NUMA-optimized In-memory and Semi-external-memory Parameterized Clustering

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

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

***Abstract*—K-means is one of the most influential and utilized machine learning algorithms today. Its computation limits the performance and scalability of many statistical analysis and machine learning tasks. We rethink k-means in terms of modern architectures to develop a novel parallelization scheme that delays and minimizes synchronization barriers. We utilize this to develop two modules k||means$_T$, and SEM-k||means$_T^{RC}$. k||means$_T$ is optimized for NUMA architectures and radically improves the performance for datasets that fit in memory. SEM-k||means$_T^{RC}$ improves k-means' scalability on a memory budget using semi-external memory, SSDs and a minimalistic triangle inequality pruning algorithm. SEM-k||means$_T^{RC}$ scales to billions of points on a single machine, using a fraction of the resources that distributed in-memory systems require. We show k||means$_T$ outperforms commercial products like H_2O, Dato and Spark's MLlib by up to an order of magnitude on $O(10^7)$ point datasets. Furthermore, SEM-k||means$_T^{RC}$ effortlessly clusters $\geq O(10^8)$ point datasets while outperforming distributed in-memory frameworks.**

I. 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 $\mathbb{R}^{nd}, \{n, d\} \in \mathbb{N}^+$ data into $k \in \mathbb{N}^+$ clusters. The most popular synchronous variant of k-means is Lloyd's [14] algorithm. Similar to Expectation Maximization [8], Lloyd's algorithm proceeds in two phases. In phase one we compute 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 III). Despite k-means' popularity, state-of-the-art machine learning libraries [18], [15], [12] 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. We argue that larger datasets on smaller/fewer machines will create savings in monetary expense and power consumption.

The decomposition of extremely large datasets into clusters of data points that are similar is a topic of great interest in industry and academia alike. For example, clustering is the backbone upon which popular user recommendation systems at Netflix [2] are built. Furthermore, partitioning multi-billion data points is essential to targeted ad-driven organizations such as Google [6] and Facebook [24]. In addition, clustering is extremely significant within neuroscience and genetics research. Connectomics [3], [16], [17], uses clustering to group anatomical regions by structural, physiological, and functional similarity, for the purposes of inference. In genetics, clustering is used to infer relationships between genetically similar species [13], [20].

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, k-means is challenging to optimize because its performance varies as a non-linear function of n: the number of data points, d: the data dimensionality, and k: the number of clusters. Altering either of n, d or k can fundamentally change the profile of computation and I/O (in the semi-external memory case). Fully asynchronous computation for Lloyd's algorithm is infeasible because data points require updated global state in each iteration. This poses a major challenge to performance and scalability especially within parallel and distributed implementations due to global barriers. Performance degradation caused by barriers is exacerbated in large scale datasets as a greater number of iterations are necessary to converge. As such, we must minimize critical sections of code.

Popular frameworks [15], [19], [18] have converged on a computation model involving scale-out, distributed processing in which data are partitioned among cluster members and global updates are transmitted at the speed of the interconnect. These frameworks are negatively affected by network traffic, straggler nodes, cluster liveness protocols, and data management protocols. These factors consume CPU cycles that would otherwise be utilized for computation. Furthermore, such frameworks would struggle to capitalize on potential gains from the use of computation pruning techniques (like TRI [9]) without introducing either worker skew, heavy network traffic or memory bloat. These drawbacks are a direct result of (M)TRI pruning data points in a near-random manner, often leaving a few workers with the bulk of the computation while many workers are idle, i.e. skew. Heavier network traffic

would result if these frameworks mitigate skew by data sharing via message passing. Data bloat would result if workers were provided with more data than necessary so as to load balance.

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 k||means, a fast in-memory, NUMA-optimized implementation that performs an order of magnitude faster than other state-of-the-art systems for datasets that fit into main memory. We further implement a practical modification to the triangle inequality distance computation pruning algorithm (TRI) [9], that we call the minimal triangle inequality (MTRI). We demonstrate its practicality for in-memory applications via k||means$_T$. TRI uses a memory increment of $O(nd)$, limiting its scalability. MTRI requires an increase of only $O(n)$ memory, drastically improving its utility for large-scale datasets. We show that in practice a k-means routine using MTRI outperforms one using TRI because MTRI requires significantly less data structure maintenance and prunes computation comparably to TRI; k||means$_T$ achieves up to an order of magnitude speedup over k||means on real-world datasets. Finally, we present SEM-k||means and SEM-k||means$_T^{RC}$, the semi-external memory analogous implementations of k||means and k||means$_T$ respectively. Semi-external memory (SEM) in this setting is an algorithm that holds $O(n)$ data in memory while streaming $O(nd)$ data from disk for a dataset, $\mathbb{R}^{nd}$. Our semi-external memory interpretation is analogous to the definition of SEM algorithms defined for graphs [1], [21], in which algorithms maintain vertex state in memory and edge lists on disk. Our SEM implementations are built on a modified FlashGraph [4] framework to seamlessly scale computations by using SSDs. Both SEM-k||means and SEM-k||means$_T^{RC}$ use a fraction of the memory of popular frameworks and outperform them by large factors.

II. Related Work

Zhao et al [27] developed a parallel k-means routine within Hadoop!, an open source implementation of MapReduce [7]. Their implementation shares a great deal of similarity with Mahout [19], a machine learning library built on top of Hadoop!. The Map-Reduce paradigm consists of a `Map` phase, a synchronization barrier in which data are shuffled, then a `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 on one machine.

MLlib is a state-of-the-art machine learning library built on top of Spark [26]. 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. As such, Spark's management of its parallelized data structure, the resilient distributed datasets [25] and its runtime, create overhead that significantly degrade performance.

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 [22]. Sophia-ML's target application is online, real-time applications, which is orthogonal to our goal of exact k-means on large scale data. Shindler et al [23] developed a fast approximation that addresses scalability by streaming data from disk sequentially, thus limiting the amount of memory necessary to iterate. This shares some similarity with SEM-k||means$_T^{RC}$, but is geared toward a single processor environment, passing over the data just once and operating on medium-sized data. We avoid approximations; they see little widespread use because questions of cluster quality are difficult to overcome.

Elkan [9] proposed the use of the triangle inequality (TRI) with bounds, to reduce the number of distance computations to fewer than $O(kn)$ per iteration. TRI 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. This is because more data points are assigned to clusters with means that remain relatively constant, causing samples to not change cluster membership in the following iterations. Conversely, when a dataset has no natural clusters, e.g. uniform random data, TRI will prune fewer computations since most data points will change cluster membership frequently. Furthermore, the method hinges upon using a sparse lower bound matrix of size $O(nk)$, doubling the size of the in-memory state. In most applications this is not tolerable. Therefore we develop MTRI for k||means$_T$ and SEM-k||means$_T^{RC}$ that costs only $O(n)$ more memory.

III. Algorithms and Nomenclature

We first define notation that we use throughout this manuscript. Let k be the number of clusters into which we wish to partition a dataset. Let V be the set of all d-dimensional vectors, $\vec{v}_i$, $i \in \mathbb{N}$ be equivalent to the dataset $\mathbb{R}^{nd} \equiv V$.

Let C be the set of all dimension d cluster means, $\vec{c}_x$, $x \in \{1...k\}$. We can cluster any such $\vec{v}_i$ into a cluster $\vec{c}_x$. We use euclidean distance, denoted as $\mathbf{d}$, as the similarity metric between any $\vec{v}_i$ and $\vec{c}_k$, such that $\mathbf{d}(\vec{v}, \vec{c}) = \sqrt{(\vec{v}_1 - \vec{c}_1) + (\vec{v}_2 - \vec{c}_2) + ... + (\vec{v}_{d-1} - \vec{c}_{d-1}) + (\vec{v}_d - \vec{c}_d)}$. Finally, let T be the number of threads of concurrent execution, P be the number of processing units, and N be the number of NUMA nodes.

Lloyd's algorithm operates in two-phases; Phase I: Compute the nearest centroid, $c_{nearest}$ to each data point, $\vec{v}_i$, Phase II: Update each centroid, $\vec{c}_x$ to be the mean value of its members. Our ||Lloyd's creates a super-phase over the original two phases by maintaining lock-free and per-thread data structures to represent centroids, assignment, and distance vectors. This results in less than 1% of computation time spent within

a serial critical region. Algorithm 1 outlines a vanilla (no optimizations included) implementation.

Algorithm 1 || Lloyd's algorithm

```
1:  procedure ||MEANS(V, C, K)
2:      ptCentroids                          ▷ Per-thread centroids
3:      clusterAssignment                    ▷ Shared, no conflict
4:      parfor v_i ∈ V do
5:          for c_i ∈ C do
6:              [dist_min, cid_min] = min(d(v_i, c_i))
7:          end for
8:          ptCentroids[CURR_THREAD][cid_min] += v_i
9:      end parfor
10:     clusterMeans = mergePtStructs(ptClusters)
11: end procedure

12: procedure MERGEPTSTRUCTS(vectors)
13:     while |vectors| > 1 do
14:         PAR_MERGE(vectors)               ▷ O(T log n)
15:     end while
16:     return vectors[0]
17: end procedure
```

A. Minimal Triangle Inequality (MTRI) Pruning

We simplify Elkan's Algorithm for triangle inequality pruning (TRI) [9] by removing the the necessity for the lower bound sparse 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. With $O(n)$ memory we implement three of the four [9] pruning clauses invoked for each data point in an iteration of a pruned routine, i.e. k||means$_T$ and SEM-k||means$_T^{RC}$. We define u_i to be the upper bound of the distance of a sample, v_i from its assigned cluster $c_{nearest}$. Finally, we define U to be an update function such that $U(u_i)$ fully tightens the upper bound of u_i.

Clause 1: if $u_i \leq \min \mathbf{d}(c_{nearest}, c_x \forall x \in k)$, then v_i remains in the same cluster for the current iteration. For SEM-k-||means$_T^{RC}$, no I/O request is made for data.

Clause 2: if $u_i \leq d(c_{nearest}, c_x)$, then the distance computation between data point v_i and centroid c_x is pruned.

Clause 3: if $U(u_i) \leq d(c_{nearest}, c_x)$, then the distance computation between data point v_i and centroid c_x is pruned.

IV. In-memory design

We prioritize practical use and performance when we implement k||means and k||means$_T$ optimizations. We make design tradeoffs to balance the opposing forces of minimizing memory usage and maximizing CPU cycles spent computing in parallel. Section IV chronicles the memory bounds we achieve and optimizations we apply.

A. In-memory Asymptotic Analysis

k||means achieves high performance with near-minimal memory usage. Its computation complexity remains at $O(ndk)$ with 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.

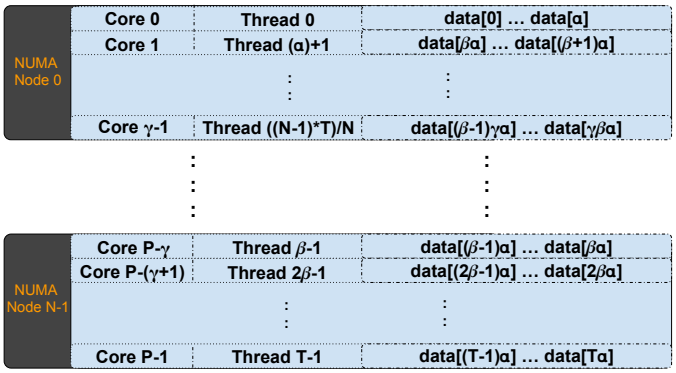


Fig. 1: The memory allocation and thread assignment scheme we utilize for k||means and k||means$_T$. $\alpha = |data|/T$ is the amount of data per thread, $\beta = T/N$ is the number of threads per NUMA node, and $\gamma = P/N$ is the number of physical processors per NUMA node.

k||means$_T$ raises the memory bound by $O(kd + n)$. The $O(kd)$ is derived from maintaining an upper/lower triangular centroid-to-centroid distance matrix, while the $O(n)$ maintains 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. The memory tradeoff is well spent because in practice the average computation time for k||means$_T$ is up to 5x faster than k||means and over 30x faster than other state-of-the-art implementations.

1) In-memory optimizations: We use the following design principles and optimizations to improve the performance of Algorithm 1 for in-memory tasks.

Prioritize data locality for NUMA architectures: 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 to local memory banks and a penalty for remote memory accesses to non-local memory. We thus 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; Figure 1 shows the data allocation and access scheme we employ that maximes local memory accesses.

Schedule effectively for NUMA architectures: Coupling a thread to a static partition of the data means a thread accesses nothing but its own partition's address space in each iteration. This inherently enforces a static task scheduling for threads that is optimal for the case of $k||means$.

When data points are pruned using k||means$_T$, we see worker thread skew. Neither the driver thread nor other worker threads are aware when any one thread is able to locally prune

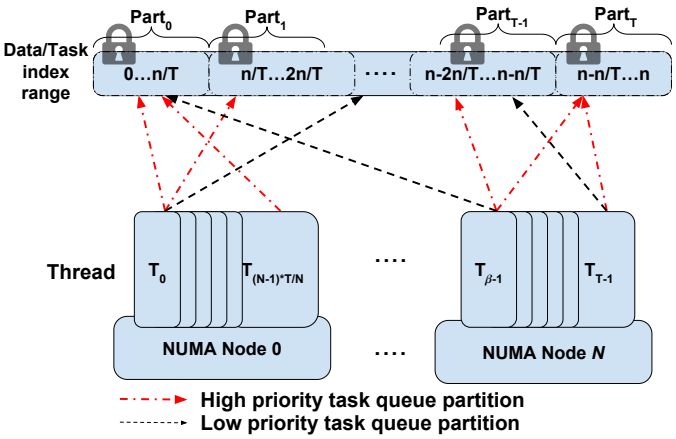


Fig. 2: The NUMA-aware partitioned task scheduler we utilize for k||means$_T$.

computation. As such, we employ dynamic task scheduling to combat skew. Dynamic task scheduling mitigates skew, but limits the ability reduce costly remote memory accesses on NUMA architectures. To limit remote memory accesses when pruning is activated, we dynamically schedule computation on each thread in a NUMA-cognizant fashion. We develop a NUMA-aware partitioned priority task queue, (Figure 2), to feed worker threads, prioritizing tasks that maximize local memory access. 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 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 and optimized memory access patterns. Figure 5 validates our scheduling claims showing our design decision's effectiveness.

Avoid interference and delay the critical section: We employ per-thread local centroids and write-conflict free shared data structures to reduce interference. Local centroids are un-finalized running totals of global centroids used in the following iteration for distance computations. Local centroids are concurrently updated. Finally, in a minimal critical section, local clusters are merged in a parallel funnelsort-like [11] merge 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 OS-level prefetching and CPU caching opportunities.

V. Semi-external Memory Design

We show that an optimized SEM implementation can outperform popular in-memory frameworks when computing k-means. SEM implementations extend their in-memory counterparts to datasets that are too large to fit into main memory by placing data on SSDs and requesting it as necessary. The SEM model allows us to reduce the asymptotic memory bounds so as to scale to larger instances. A SEM routine uses $O(n)$ memory for a dataset, $\mathbb{R}^{nd}$ that when processed completely in memory requires $O(nd)$ memory.

A. FlashGraph and SAFS

We build our SEM implementations on top of a modified FlashGraph [4] framework. FlashGraph 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 built on top of a userspace filesystem called SAFS [5]. 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. We choose the size of a page within the cache to be 4KB. This is equivalent to the minimum I/O request size that can be made to SSDs.

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-dimensional data point, $\vec{v}_i$, $i \in \mathbb{N}$. Each row is composed of a unique identifier, *row-ID*, and d-dimensional data vector, *row-data*. We replace FlashGraph's `page_vertex` type with a `page_row` data type 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 stores the row-ID in memory and row-data on disk. The `page_row` reduces the memory necessary to use FlashGraph by $O(n)$ because the `page_row` computes its row-data's location on SSDs autonomously unlike a `page_vertex` that stores this metadata. This allows SEM-k||means and SEM-k||means$_T^{RC}$ to scale to larger datasets than possible before on a single machine.

B. Semi-external Memory Asymptotic Analysis

We develop SEM implementations to handle data too large for main memory. Both SEM-k||means and SEM-k||means$_T$ perform better than distributed and in-memory systems despite operating on a single machine and using a fraction of the memory of other systems. SEM implementations do not alter the computation bounds, but do lower the memory bounds for k-means to $O(n + Tkd)$; this improves on the $O(nd +$

Tkd) bound of k||means and the $O(nd+kd)$ bound of Lloyd's original algorithm. In practice the disk I/O bound of SEM-k||means$_T^{RC}$ is much lower than the worst case of $O(nd)$ (e.g., SEM-k||means), especially for data with natural clusters. We show that even on datasets containing $O(10^7)$ data points, we require less than 50% of the worst case data brought into memory.

1) I/O minimization efforts: I/O bounds the performance of k-means in the SEM model. Accordingly, we aim to reduce the number of data-rows that need to be brought into memory each iteration. Only Clause 1 of MTRI (Section III-A) 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 retrive significantly more data than necessary from SSDs due to fragmentation. Reducing the filesystem *page size*, i.e. minimum read size from SSDs aleviates 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 V-B2, that drastically reduces the amount of data brought into memory. Figure 6a shows a reduction in I/O of over 30% when we activate the row cache.

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 the performance of SEM-k||means$_T^{RC}$ 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 udpate 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 dropped and 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 VI). The design trade-off is cache freshness for reduced cache maintenance overhead. We demonstrate the effectiveness of this design in Figure 7.

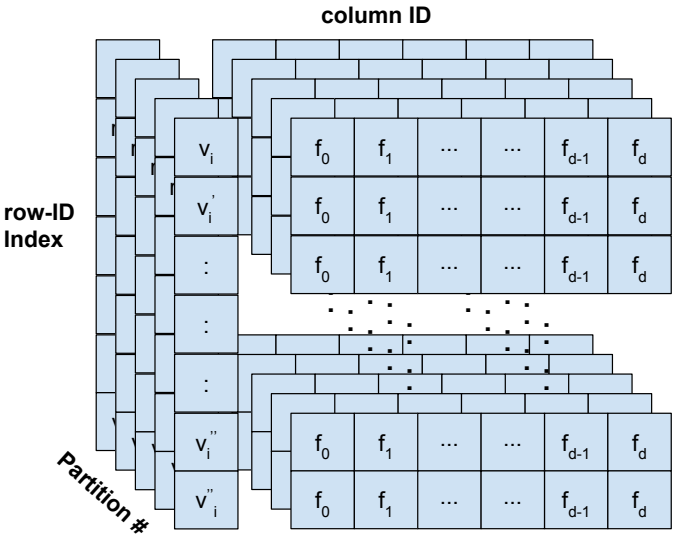


Fig. 3: The design of the row cache for SEM clustering.

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.

VI. Experimental Evaluation

Sections VI-A and VI-B evaluate the effect of specific optimizations on our in-memory and semi-external memory tools respectively. Section VI-C evaluates our performance relative to other popular frameworks from the perspective of time and resource consumption. We use the datasets shown in Table I to demonstrate the performance and scalability of our routines.

We run all experiments on single 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 g++ version 4.8.4 with the -O3 flag.

We use the following abbreviations in figures; SEM-k||means (SEM-k), SEM-k||means$_T$ (SEM-k$_T$) for SEM-k||means with the minimal triangle inequality (MTRI) and no row cache, and SEM-k||means$_T^{RC}$ (SEM-k$_T^{RC}$) for SEM-k||means with MTRI and the row cache enabled.

A. In-memory optimization evaluation

We show NUMA-node thread binding, maintaining NUMA memory locality, and static task scheduling for k||means is

Data Matrix	n	d	Size
Friendster-8 [10] eigenvectors	66M	8	4GB
Friendster-32 [10] 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

TABLE I: The datasets we use for evaluation.

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 hyperthreading. The NUMA-aware implementation is nearly 6x faster at 64 threads when compared to a NUMA-oblivious implementation. We further show that a NUMA-oblivious implementation has a lower linear constant for speedup when compared with a NUMA-aware implementation.

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.

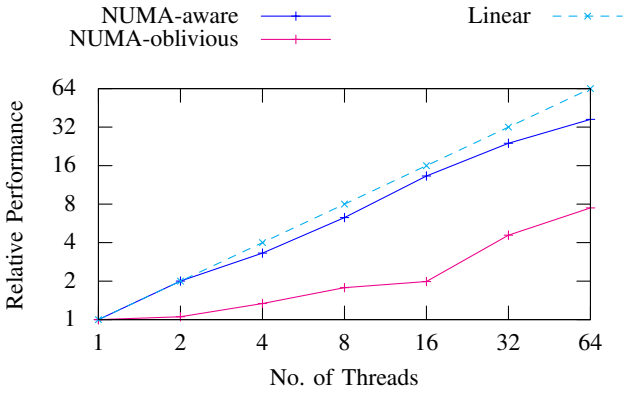


Fig. 4: Speedup of NUMA-aware vs. NUMA-oblivious routine for k||means on the Friendster top-8 eigenvector dataset, with $k = 10$.

We demonstrate the effectiveness of a NUMA-aware partitioned task scheduler for pruned computations via k||means$_T$ (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 have observed similar trends in other datasets. We have omitted these results for space reasons and to avoid redundancy.

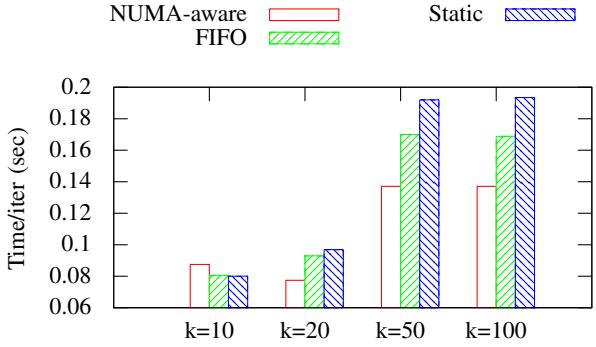
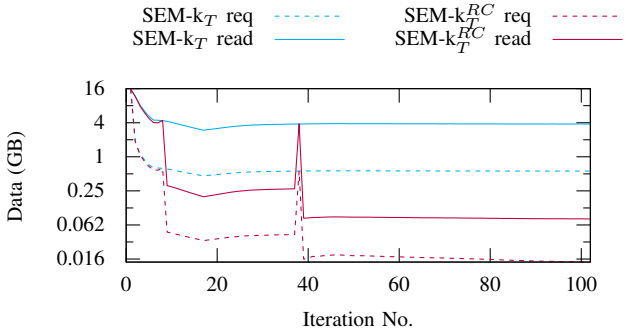


Fig. 5: Performance of the partitioned NUMA-aware scheduler vs. FIFO and static scheduling for k||means$_T$ on the Friendster top-8 eigenvector dataset.

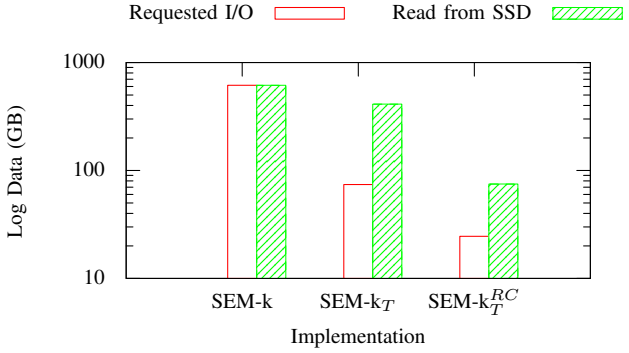
B. Semi-external memory evaluation

We evaluate SEM-k||means$_T$ 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 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, although to lesser extent on uniformly randomly generated data (Section VI-C).

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 an implementation requests from SSDs with the amount of data SAFS actually reads and transports into memory. SEM-k does not utilize pruning so every request made for row-data issued to FlashGraph is either served by FlashGraph's page cache or read from SSDs. SEM-k$_T$ prunes using MTRI, but does not use the row cache and as a result, reads an order of magnitude more data from SSDs than SEM-k$_T^{RC}$, which does use the row cache. We demonstrate that a page cache is not sufficient for the k-means and that caching at



(a) Data requested (req) from SSDs vs. data read (read) from SSDs each iteration. Because of pruning, algorithms may request only a few points from any block, but the entire block must be read from SSD.



(b) Total data requested vs. data read from SSDs. Without pruning, all data are requested and read.

Fig. 6: The effect of the row cache and MTRI on I/O for the Friendster top-32 eigenvectors dataset. Row cache size = 512MB, page cache size = 1GB, $k = 10$.

the granularity of row-data is necessary to achieve significant reductions in I/O and improvements in performance for real-world datasets.

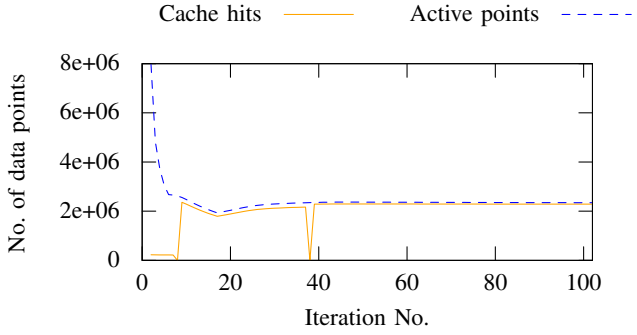


Fig. 7: Row cache hits per iteration contrasted with the maximum achievable number of hits on the Friendster top-32 eigenvectors dataset.

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 SEM-k||means$_T$ operates at in-memory speeds for the vast majority of iterations. In experiments that we omit due to space considerations, we observe that a fresh cache has minimal performance benefit in early iterations, owing to near-random row activation patterns. We further observe that the utility of the cache is limited when convergence is achieved within a very small number of iterations, but this is a rare case with large scale datasets.

C. Comparison against other frameworks

We evaluate the performance of our routines in comparison with other frameworks on all datasets and show that we achieve greater than an order of magnitude improvement in many situations. Both k||means and SEM-k||means utilize no pruning optimizations and are algorithmically identical to the implementations of MLlib, Dato and H_2O, showing that we outperform popular frameworks even without pruning.

Both our in-memory and semi-external memory routines incur little memory overhead when compared with other frameworks. Figure 8 shows memory consumption. We note that MLlib requires the placement of temporary Spark block manager files. Since the block manager cannot be disabled, we provide an in-memory RAM-disk so as to not influence MLlib's performance negatively.

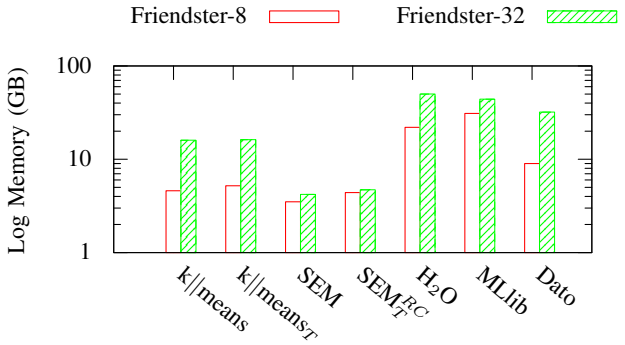
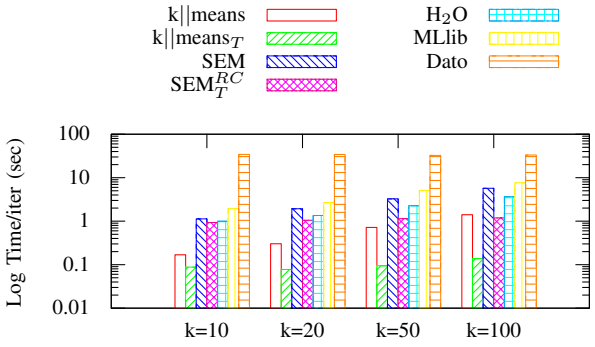


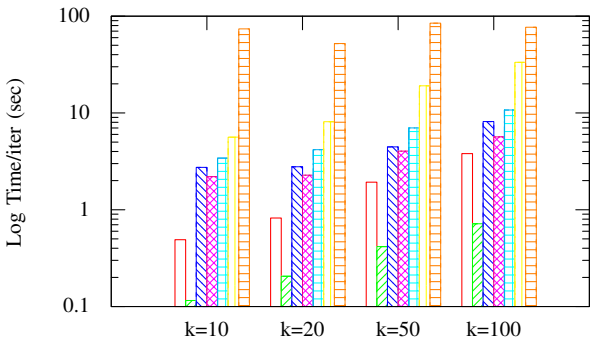
Fig. 8: Peak memory consumption of on the Friendster eigenvectors dataset, with $k = 10$. Row cache size = 512MB, page cache size = 1GB.

We demonstrate that k||means$_T$ is no less than an order of magnitude faster than all competitor frameworks (Figure 9). k||means$_T$ is often hundreds of times faster than Dato, furthermore SEM-k||means$_T^{RC}$ is consistently twice as fast as other in-memory frameworks. We further demonstrate performance improvements over competitor frameworks on algorithmically identical implementations via k||means and SEM-k||means. k||means is consistently over 7x faster than competitor solutions, whereas SEM-k||means is comparable and often faster than in-memory solutions. We attribute their performance to our parallelization scheme we develop for Lloyd's (Algorithm 1). Lastly, Figure 9 demonstrates a consistent 30% improvement of SEM-k||means$_T^{RC}$ over SEM-k||means$_T$, justifying the utility of the row cache.

We configure all frameworks using the minimum amount of memory necessary to achieve their highest performance. We



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



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

Fig. 9: Performance comparison on matrices from the Friendster [10] graph top-8 and top-32 eigenvectors.

acknowledge that a reduction in memory for these frameworks is possible, but would degrade computation time.

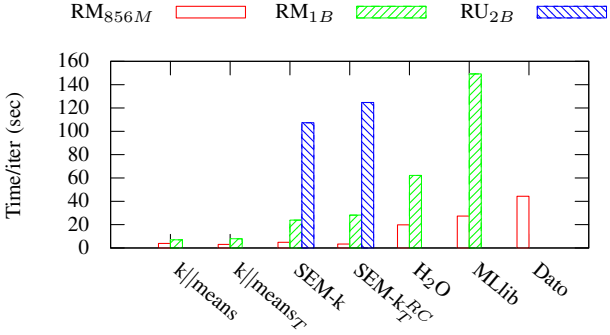
1) Random Dataset Evaluation: To demonstrate scalability, we compare performance on synthetic datasets drawn from random distributions that contain hundreds of millions to billions of data points. Uniformly random data are typically the worst case scenario for the convergence of k-means, because many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

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

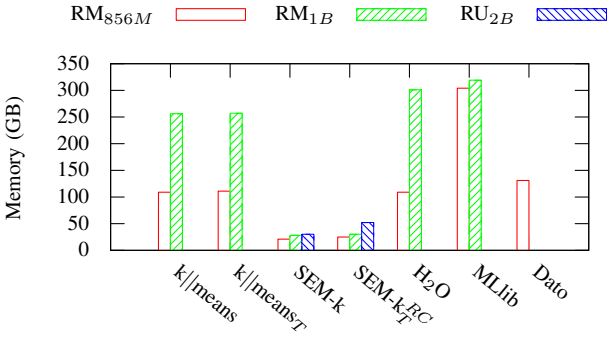
MTRI's effectiveness in reducing computation is minimal on the random multivariate datasets (Figure 10a). k||means$_T$ and SEM-k||means$_T^{RC}$ perform marginally better (10-20%) than their unpruned counterparts. Conversely, SEM-k||means marginally outperforms SEM-k||means$_T^{RC}$ on the 2 billion point random univariate dataset because the data has no natural clusters. Accordingly, pruning efforts are less effective than on the real-world dataset.

As data increases in size, the difference between in-memory performance and the I/O bound SEM routines decreases. SEM algorithms are only 3-4x slower than their in-memory counterparts.

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 time elapsed of each routine.



(b) Memory consumption of each routine.

Fig. 10: Performance comparison on RM$_{856M}$ and RM$_{1B}$ datasets. Dato 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.

VII. 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 (k||means) and semi-external memory applications (SEM-k||means) as we outperform state-of-the-art frameworks running the exact same algorithms.

We formulate a minimal triangle inequality pruning technique (MTRI) that further boosts the performance of k-means on real-world billion point datasets by over 100x when compared to popular frameworks. MTRI 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.

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] ABELLO, J., BUCHSBAUM, A. L., AND WESTBROOK, J. R. A functional approach to external graph algorithms. In Algorithmica (1998), Springer-Verlag, pp. 332–343.

[2] BENNETT, J., AND LANNING, S. The netflix prize. In Proceedings of KDD cup and workshop (2007), vol. 2007, p. 35.

[3] BINKIEWICZ, N., VOGELSTEIN, J. T., AND ROHE, K. Covariate assisted spectral clustering. arXiv preprint arXiv:1411.2158 (2014).

[4] CITATION-ANNOYMIZED. Annoymized title.

[5] CITATION-ANNOYMIZED. Annoymized title.

[6] DAS, A. S., DATAR, M., GARG, A., AND RAJARAM, S. Google news personalization: scalable online collaborative filtering. In Proceedings of the 16th international conference on World Wide Web (2007), ACM, pp. 271–280.

[7] DEAN, J., AND GHEMAWAT, S. MapReduce: Simplified data processing on large clusters. In Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6 (2004).

[8] DEMPSTER, A. P., LAIRD, N. M., AND RUBIN, D. B. Maximum likelihood from incomplete data via the em algorithm. Journal of the royal statistical society. Series B (methodological) (1977), 1–38.

[9] ELKAN, C. Using the triangle inequality to accelerate k-means. In ICML (2003), vol. 3, pp. 147–153.

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

[11] FRIGO, M., LEISERSON, C. E., PROKOP, H., AND RAMACHANDRAN, S. Cache-oblivious algorithms. In Foundations of Computer Science, 1999. 40th Annual Symposium on (1999), IEEE, pp. 285–297.

[12] H2O. h2o. http://h2o.ai/, 2005–2015.

[13] JORDE, L. B., AND WOODING, S. P. Genetic variation, classification and'race'. Nature genetics 36 (2004), S28–S33.

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

[15] LOW, Y., GONZALEZ, J. E., KYROLA, A., BICKSON, D., GUESTRIN, C. E., AND HELLERSTEIN, J. Graphlab: A new framework for parallel machine learning. arXiv preprint arXiv:1408.2041 (2014).

[16] LYZINSKI, V., SUSSMAN, D. L., FISHKIND, D. E., PAO, H., CHEN, L., VOGELSTEIN, J. T., PARK, Y., AND PRIEBE, C. E. Spectral clustering for divide-and-conquer graph matching. Parallel Computing (2015).

[17] LYZINSKI, V., TANG, M., ATHREYA, A., PARK, Y., AND PRIEBE, C. E. Community detection and classification in hierarchical stochastic blockmodels. arXiv preprint arXiv:1503.02115 (2015).

[18] MENG, X., BRADLEY, J., YAVUZ, B., SPARKS, E., VENKATARAMAN, S., LIU, D., FREEMAN, J., TSAI, D., AMDE, M., OWEN, S., ET AL. Mllib: Machine learning in apache spark. arXiv preprint arXiv:1505.06807 (2015).

[19] OWEN, S., ANIL, R., DUNNING, T., AND FRIEDMAN, E. Mahout in action. Manning Shelter Island, 2011.

[20] PATTERSON, N., PRICE, A. L., AND REICH, D. Population structure and eigenanalysis.

[21] PEARCE, R., GOKHALE, M., AND AMATO, N. M. 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).

[22] SCULLEY, D. Web-scale k-means clustering. In ACM Digital library (2010), pp. 1177–1178.

[23] SHINDLER, M., WONG, A., AND MEYERSON, A. W. Fast and accurate k-means for large datasets. In Advances in neural information processing systems (2011), pp. 2375–2383.

[24] UGANDER, J., KARRER, B., BACKSTROM, L., AND MARLOW, C. The anatomy of the facebook social graph. arXiv preprint arXiv:1111.4503 (2011).

[25] ZAHARIA, M., CHOWDHURY, M., DAS, T., DAVE, A., MA, J., MCCAULEY, M., FRANKLIN, M. J., SHENKER, S., AND STOICA, I. 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 (2012), USENIX Association, pp. 2–2.

[26] ZAHARIA, M., CHOWDHURY, M., FRANKLIN, M. J., SHENKER, S., AND STOICA, I. Spark: Cluster computing with working sets. HotCloud 10 (2010), 10–10.

[27] ZHAO, W., MA, H., AND HE, Q. Parallel k-means clustering based on mapreduce. In Cloud Computing. Springer, 2009, pp. 674–679.