memory hierarchy for SEM applications by designing a lazily-updated row cache (Figure 3). 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 its row-data. 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.

The row cache lazily updates at certain iterations of the k-means algorithm based on a user defined *cache update interval* (I_{cache}). The cache updates/refreshes at iteration I_{cache} then the update frequency increases exponentially such that the next RC update is performed after $2I_{cache}$ iterations and so forth. This means that row-data in the RC remains static for several iterations before the RC is flushed then repopulated. We justify lazy updates by observing that k-means, especially on real-world data, follows predictable row activation patterns. In early iterations the cache's utility is of minimal benefit as the row activation pattern is near-random. As the algorithm progresses, data points that are active tend to stay active for many iterations as they are near more than one established centroid. This means the cache can remain static for longer periods of time while achieving very high cache hit rates. We set I_{cache} to 5 for all experiments in the evaluation (Section 8). The design trade-off is cache freshness for reduced cache maintenance overhead. We demonstrate the efficacy of this design in Figure 7.

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. At the completion of an iteration in which the RC refreshes, each partition updates a global map that stores pointers to the actual data. The size of the cache is user-defined, but $1GB$ is sufficient to significantly improve the performance of billion-point datasets.

7 DISTRIBUTED DESIGN

knord scales to distributed clusters through the Message Passing Interface (MPI). We employ modular design principles and build our distributed routines as a layer above our parallel in-memory routines. The implication is that each machine maintains a decentralized *driver* (MPI) process that launches *worker* (pthread) threads that retain the NUMA performance optimizations we develop for knori in Section 5.

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 implement. We further argue that the gains in performance of our data partitioning scheme (shown in Figure 1) outweigh the effects of skew in this setting. We validate these assertions empirically in Section 8.9.

8 EXPERIMENTAL EVALUATION

We evaluate knor by benchmarking optimizations in addition to comparing its performance against other state-of-the-art frameworks. In Section 8.3 we evaluate the performance of our baseline single threaded implementation to ensure all speedup experiments are relative to a state-of-the-art baseline performance. Sections 8.4 and 8.5 evaluate the effect of specific optimizations on our in-memory and semi-external memory tools respectively. Section 8.7 evaluates the performance of knori and knors relative to other popular frameworks from the perspective of time and resource consumption. Section 8.9 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 8.7). The datasets we use for performance and scalability evaluation are shown in Table 2.

We use the following notation throughout the evaluation:

- **knori-** is knori with MTI pruning *disabled.*
- **knors-** is knors with MTI pruning *disabled.*
- **knors--** is knors with both MTI pruning and the row cache (RC) *disabled.*
- **knord-** is knord with MTI pruning *disabled.*
- **MLlib-EC**2 is MLlib's k-means routine running on Amazon EC2 instances [18].
- **MPI** is a pure MPI [12] distributed implementation of our ||Lloyd's algorithm (Section 4) with MTI pruning. We develop this in order to compare its performance to knord.
- **MPI-** is a pure MPI distributed implementation of our ||Lloyd's algorithm with MTI pruning *disabled.*

Table 1: Asymptotic memory complexity of knor routines.

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

Table 2: The datasets under evaluation in this study.

Data Matrix	n	d	Size
Friendster-8 [13] eigenvectors	66M	8	4GB
Friendster-32 [13] 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.

8.1 Single Node Evaluation Hardware

We perform single node experiments relating to knori, knors 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 v3.13.0. The C++ code is compiled using mpicxx.mpich2 version 4.8.4 with the -O3 flag.

8.2 Cluster Evaluation Hardware

We perform distributed memory experiments relating to knord 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.

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 [26] and BLAS [21]. We find the same to be true of popular statistics packages and frameworks such as MLpack [5], Scikit-learn [32] and R [33] all of which use highly optimized C/C++ code, although some use scripting language wrappers. Table 3 shows performance at 1 thread. Table 3 provides credence to our speedup results since our baseline single threaded performance tops other state-of-the-art serial routines.

Table 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

8.4 In-memory Optimization Evaluation

We show NUMA-node thread binding, maintaining NUMA memory locality, and NUMA-aware task scheduling for knori is highly effective in improving performance. We achieve near-linear speedup (Figure 4). 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 linearly, the NUMA-oblivious routine has a lower linear constant when compared with a NUMA-aware implementation (Figure 4).

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 T increases;
- a dynamic NUMA-oblivious task scheduler may give tasks to threads that cause worker threads to perform many more remote memory accesses than necessary when thread-binding and static scheduling are employed.

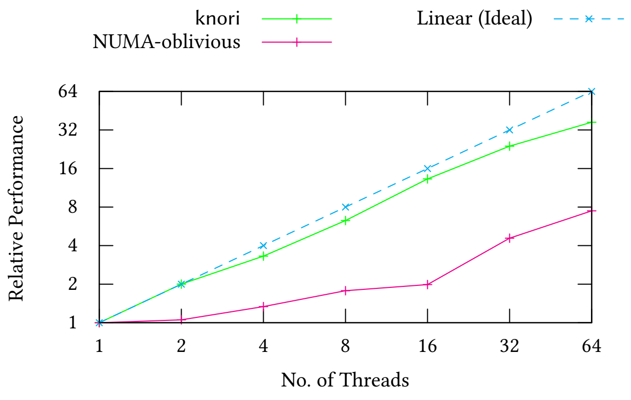


Figure 4: Speedup of knori (which is NUMA-aware) vs. a NUMA-oblivious routine for on the Friendster top-8 eigenvector dataset, with $k = 10$.

We demonstrate the effectiveness of a NUMA-aware partitioned task scheduler for pruned computations via knori (Figure 5). 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. 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 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 results for space reasons and to avoid redundancy.

8.5 Semi-external Memory Evaluation

We evaluate knors optimizations, performance and scalability. knors utilizes a 4KB FlashGraph *page cache* size, minimizing 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 6a 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 6b 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

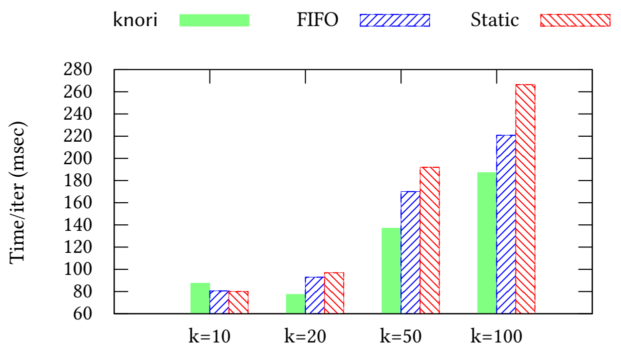
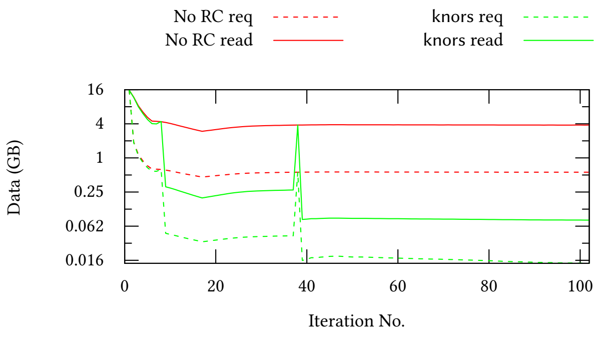
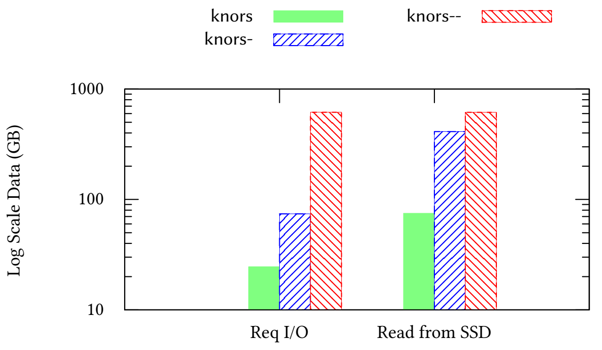


Figure 5: Performance of the partitioned NUMA-aware scheduler (default for knori) vs. FIFO and static scheduling for knori on the Friendster-8 dataset.



(a) knors data requested (req) from SSDs vs. data read (read) from SSDs each iteration when the row cache (RC) was *enabled* or *disabled*. Because of MTI pruning, algorithms may request only a few points from any block, but the entire block must still be read from SSD.



(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 6: 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.

issued to FlashGraph for row-data is either served by FlashGraph's 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 6 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.

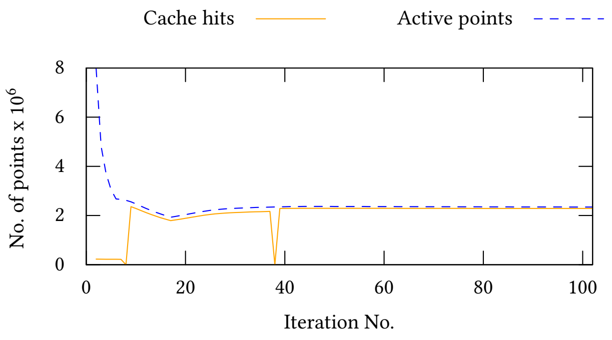


Figure 7: Row cache hits per iteration contrasted with the maximum achievable number of hits on the Friendster top-32 eigenvectors dataset.

Lazy row cache updates reduce I/O significantly. Figure 7 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.

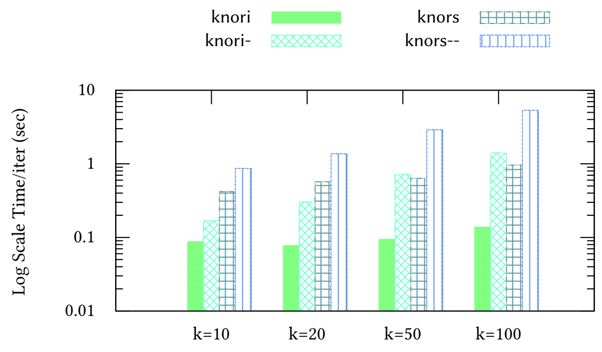
8.6 MTI Evaluation

Figures 8a and 8b 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 even without some of the pruning ability Elkan's TI [11] provides. Figure 8c 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.

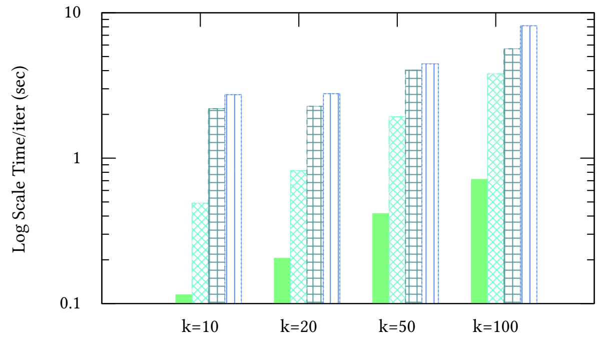
8.7 Comparison with Other Frameworks

We evaluate the performance of our routines in comparison with other frameworks on the datasets in Table 2. We show that knori achieves greater than an order of magnitude improvement over other state-of-the-art frameworks. Finally, we demonstrate knors outperforms other state-of-the-art frameworks by several factors.

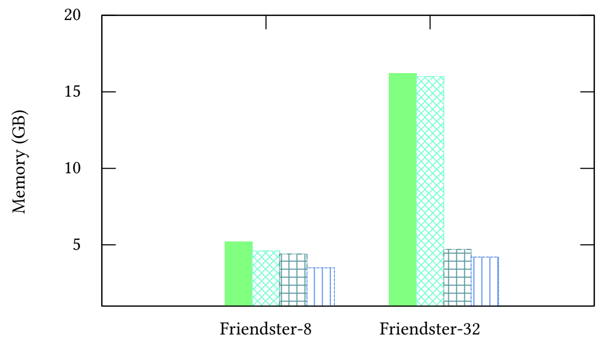
Both our in-memory and semi-external memory modules incur little memory overhead when compared with other frameworks. Figure 9c shows memory consumption. We note that MLlib requires the placement of temporary Spark block manager files. Because the block manager cannot be disabled, we provide an in-memory 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



(a) The Friendster graph top-8 eigenvector dataset.

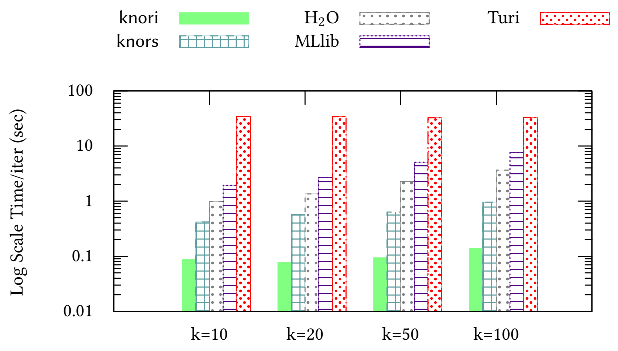


(b) The Friendster graph top-32 eigenvector dataset.

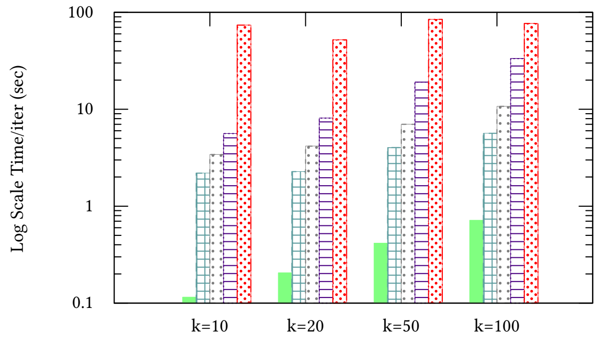


(c) Memory comparison of fully optimized knor routines (knori, knors) compared to more vanilla knor routines (knori-, knors--)

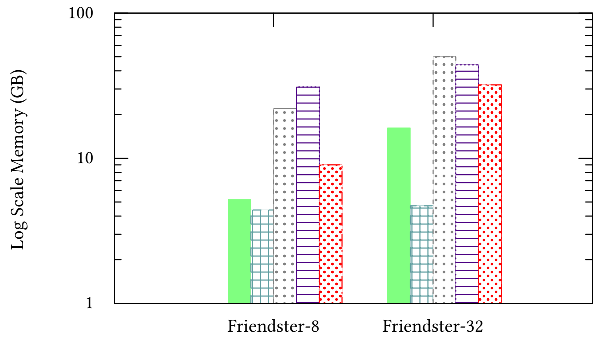
Figure 8: Performance and memory usage comparison of knor modules on matrices from the Friendster graph top-8 and top-32 eigenvectors.



(a) The Friendster graph top-8 eigenvector dataset.



(b) The Friendster graph top-32 eigenvector dataset.



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

Figure 9: Performance comparison on matrices from the Friendster [13] graph top-8 and top-32 eigenvectors.

comparisons. All measurements are an average of 10 runs; we drop all caches between runs.

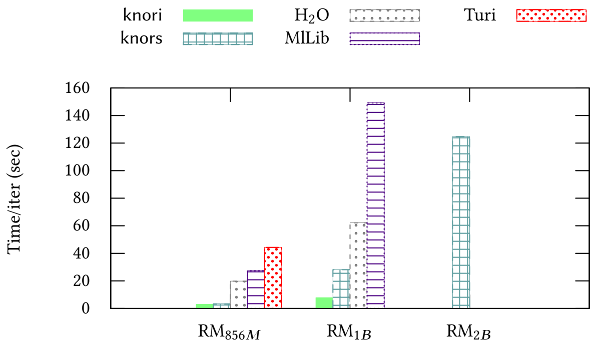
We demonstrate that knori is no less than an order of magnitude faster than all competitor frameworks (Figure 9). knori is often hundreds of times faster than Turi. Furthermore, knors is consistently twice as fast as competitor in-memory frameworks. We further demonstrate performance improvements over 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 8 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 8 and 9) 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.

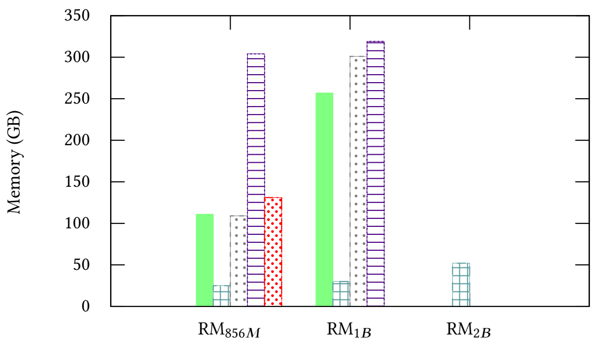
8.8 Single-node Scalability Evaluation

To demonstrate scalability, we compare performance 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 MLlib, H_2O and Turi. As data increases in size, the performance difference between knori and knors narrows since 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.



(a) Per iteration time elapsed of each routine.



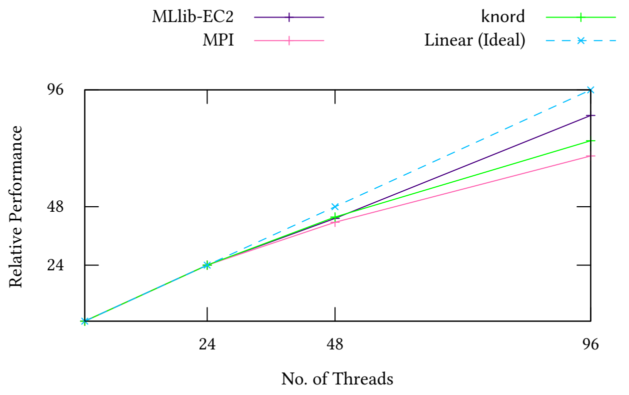
(b) Memory consumption of each routine.

Figure 10: 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.

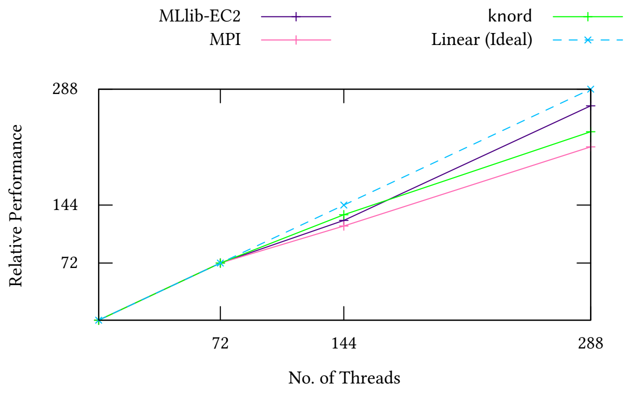
Memory capacity limits the scalability of k-means and semi-external memory allows algorithms to scale well beyond the limits of physical memory. The 1B point matrix (RM_{1B}) is the largest that fits in 1TB of memory on our machine. At 2B points (RU_{2B}), semi-external memory algorithms continue to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

8.9 Distributed Comparison vs. Other Frameworks

We analyze performance of knord and knord- on Amazon's EC2 cloud in comparison to that of (i) MLlib (**MLlib-EC**2), (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* (**MPI-**). Note that H_2O has no distributed memory implementation and Turi discontinued their distributed memory interface prior to our experiments.



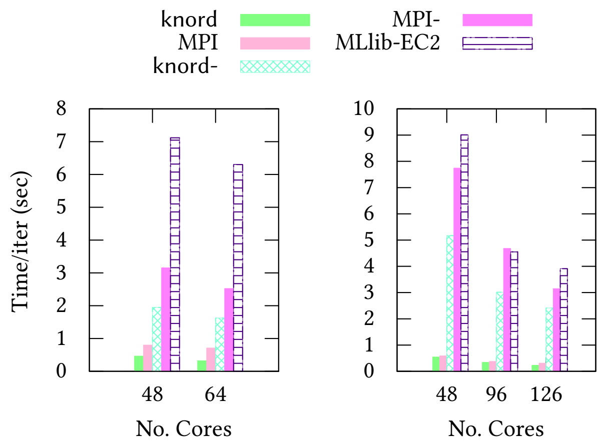
(a) Distributed speedup comparison on the Friendster-32 dataset.



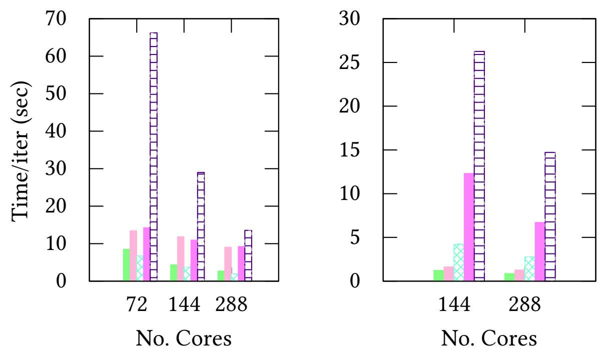
(b) Distributed speedup comparison on the RM_{1B} dataset.

Figure 11: Speedup experiments are normalized to each implementation's serial performance. Each machine has 18 physical cores with 1 thread per core.

Figures 11 and 12 reveal several fundamental and important results. Figure 11 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 organization push big-data computation to the cloud. Figure 12 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 12 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 12 shows that MTI remains a low-overhead, effective method to reduce computation even in the distributed setting.



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



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

Figure 12: Distributed performance comparison of knord, MPI and MLlib on Amazon's EC2 cloud. Each machine has 18 physical cores with 1 thread per core.

8.9.1 Semi-External Memory in the Cloud. We conclude our experiments 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 13 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.

9 FUTURE WORK AND DISCUSSION

We intend to expand the distributed aspects of knor into a generalized programming framework for distributed machine learning on NUMA machines. We observe that NUMA architectures are

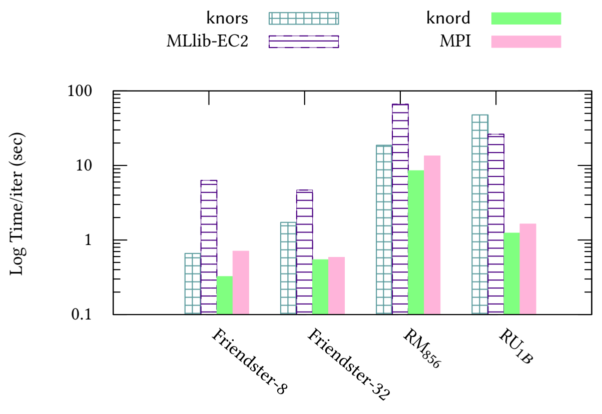


Figure 13: Performance comparison of knors to distributed packages. knors uses one i3.16xlarge machine with 32 physical cores. knord, MLlib-EC2 and MPI use 3 c4.8xlarge with a total of 48 physical cores for all datasets other than RU_{1B} where they use 8 c4.8xlarge with a total of 128 physical cores.

prevalent today in cloud computing. A large contingency of machines available through Unix-based cloud service providers such as Amazon EC2, IBM Cloud [46] and Google Cloud Platform [19] are indeed NUMA systems.

We aim to build a suite of machine learning algorithms on top of the generalized derivative framework. We aim to demonstrate that the NUMA optimizations we deploy for knor are applicable to a variety of compute-intensive applications. The initial phase will target other variants of k-means like spherical k-means [17], semi-supervised k-means++ [40] etc. Later phases will target machine learning algorithms like GMM [15], agglomerative clustering [34] and k-nearest neighbors [10]. Finally, we aim to provide a C++ interface upon which users may implement custom algorithms and benefit from our NUMA and external memory optimizations.

We are an open source project available at https://github.com/flashxio/knor

10 CONCLUSION

We accelerate k-means by over an order of magnitude by rethinking Lloyd's algorithm for modern multiprocessor NUMA architectures through the minimization of critical regions. We demonstrate that our modifications to Lloyd's are relevant to both in-memory (knori), distributed memory (knord) and semi-external memory (knors) applications as we outperform state-of-the-art frameworks running the exact same algorithms.

We formulate a minimal triangle inequality pruning technique (MTI) 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.

The addition of a row caching layer yields performance improvements over vanilla SEM implementations. We demonstrate that k-means in SEM performs only a small constant factor slower than in-memory algorithms for large scale datasets and scales beyond the limits of memory at which point in-memory algorithms fail.

Finally, we demonstrate that there are large performance benefits associated with NUMA-targeted optimizations. We show that data locality optimizations such as NUMA-node thread binding, NUMA-aware task scheduling, and NUMA-aware memory allocation schemes provide several times speedup for k-means on NUMA hardware.

11 ACKNOWLEDGMENTS

This work is partially supported by DARPA GRAPHS N66001-14-1-4028 and DARPA SIMPLEX program through SPAWAR contract N66001-15-C-4041.

REFERENCES

[1] J. Abello, A. L. Buchsbaum, and J. R. Westbrook. A functional approach to external graph algorithms. In *Algorithmica*, pages 332–343. Springer-Verlag, 1998.

[2] 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*, pages 25–32. IEEE, 2014.

[3] J. Bennett and S. Lanning. The netflix prize. In *Proceedings of KDD cup and workshop*, volume 2007, page 35, 2007.

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

[5] 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*, 14(Mar):801–805, 2013.

[6] 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*, pages 271–280. ACM, 2007.

[7] 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.

[8] 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)*, pages 1–38, 1977.

[9] 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)*, pages 579–587, 2015.

[10] R. O. Duda, P. E. Hart, et al. *Pattern classification and scene analysis*, volume 3. Wiley New York, 1973.

[11] C. Elkan. Using the triangle inequality to accelerate k-means. In *ICML*, volume 3, pages 147–153, 2003.

[12] M. P. Forum. Mpi: A message-passing interface standard. Technical report, Knoxville, TN, USA, 1994.

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

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

[15] 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.

[16] h2o. h2o. http://h2o.ai/, 2005–2015.

[17] K. Hornik, I. Feinerer, M. Kober, and C. Buchta. Spherical k-means clustering. *Journal of Statistical Software*, 50(10):1–22, 2012.

[18] A. Inc. Amazon web services.

[19] G. Inc. Google cloud platform.

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

[21] 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)*, 5(3):308–323, 1979.

[22] S. P. Lloyd. Least squares quantization in pcm. *Information Theory, IEEE Transactions on*, 28(2):129–137, 1982.

[23] 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.

[24] 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.

[25] 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.

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

[27] 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.

[28] 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.

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

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

[31] R. Pearce, M. Gokhale, and N. M. Amato. Multithreaded asynchronous graph traversal for in-memory and semi-external memory. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, 2010.

[32] 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*, 12:2825–2830, 2011.

[33] R Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria, 2015.

[34] L. Rokach and O. Maimon. Clustering methods. In *Data mining and knowledge discovery handbook*, pages 321–352. Springer, 2005.

[35] Scalability! but at what cost? http://www.frankmcsherry.org/graph/scalability/cost/2015/01/15/COST.html, Accessed 9/3/2016.

[36] D. Sculley. Web-scale k-means clustering. In *ACM Digital library*, pages 1177–1178, 2010.

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

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

[39] 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*, 344(6182):386–392, 2014.

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

[41] 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*, pages 2–2. USENIX Association, 2012.

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

[43] W. Zhao, H. Ma, and Q. He. Parallel k-means clustering based on mapreduce. In *Cloud Computing*, pages 674–679. Springer, 2009.

[44] 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.

[45] 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.

[46] J. Zhu, X. Fang, Z. Guo, M. H. Niu, F. Cao, S. Yue, and Q. Y. Liu. Ibm cloud computing powering a smarter planet. In *IEEE International Conference on Cloud Computing*, pages 621–625. Springer, 2009.