clusterNOR: A NUMA-Optimized Clustering Framework

Disa Mhembere[1], Da Zheng[2], Carey E. Priebe[3], Joshua T. Vogelstein[4], and Randal Burns[1]

[1]Department of Computer Science, Johns Hopkins University
[2]Amazon Inc.
[3]Department of Applied Mathematics and Statistics, Johns Hopkins University
[4]Institute for Computational Medicine, Department of Biomedical Engineering, Johns Hopkins University

Abstract—Clustering algorithms are iterative and have complex data access patterns that result in many small random memory accesses. The performance of parallel implementations suffer from synchronous barriers for each iteration and skewed workloads. We rethink the parallelization of clustering for modern non-uniform memory architectures (NUMA) to maximizes independent, asynchronous computation. We eliminate many barriers, reduce remote memory accesses, and maximize cache reuse. We implement 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. For algorithms that rely on Euclidean distance, clusterNOR defines an updated Elkan's triangle inequality pruning algorithm that uses asymptotically less memory so that it works on billion-point data sets. clusterNOR extends and expands the scope of the knor library for k-means clustering by generalizing underlying principles, providing a uniform programming interface and expanding the scope to hierarchical and linear algebraic classes of algorithms. The compound effect of our optimizations is an order of magnitude improvement in speed over other state-of-the-art solutions, such as Spark's MLlib and Apple's Turi.

Index Terms—NUMA, clustering, parallel, k-means, SSD, cloud-computing

1 Introduction

Clustering data to maximize within-cluster similarity and cross-cluster variance is highly desirable for the analysis of structured big data. Many iterative clustering algorithms roughly follow the Majorize-Minimization or Minorize-Maximization (MM) [20] pattern of computation. In this setting the raw data are not modified but are processed continuously in each iteration of the algorithm with only algorithmic metadata being modified. State-of-the-art machine learning frameworks that tackle clustering [23], [29], [32], [35] fail to take advantage of:

- The predictable pattern of computation followed by MM algorithms for caching purposes
- Modern multi-core NUMA machines that represent the vast majority of commodity servers in use today both commercially and academically
- The abundant availability of low-cost, high speed storage devices like Solid State Drives (SSDs)

These frameworks 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 [5] are built. Furthermore, partitioning multi-billion data points is essential to targeted ad-driven organizations such as Google [9] and Facebook [42]. In addition, clustering is highly applicable to neuroscience and genetics research. Connectomics [7], [24], [25], uses clustering to group anatomical regions by structural, physiological, and functional similarity, for the purposes of inference. Behavioromics [43] uses clustering to map neurons to distinct motor patterns. In genetics, clustering is used to infer relationships between genetically similar species [18], [33].

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 each iteration updates global state, the cluster membership. 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], [29], [32] 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 the MM computation pattern. 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) [13] for algorithms that use k-means (Section 5.1) in part or wholly. 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, clusterNOR prefers scale-up computation on shared-memory multicore machines in order to eliminate network traffic and perform fine-grained synchronization. clusterNOR generalizes and expands the core capabilities of the knor [30] library for k-means clustering. 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 [3]. Recent findings [28], [47], 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. Applications (Section 5) are benchmarked on a single or few machines to minimize network bottlenecks. We find that even single node performance (with SSDs) outperforms competitor distributed performance, in many instances.

Our framework improves performance for iterative machine learning algorithms that use the MM pattern for objective function optimization. We utilize NUMA-aware allocation and task scheduling and caching policies to account for modern architectures. Additionally, we maximize parallelism by significantly merging both MM steps within algorithms that utilize k-means. We combine this optimization with the development of a practical modification to TI, that we call the minimal triangle inequality (MTI). TI incurs a memory increment of $\mathcal{O}(nd)$. As memory capacity limits scaling, this memory overhead renders TI impractical. In contrast, MTI requires an increase of only $\mathcal{O}(n)$ memory. In practice, MTI outperforms TI because it requires significantly less data structure maintenance, while still pruning computation comparably. The resulting clusterNOR framework is capable of clustering data an order of magnitude faster than competitors.

We also demonstrate that the computation principles apply in a distributed setting (Section 11.9) and are transferable to semi-external memory (SEM) (Section 11.5). We define SEM as holding $\mathcal{O}(n)$ data in memory while streaming $\mathcal{O}(nd)$ data from disk for a dataset, $\vec{V} \in \mathbb{R}^{n \times d}$. This notion of SEM is analogous to that of graph algorithms in the literature [1], [34], in which vertex state is kept in memory and edge list on disk. We develop a modified FlashGraph engine [47] to support SEM computation and perform overlapped asynchronous I/O and computation on a single machine.

This work demonstrates that clustering extremely large datasets can be run on increasingly smaller/fewer machines. This reduces monetary expense and power consumption. Furthermore, our routines are highly portable. We simply require the C++11 standard library and thread-level parallelism is implemented using the POSIX thread (p-threads) library. Distributed routines rely on the Message Passing Library, MPI [14]. The I/O components for SEM routines are implemented using low-level Linux interfaces.

2 Related Work

Mahout [32] is a machine learning library that combines canopy (pre-)clustering [27] alongside MM-style algorithms to cluster large-scale datasets. Mahout relies on the Hadoop! an open source implementation of MapReduce [10] 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 reduce phase that arises from skew in the assignment of data points to clusters.

MLlib is a machine learning library for Spark [45]. 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 the resilient distributed datasets (RDD) [44]. 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 [35], ClusterR [31], and mlpack [8], 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 [40], [41] focus on developing serialized clustering approximations. Sophia-ML uses a mini-batch algorithm 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 [40]. 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 [41] 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 maximum parallelism.

Euclidean distance (Section 3) 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 [13], 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 [12] 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. We present a *minimal* triangle inequality (MTI) for computation pruning, that is nearly as effective and uses only $\mathcal{O}(n)$ memory, which makes it practical for use with big-data.

The semi-external memory (SEM) optimizations we implement were inspired by FlashGraph [47] and implemented using the same techniques for asynchronous I/O and overlapped computation. FlashGraph is an SEM graph computation framework that places edge data on SSDs and allows user-defined vertex state to be held in memory. 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 runs on top of a userspace filesystem called SAFS [46] that merges independent I/O requests into larger transfers and manages a *page cache* that keeps frequently touched pages in memory. Section 9 discusses how we modify the FlashGraph to build SEM computation into clusterNOR.

3 Nomenclature

Throughout the manuscript, we will use the following terms. 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.

4 Application Programming Interface (API)

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

- 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`.

4.1 `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 5.1, 5.4, 5.5, and 5.6).

- `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.

4.2 `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 5.7, 5.8, and 5.9). For performance reasons, this interface is iterative rather than recursive. We discuss this design decision and its numerable merits in Section 8. `hclust` provides the following additional abstract methods for user definition:

- `SplitStep()`: Used to determine when a cluster should split or merged with another cluster.
- `HclustUpdate()`: Used to update the hierarchical centroids from one iteration to the next.

4.3 `sem`

The SEM interface builds upon `base` and `hclust` and incorporates a modified FlashGraph [47] 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[]`.

4.4 `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.

4.5 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
    }
  }
}
```

5 Applications

We implement several algorithms to demonstrate the utility, extensibility and performance of clusterNOR.

5.1 k-means

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 6. For SEM, the computation complexity remains unchanged, but the asymptotic memory consumption drops to $\mathcal{O}(n + Tkd)$. We build k-means with `base` and minimize the following objective function for each data point, $\vec{v}$ [22]:

$$min \sum_{\vec{v} \in \vec{V}} ||\mathbf{d}(\vec{v}, \vec{c}^t)|| \tag{1}$$

5.2 Spherical k-means (sk-means)

Spherical k-means (sk-means) [11] 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. We build spherical k-means with `base`.

5.3 k-means++

We develop a standalone k-means++ [4] 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 squared euclidean distance between a centroid and constituent cluster members. 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:

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

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

5.4 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) [40] 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:

$$\vec{C}^t \leftarrow (1 - \eta)C^{t-1} + \eta\vec{V} \tag{3}$$

5.5 Fuzzy C-means (fc-means)

Fuzzy C-means (fcm) [6] 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 $J \in \mathbb{R}^{n \times k}$:

$$J = \sum_{i=1}^{|N|} \sum_{k=1}^{|C|} u_{ik}^z ||\vec{v}_i - \vec{c}_j||^2, 1 \leq z < \inf \tag{4}$$

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

5.6 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 complexity of $\mathcal{O}(n-k)^2$. We reduce the computation cost by implementing a sampled variant called (CLARA) [19] that is more practical, but still has a high asymptotic complexity of $\mathcal{O}(k^3+nk)$.

5.7 Hierarchical k-means (H-means)

We implement a divisive version of k-means using the `hclust` 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.

5.8 X-means

X-means [36] 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) [39]. Computationally, it differs from H-means (Section 5.7) 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 our `hclust` interface.

5.9 Gaussian Means (G-means)

G-means is built on `hclust` 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 [2] as the test to decide splits. The Anderson-Darling statistic performs roughly four times more computations than BIC, despite having the same asymptotic complexity.

6 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 use k-means have this property. Our implementation modifies the most popular synchronous algorithm for k-means, Lloyd's algorithm [22]. 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:

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 $\mathcal{O}(T)$ over Lloyd's. This algorithm design naturally leads to lock-free routines that require fewer synchronization barriers as we show in Algorithm 1.

Algorithm 1 || Lloyd's algorithm

```
1:  procedure ||MEANS(V⃗, C⃗^t, k)
2:    pt⃗C^t                                  ▷ Per-thread centroids
3:    clusterAssignment^t                    ▷ Shared, no conflict
4:    tid                                    ▷ Current thread ID
5:    parfor v⃗ ∈ V⃗ do
6:      dist = ∞
7:      c_nearest^t = INVALID
8:      for c⃗^t ∈ C⃗^t do
9:        if d(v⃗, c⃗^t) < dist then
10:         dist = d(v⃗, c⃗^t)
11:         c_nearest^t = c⃗^t
12:       end if
13:     end for
14:     pt⃗C^t[tid][c_nearest^t] += v⃗
15:   end parfor
16:   clusterMeans = MERGEPTSTRUCTS(pt⃗C^t)
17: end procedure

18: procedure MERGEPTSTRUCTS(vectors)
19:   while |vectors| > 1 do
20:     PAR_MERGE(vectors)                   ▷ O(Tlogn)
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) [13] 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 11.6.1 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 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 [13] pruning clauses in an iteration of k-means using MTI. Let $u^t = \mathbf{d}(\vec{v}, \vec{c\_nearest}^t) + f(\vec{c\_nearest}^t)$, be the upper bound of the distance of a sample, $\vec{v}$, in iteration t from its assigned cluster $\vec{c\_nearest}^t$. 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 $u^t \leq \min \mathbf{d}(\vec{c\_nearest}^t, \vec{c}^t \, \forall \, \vec{c}^t \in \vec{C}^t)$, then $\vec{v}$ remains in the same cluster for the current iteration. For semi-external memory, this is extremely significant because no I/O request is made for data.

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

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

7 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: 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 NUMAlink interconnect. The effect 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 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.

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]

Fig. 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.

Customized scheduling and work stealing: clusterNOR customizes scheduling for algorithm-specific computation patterns. For example, Fuzzy C-means 5.5 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 2) 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 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 an external-memory parallel reduction operator, much like funnel-sort [16], 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.

Effective data layout for CPU cache exploitation and

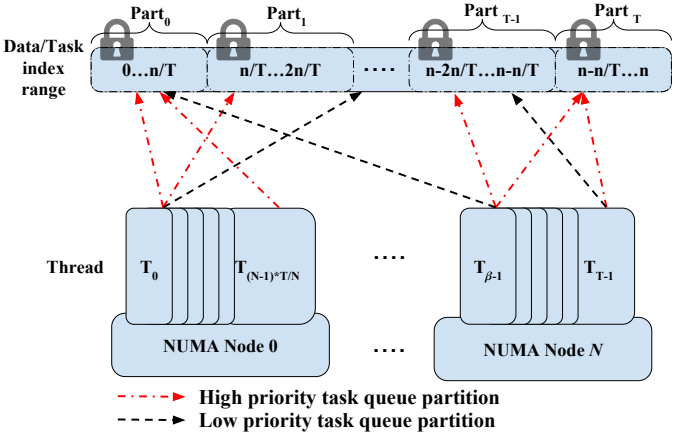


Fig. 2: The NUMA-aware partitioned task scheduler. The scheduler minimizes task queue lock contention and remote memory accesses by prioritizing tasks with data in the local NUMA memory bank.

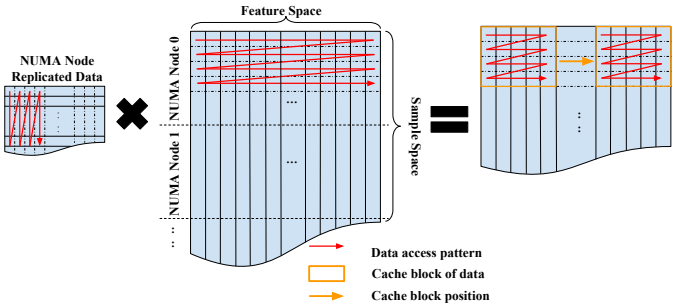


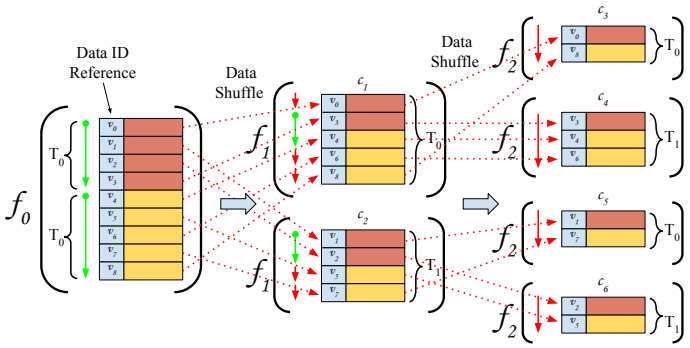
Fig. 3: Data access patterns support NUMA locality, utilize prefetched data well and optimize cache reuse through a cache blocking scheme.

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 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.

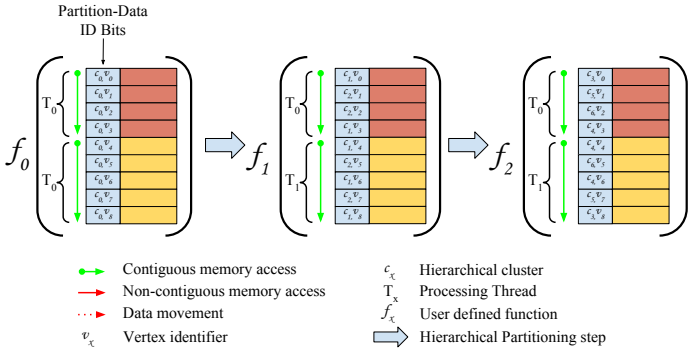
8 Hierarchical design

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 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 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.

Fig. 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.

9 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, $\vec{V} \in \mathbb{R}^{n \times d}$ that when processed completely in memory would require $\mathcal{O}(nd)$ memory.

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.

9.1 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 6) 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.

9.1.1 Partitioned Row Cache (RC)

We add a layer to the memory hierarchy for SEM applications by designing an optionally lazily-updated row cache (Figure 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 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.

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:

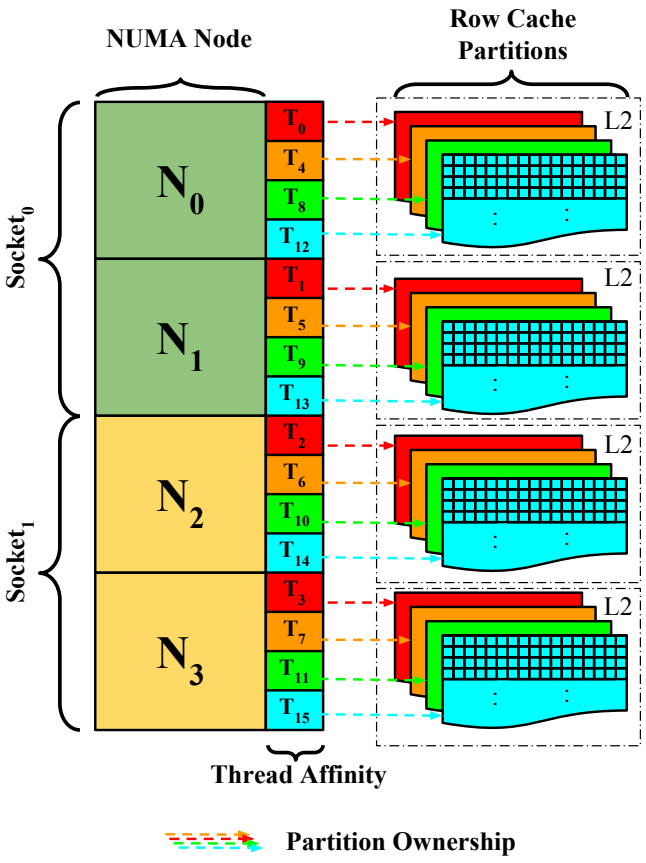


Fig. 5: The structure of the row cache for SEM applications in a two socket, four NUMA node machine utilizing 16 threads. Partitioning the row cache eliminates the need for locking during cache population and false sharing during data access. The aggregate size of all row cache partitions resides within 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, 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 9.

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. This differs from FlashGraph's page cache in that we cache rows at a time and not full pages. This mode has higher maintenance overhead, but is

more general for cases in which data access patterns are less predictable.

10 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 routines. Each machine maintains a decentralized *driver* (MPI) process that launches *worker* (pthread) threads that retain the NUMA performance optimizations across its multiple processors.

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 1) outweigh the effects of skew in this setting. We validate these assertions empirically in Section 11.9.

11 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 12.

We evaluate knor optimizations and benchmark against other state-of-the-art frameworks. In Section 11.3 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 11.4 and 11.5 evaluate the effect of specific optimizations on our in-memory and semi-external memory tools respectively. Section 11.7 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 11.9 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 11.7). The datasets we use for performance and scalability evaluation are shown in Table 2. Additionally, a summary of knor routine memory bounds is shown in Table 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-EC2**: MLlib's k-means, on Amazon EC2 instances [17].
- **MPI**: a pure MPI [14] distributed implementation of ||Lloyd's (Section 6) with MTI pruning.
- **MPI-**: a pure MPI distributed implementation of ||Lloyd's with MTI pruning *disabled*.

TABLE 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 2: The datasets under evaluation in this study.

Data Matrix	n	d	Size
Friendster-8 [15] eigenvectors	66M	8	4GB
Friendster-32 [15] 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.

11.1 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 v3.13.0. The C++ code is compiled using mpicxx.mpich2 version 4.8.4 with the -O3 flag.

11.2 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.

11.3 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 [8], Scikit-learn [35] and R [38] 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 because 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

11.4 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 6). 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 6).

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 when thread-binding and static scheduling are employed.

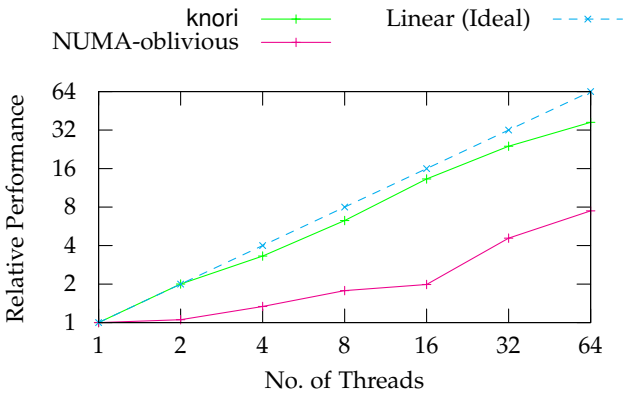


Fig. 6: Speedup of knori (which is NUMA-aware) vs. a NUMA-oblivious routine on the Friendster top-8 eigenvector dataset, with $k = 10$.

We demonstrate the effectiveness of a NUMA-aware partitioned task scheduler for pruned computations via knori (Figure 7). 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.

11.5 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 8a shows that as

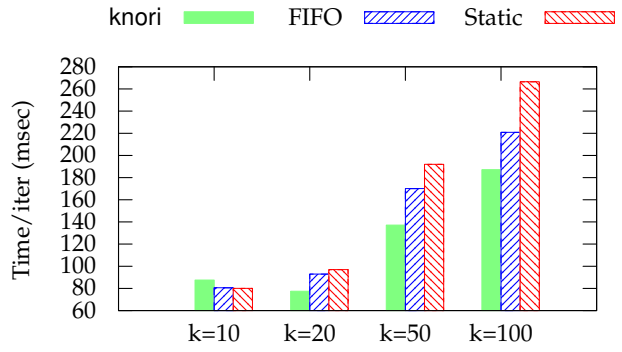
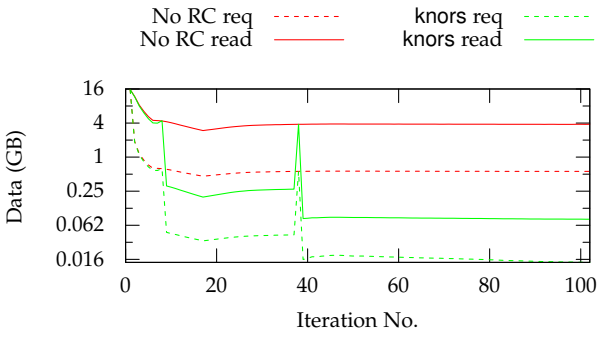
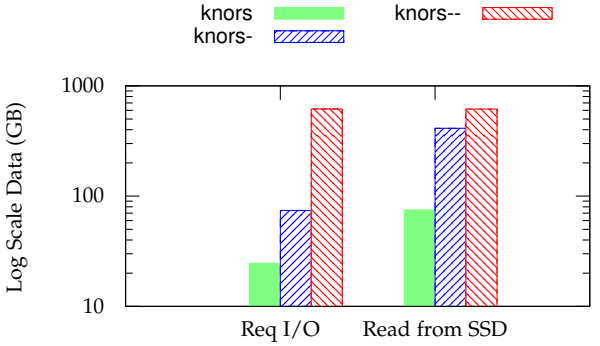


Fig. 7: Performance of the partitioned NUMA-aware scheduler (clusterNOR default) vs. FIFO and static scheduling for knori on the Friendster-8 dataset.



(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.

Fig. 8: 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, $k = 10$.

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 8b 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 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 8 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.

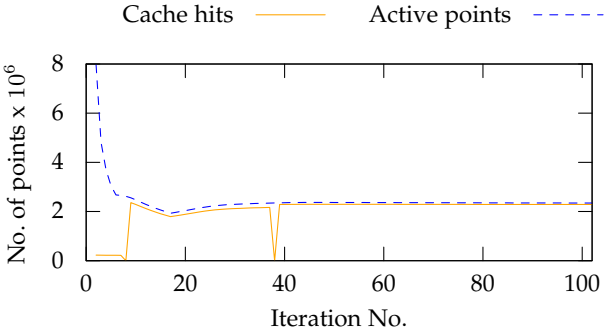


Fig. 9: Row cache hits per iteration contrasted with the maximum achievable number of hits on the Friendster top-32 eigenvectors dataset.

clusterNOR's lazy row update mode reduces I/O significantly for this application. Figure 9 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.

11.6 MTI Evaluation

We begin by evaluating the pruning efficacy, performance and memory consumption of MTI when compared with TI pruning in Section 11.6.1. We then show how MTI improves the performance of k-means compared to an implementation without pruning in Section 11.7.

11.6.1 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 10 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 10 demonstrates that MTI is comparable to TI in computation pruning capacity. MTI is within 15% of the pruning ability of TI. Furthermore, Figure 10 shows that as the number of clusters increase, MTI performance rapidly approaches that of TI. Finally, Figure 10 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.

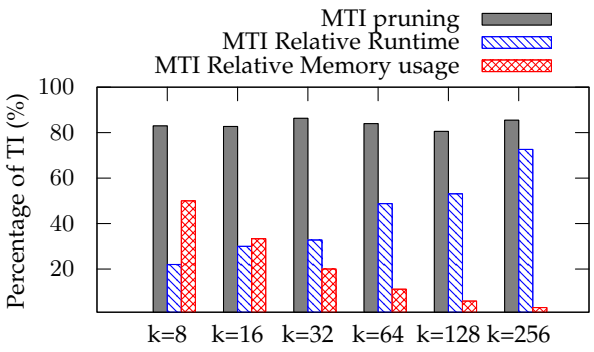


Fig. 10: Comparison of the pruning efficacy, memory consumption and runtime performance of MTI vs. TI on the Friendster-32 dataset using k-means.

11.6.2 MTI Performance Characteristics

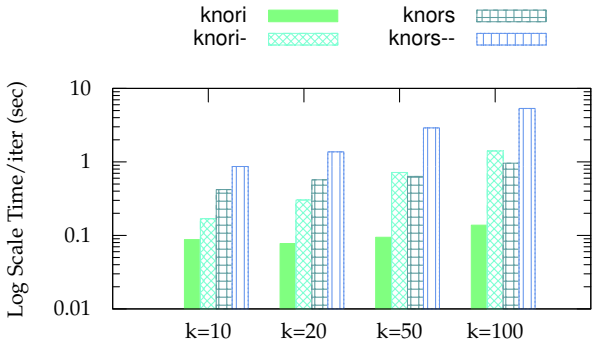
Figures 11a and 11b highlight the performance improvement of knor modules with MTI *enabled* over MTI *disabled* counterparts. We show that MTI provides a few factors of improvement in time when enbabled. Figure 11c highlights that MTI increases the memory load by negligible amounts compared to non-pruning modules. We conclude that MTI (unlike TI) is a viable optimization for large-scale datasets.

11.7 knor vs. Other Frameworks

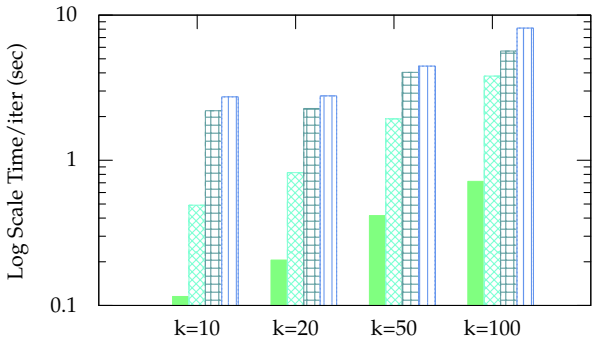
We evaluate the performance of knor in comparison with other frameworks on the datasets in Table 2. We show that knori achieves greater than an order of magnitude improvement over other state-of-the-art frameworks. Finally, we demonstrate knors outperforms other state-of-the-art frameworks by several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

Both our in-memory and semi-external memory modules incur little memory overhead when compared with other frameworks. Figure 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.

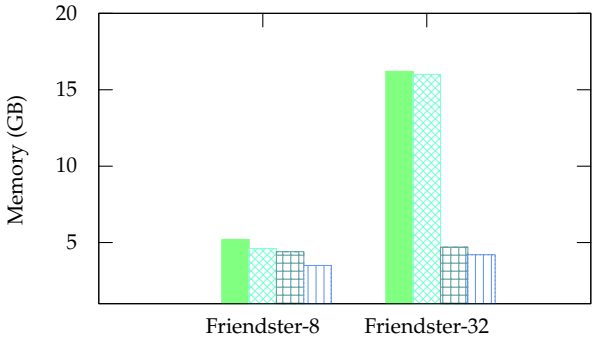
We demonstrate that knori is no less than an order of magnitude faster than all competitor frameworks (Figure 12). knori is often hundreds of times faster than Turi. Furthermore, knors is consistently twice as fast as competitor in-memory frameworks. We further demonstrate performance improvements over 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



(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--).

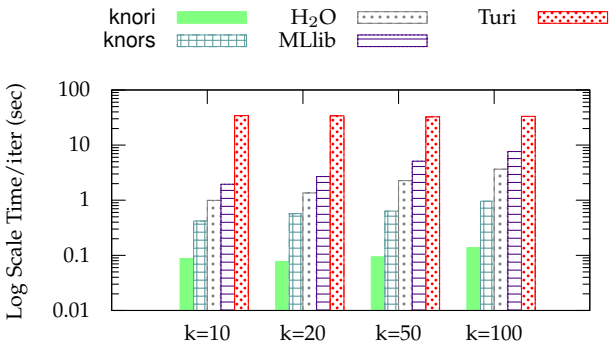
Fig. 11: Performance and memory usage comparison of knor modules on matrices from the Friendster graph top-8 and top-32 eigenvectors.

scheme for Lloyd's (Algorithm 1). Lastly, Figure 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.

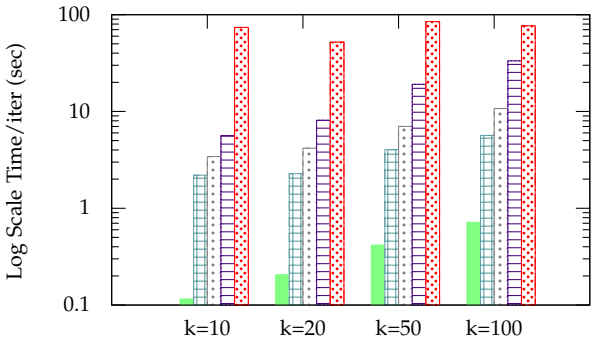
Finally, comparing knori- and knors-- to MLlib, H_2O and Turi (Figures 11 and 12) reveals knor to be several times faster and to use significantly less memory. This is relevant because knori- and knors-- are algorithmically identical to k-means within MLlib, Turi and H_2O.

11.8 Single-node Scalability Evaluation

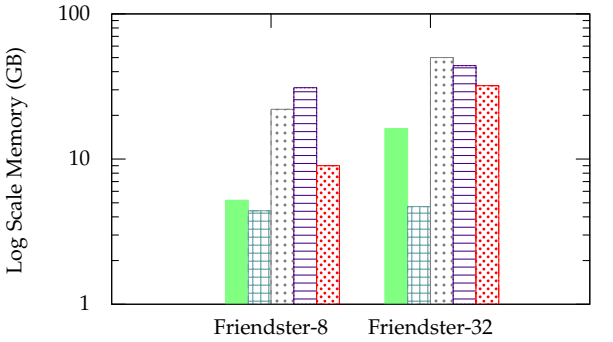
To demonstrate scalability, we compare k-means performance on synthetic datasets drawn from random distributions that contain hundreds of millions to billions of data points. Uniformly random data are typically the worst case scenario for the convergence of k-means, because many data points tend to be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



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



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

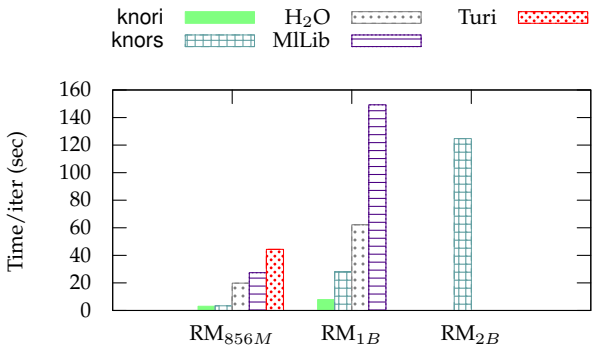


(c) Peak memory consumption on the Friendster datasets, with $k = 10$. Row cache size = 512MB, page cache size = 1GB.

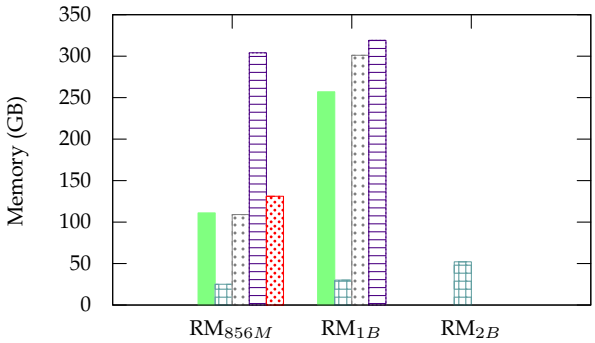
Fig. 12: knor routines outperform competitor solutions in runtime performance and memory consumption.

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 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.



(a) Per iteration runtime of each routine.



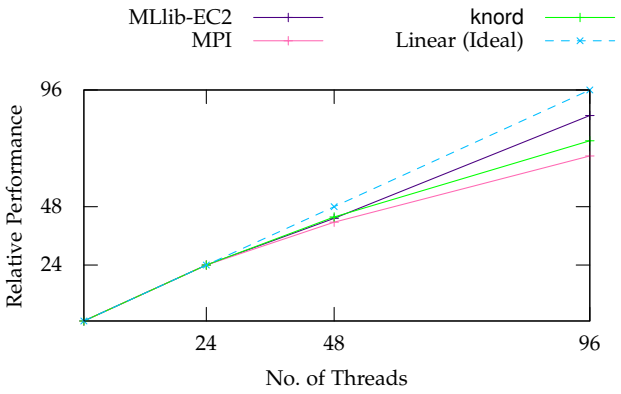
(b) Memory consumption of each routine.

Fig. 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.

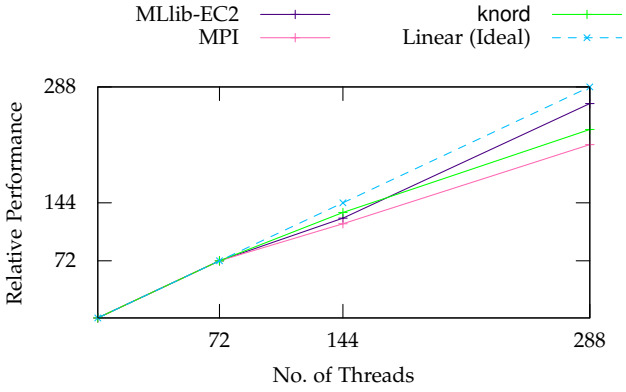
11.9 Distributed Comparison vs. Other Frameworks

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

Figures 14 and 15 reveal several fundamental and important results. Figure 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 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 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 15 shows that MTI remains a low-overhead, effective method to reduce computation even in the distributed setting.

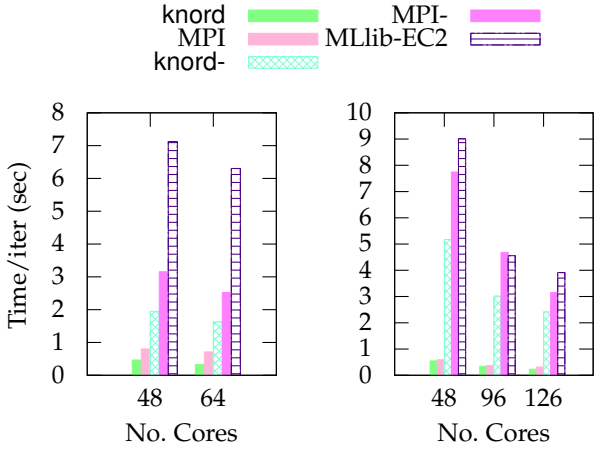


(a) Distributed speedup comparison on the Friendster-32 dataset.

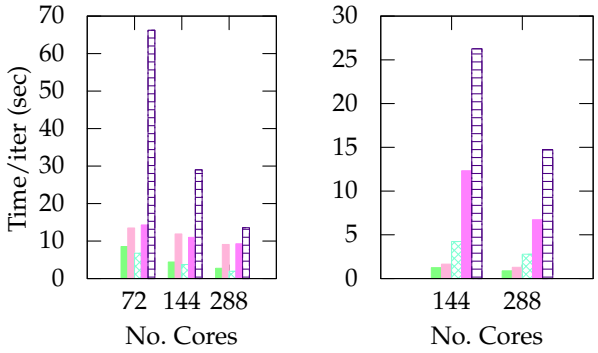


(b) Distributed speedup comparison on the RM_{1B} dataset.

Fig. 14: Speedup experiments are normalized to each implementation's serial performance. Each machine has 18 physical cores with 1 thread per core.



(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 $k = 10$.

Fig. 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.

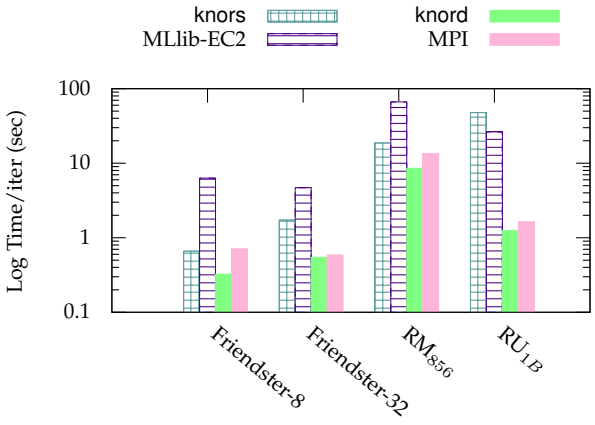


Fig. 16: Performance comparison of knors to distributed packages. knors uses one i3.16xlarge machine with 32 physical cores. knord, MLlib-EC2 and MPI use 3 c4.8xlarge with a total of 48 physical cores for all datasets other than RU_{1B} where they use 8 c4.8xlarge with a total of 128 physical cores.

11.9.1 *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 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.

12 Application Evaluation

We benchmark the performance of the nine applications developed using clusterNOR (Section 5). We present results for in-memory execution only for space reasons. The relative performance in other settings, SEM and distributed memory, track in-memory results closely. Figure 17 demonstrates that for applications with similar computational complexity as k-means, clusterNOR achieves comparable performance to knor, which we consider state of the art. This indicates 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 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 [40] 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 3). 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*, a $\mathcal{O}(kn)$ data structure containing the probability of a data point being in a cluster.

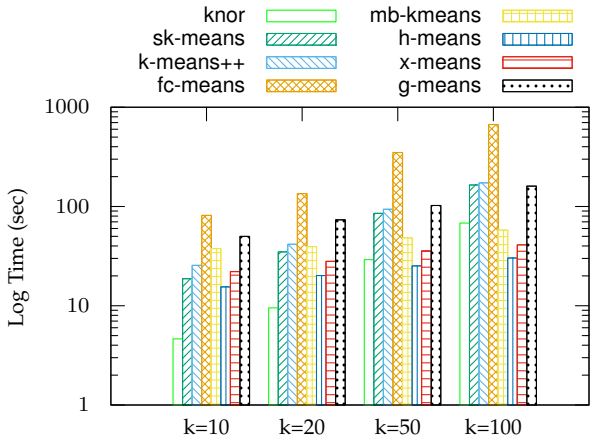


Fig. 17: In-memory performance of clusterNOR benchmark applications on the Friendster-32 dataset. We fix the number of iterations to 20 for all applications and use a mini-batch size of 20% of the data size for mb-kmeans.

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 `SplitSteps`. This increases the cost of hierarchical clustering over H-means (Figure 18), in which X-means and G-means perform at about 70% and 30% of the performance of H-means.

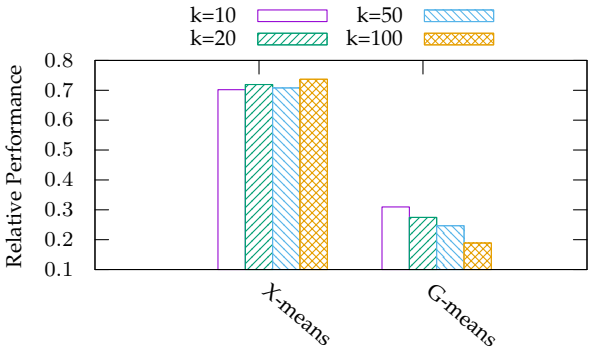


Fig. 18: The relative performance of hierarchical algorithms in comparison to H-means, the baseline hierarchical cluster application on the Friendster-32 dataset

We present the result of the k-medoids experiment (Table 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.

TABLE 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	$2154.81s$	$2616.57s$	$1801.56s$	$761.98s$

13 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 [37] 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.

ACKNOWLEDGMENTS

This work is partially supported by DARPA GRAPHS N66001-14-1-4028 and DARPA SIMPLEX program through SPAWAR contract N66001-15-C-4041. We thank Nikita Ivkin for discussions that assisted immensely in realizing this work.

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

[3] 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.

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

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

[6] J. C. Bezdek, R. Ehrlich, and W. Full. Fcm: The fuzzy c-means clustering algorithm. *Computers & Geosciences*, 10(2-3):191–203, 1984.

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

[8] 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.

[9] 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.

[10] 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.

[11] I. S. Dhillon and D. S. Modha. Concept decompositions for large sparse text data using clustering. *Machine learning*, 42(1-2):143–175, 2001.

[12] 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.

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

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

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

[16] 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.

[17] A. Inc. Amazon web services.

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

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

[20] K. Lange. *MM optimization algorithms*, volume 147. SIAM, 2016.

[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] 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*, pages 169–178. ACM, 2000.

[28] 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.

[29] 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.

[30] D. Mhembere, D. Zheng, C. E. Priebe, J. T. Vogelstein, and R. Burns. knor: A numa-optimized in-memory, distributed and semi-external-memory k-means library. In *Proceedings of the 26th International Symposium on High-Performance Parallel and Distributed Computing*, pages 67–78. ACM, 2017.

[31] L. Mouselimis. *ClusterR: Gaussian Mixture Models, K-Means, Mini-Batch-Kmeans, K-Medoids and Affinity Propagation Clustering*, 2018. R package version 1.1.7.

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

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

[34] 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.

[35] 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.

[36] D. Pelleg, A. W. Moore, et al. X-means: Extending k-means with efficient estimation of the number of clusters. In *Icml*, volume 1, pages 727–734, 2000.

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

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

[39] G. Schwarz et al. Estimating the dimension of a model. *The annals of statistics*, 6(2):461–464, 1978.

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

[41] 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.

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

[43] 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.

[44] 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.

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

[46] 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.

[47] 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.