CS599: Parallel Programming
Semester Project Proposal
Chris Keefe & Anthony Simard

Abstract—We present a benchmarking study on parallel implementations of the Elkan and Lloyd's algorithms for unsupervised K-Means clustering. CPU and GPU parallelization approaches have been implemented in C using OpenMP and CUDA respectively. Results have been validated against a reference implementation of Lloyd's algorithm for K-Means clustering, and benchmarked against a serial implementation of Lloyd's algorithm, using large-scale data. Additional benchmarking explores the scalability of the CPU algorithm over 1 to 32 processors.

Keywords—k-means, clustering, Elkan Algorithm, Lloyd's Algorithm, parallelization, performance benchmarks, scalability

Introduction: K Means clustering is one of the most prominent methods for clustering unlabeled data into meaningful groups. It attempts to partition some set of data into k clusters, by iteratively assigning each observation to its nearest cluster center, and once all observations have been assigned, adjusting the position of the cluster centers to the mean position of all in-cluster observations, continuing until convergence is reached. This approach works for large data sets, and is commonly applied both independently and as data pre-processing step for more complex ML methods.

The traditional approach to K means clustering has a complicated and data-dependent time complexity, which can be superpolynomial in certain worst-case scenarios [1]. Improving its run time in practice has been the subject of significant research, including the development of many different algorithmic optimizations [2][3][4], and this study presents a developer-facing perspective on two approaches.

Background: The classical algorithm for k-means clustering is attributed independently to two authors, Lloyd [5] and Forgy [6], and will here be called Lloyd's algorithm. Since its introduction in the mid-20th century, researchers have taken many approaches to improving its performance, but Lloyd's algorithm has remained dominant. Ding et al argue that this a successful replacement must be well-trusted, produce consistent speedups, and be simple to develop. This third point seems to be critical.

Our study attempts to use shared-memory parallelism to improve algorithm performance, and shows that optimizations through basic algorithmic transformations may outperform naive implementations of more complex algorithms in practice. During our implementation and parallelization of Lloyd's and Elkan's [7] algorithms, we found that the simplicity and clarity of Lloyd's algorithm made simple optimizations straightforward. The algorithm proposed by Ding et al is far more imposing than either of these, which may work against it, despite its stronger performance characteristics.

Approach: We present a preliminary study of performance improvements to K-Means clustering, focused on CPU and GPU parallelizations of Lloyd's and Elkan's algorithms. We have implemented both algorithms using OpenMP for CPU parallelization, and CUDA for GPU parallelization. Our implementations were ground-truthed against a serial implementation of LLoyd's algorithm (possible because Elkan K-Means produces proven identical results to LLoyd's algorithm).

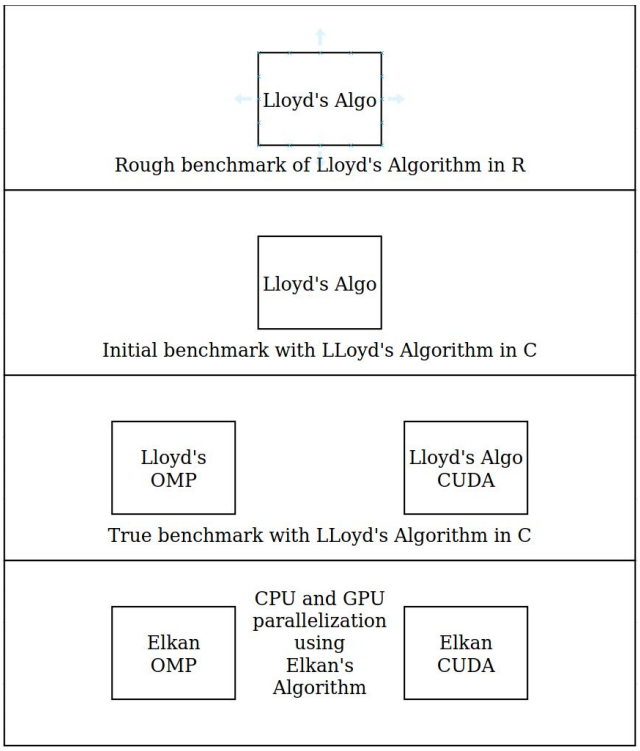

Figure 1: Software development overview

Overall performance was measured in terms of response time over a 113MB data set of real-number coordinate pairs, using dedicated **INSERT CPU SPEC** cpu and **INSERT GPU SPEC** gpu nodes. Scalability was measured for CPU implementations by computing speedup `(s = response time / n threads)`, and parallel efficiency (`pe = speedup / n cores)` for 1, 8, 16, 24, and 32 cores on the same dedicated compute node.

Data: Initial cluster centroids $c(1), c(2), \ldots, c(K)$ and feature vectors $x(1), x(2), \ldots, x(N)$

repeat
  {assignment step}
  for $i \leftarrow 1$ **to** N **do**
    $a(i) \leftarrow \arg\min_j \mathrm{d}(x(i), c(j))$
  {update step}
  for $j \leftarrow 1$ **to** K **do** $y(j) \leftarrow \mathbf{0}$, $z(j) \leftarrow 0$
  for $i \leftarrow 1$ **to** N **do**
    $y(a(i)) \leftarrow y(a(i)) + x(i)$
    $z(a(i)) \leftarrow z(a(i)) + 1$
  for $j \leftarrow 1$ **to** K **do** $c(j) \leftarrow y(j)/z(j)$
until TerminationCondition

Result: Final cluster centroids $c(1), c(2), \ldots, c(K)$ and cluster assignment $a(1), a(2), \ldots, a(N)$

Lloyd's Algorithm [2][5][6]

Data: Initial cluster centroids $c(1), c(2), \ldots, c(K)$ and feature vectors $x(1), x(2), \ldots, x(N)$

{Initialization of bounds}
for $i \leftarrow 1$ **to** N **do**
  $a(i) \leftarrow 1$, $u(i) \leftarrow \infty$
  for $k \leftarrow 1$ **to** K **do** $l(i,k) \leftarrow 0$
repeat
  Compute inter-centroid distance matrix C
  for $k \leftarrow 1$ **to** K **do** $s(k) \leftarrow \min_{j \neq k} C(j,k)/2$
  for $i \leftarrow 1$ **to** N **do**
    if $u(i) > s(a(i))$ **then**
      $r \leftarrow$ **true**
      for $k \leftarrow 1$ **to** K **do**
        $z \leftarrow \max(l(i,k), C(a(i),k)/2)$
        if $k = a(i)$ **or** $u(i) \leq z$ **then continue**
        if r **then**
          $u(i) \leftarrow \mathrm{d}(x(i), c(a(i)))$
          $r \leftarrow$ **false**
          if $u(i) \leq z$ **then continue**
        $l(i,k) \leftarrow \mathrm{d}(x(i), c(k))$
        if $l(i,k) < u(i)$ **then**
          $a(i) \leftarrow k$
          $u(i) \leftarrow l(i,k)$
  $c' \leftarrow c$, $c \leftarrow$ **UpdateStep**
  for $k \leftarrow 1$ **to** K **do** $\delta(k) = \mathrm{d}(c(k), c'(k))$
  for $i \leftarrow 1$ **to** N **do**
    $u(i) \leftarrow u(i) + \delta(a(i))$
    for $k \leftarrow 1$ **to** K **do** $l(i,k) \leftarrow l(i,k) - \delta(k)$
until TerminationCondition

Result: Final cluster centroids $c(1), c(2), \ldots, c(K)$ and cluster assignment $a(1), a(2), \ldots, a(N)$

Elkan's Algorithm [2][7]

Figure 2: Algorithms implemented

Results: All algorithms developed for this project were developed, verified, and initially timed on small-scale data, on local computers using 4-core intel i7 CPUs (8th gen and 10th gen). GPU parallelization was not possible on these machines for lack of hardware, so all CUDA code was verified and timed on Monsoon's Nvidia P100 GPU nodes. GPU code refinement was done on Monsoon, and all final timings were performed on Monsoon. Specifically, all CPU timings were taken on Monsoon's 32-core node `cn4`.

All code was verified against two reference implementations, after initial development. First, we vetted a serial version of Lloyd's algorithm we developed in R against the R stats package's `stats::kmeans` implementation, finding them to produce identical results. We then verified our C code implementation, against these, finding minor differences between our final cluster centers when run against the iris data set and those produced by the R packages. Final *clusterings* and number of iterations were identical, however. After extensive troubleshooting, we believe this was the product of floating-point error, but closer investigation might show otherwise.

Reference implementation results

```
Centers:
  Sepal.Length Sepal.Width Petal.Length Petal.Width
1     5.006000    3.428000     1.462000    0.246000
2     5.901613    2.748387     4.393548    1.433871
3     6.850000    3.073684     5.742105    2.071053
```

C implementation results

```
Centers:
  Sepal.Length Sepal.Width Petal.Length Petal.Width
1     5.006000    3.418000     1.464000    0.244000
2     5.901613    2.748387     4.393548    1.433871
3     6.850000    3.073684     5.742105    2.071053
```

Figure 3: Validation results from our Serial Lloyd's implementation of K means. Note slight differences in cluster center positions across implementations, in bold.

Initial timings were calculated for all CPU implementations on one of our local machines, using a set of ~five million two-dimensional coordinate pairs. All timings used the same seed to ensure the same initial centers were chosen across implementations for a given K. Only the algorithm itself is timed, not the reading of the data or the initial center selection. The mean value of three timings was used, to reduce noise. We used OpenMP's `omp_get_wtime` function across the board for consistency.

These initial timings produced our first interesting result. Because Lloyd's algorithm was so simple to parallelize, we found that we were able to optimize it significantly enough to consistently outperform Elkan's algorithm.

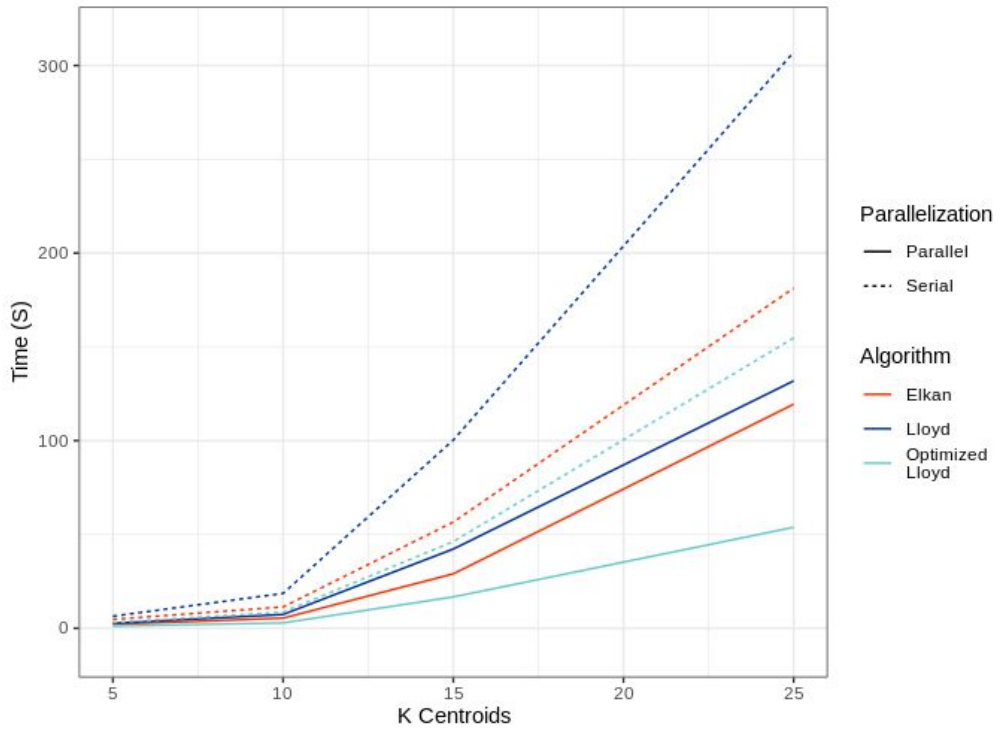

Figure 4: Initial (rough) timing results for all algorithms. Parallel computations performed with four threads, on an 8th-gen intel core-i7 processor.

These Lloyds' algorithm optimizations focused on reductions in the number of distance calculations, and reduction in number of floating-point operations, including the removal of the square root calculation. Notably, we believe this removal would preclude reporting true within-cluster sum of squares values, which are often useful in measuring model performance. This issue could be alleviated in production software by calculating the sum of squares *only once the final clusterings have been selected*, dramatically improving algorithm performance without sacrificing utility. This was outside of the scope of our work, however, so we were not able to compare timings of this approach to Elkan's algorithm.

Final timings were computed on the same data, and with the same practices as were used in the rough benchmarks described above. Unfortunately, significant and implementation changes had taken place between the initial verified implementation, and the final timing implementation. In the process, the parallel naive Lloyd's algorithm was broken, and insufficient time remained to correct the issue. Timings have been included here for completeness, but the `INF` values produced by naive parallel Lloyd's make it very clear that they may not be trusted.

Our results from the parallel implementation of Elkan's algorithm are also rather unexpected, though we have not had sufficient time to investigate thoroughly. The algorithm produces correct clusterings at all tested sizes of K, but takes far longer to compute k=5 clusterings than k=10. It takes 1.555x the number of iterations to calculate k=10 cluster centers, so this result surprised us.

We suspect that the increased space complexity of Elkan's algorithm may introduce significant enough overhead that scaling up the number of CPU cores negatively impacts run time with this data for small values of K.

Run times for different numbers of centroids K, by algorithm.

Cores	Algorithm	K=5	K=10	K=15	K=25
1	Naive Lloyd	10.993623	30.388399	*155.113633*	*498.879177*
32	Naive Lloyd Par	~~0.60198~~	~~1.576566~~	~~1.27542~~	~~2.038268~~
1	Optimized Lloyd	4.914567	13.868914	70.262761	227.059771
32	Optimized Lloyd Par	1.311343	2.108923	8.796894	13.776687
1	Elkan	6.044888	20.556103	57.894389	266.198318
32	Elkan Par	*45.11826*	*36.994304*	138.62655	309.234684
GPU	Lloyd CUDA	**0.593733**	**0.843285**	**2.828949**	**4.341726**
GPU	Elkan CUDA	0.641329	1.238445	5.747807	19.790917

Figure 5: Final timings. Best performance per value K is indicated by boldface, and worst performance per value K is indicated by italics. Failed runs indicated with strikethrough.

The remaining timings are in alignment with our expectations and initial rough testing. Lloyd's algorithm, as optimized and implemented for GPU with CUDA, significantly outperformed all other implementations for all values of K. At small values of K, parallel Elkan was the worst performer, but the extra overhead Elkan introduces was overshadowed by the poor scaling of our naive, serial Lloyd implementation for K=15 and higher.

We can view these results in a more informative way, by considering each implementation's speedup over a common baseline: the Serial Naive Lloyd implementation. GPU Optimized Lloyd's outperformed Serial Naive Lloyd 114x at K=25, and that margin is very likely to increase as the number of centroids, size, or dimensionality of the data increases. Parallel Elkan, at k=5, actually underperformed the baseline Serial Lloyd implementation by 4x; a stark difference, given the overall short run times at that scale. Graphing this data would make it far more approachable, but will not be possible after the significant time lost to GPU access issues, and other quirks of the Monsoon computing environment.

Speedup over Serial Naive Lloyd for different numbers of centroids K, by algorithm

Cores	Algorithm	K=5	K=10	K=15	K=25
1	Naive Lloyd	1	1	*1*	*1*
32	Naive Lloyd Par	~~18.26243895~~	~~19.27505667~~	~~121.6176891~~	~~244.7564192~~
1	Optimized Lloyd	2.236946409	2.191115973	2.207622228	2.197127104
32	Optimized Lloyd Par	8.383483955	14.40943979	17.63277277	36.21183939
1	Elkan	1.818664465	1.478315175	2.679251577	1.874088389
32	Elkan Par	*0.2436623886*	*0.8214345376*	1.11893164	1.613270447
GPU	Lloyd CUDA	**18.51610572**	**36.03573999**	**54.830834**	**114.9034225**
GPU	Elkan CUDA	17.14193963	24.53754426	26.98657645	25.20748164

Figure 6: Speedup. Best performance per value K is indicated by boldface, and worst performance per value K is indicated by italics. Failed runs indicated with strikethrough.

CPU scalability: While the primary focus of our work here was the development, parallelization, and comparison of these algorithms, we also considered the scalability of both algorithms over varying numbers of CPU cores. Our optimized Lloyd's algorithm, when run on 32 cores, performed nearly as well as Elkan CUDA. Given that many users may not have ready access to CPU computing facilities, quantifying speedup and parallel efficiency on the CPU is particularly relevant.

After a quick review of the literature showed that very few papers discuss K means clustering with values of K greater than 36, we chose k=15 as a reasonably representative hyperparameter for a first look at CPU scaling. With more time, we would have tested more extensively.

Our testing revealed a dramatic difference in the way performance changed between the two algorithms. While Lloyd's algorithm sped up predictably as we added cores, Elkan's algorithm actually underperformed for all tested numbers of cores greater than 1. Interestingly, performance was worst with 8 cores, and then increased slightly for 16 and 24 before dropping off again. In all cases, a roughly 50% performance decrease was measured.

Speedup and Parallel Efficiency

Cores	Lloyd Speedup	Elkan Speedup	Lloyd Par Ef	Elkan Par Ef
1	1	1	1	1
8	5.137935495	0.446906369	0.6422419369	0.05586329612
16	7.573268992	0.5225017717	0.473329312	0.03265636073
24	9.824686017	0.5619740159	0.4093619174	0.023415584
32	10.73289948	0.5372592848	0.3354031088	0.01678935265

Speedup calculations for each algorithm performed against single-core timing of the same algorithm. All speedup timings taken with K=15 centroids.

Lloyd's algorithm, as expected, speeds up predictably as we add cores, but loses parallel efficiency along the way. We believe this is due to the increased overhead associated with splitting the computation across greater and greater numbers of threads.

Discussion and Conclusions

Implementation and comparison: Our work here reinforces the idea put forward by Ding et al, that ease of implementation is an important characteristic in the viability of an algorithm in production contexts. While the unexpected timings in our Elkan implementation raise a number of interesting questions, the complexity of the algorithm makes it harder to reason about exactly why we're witnessing this behavior. Simultaneously, the simplicity of Lloyd's algorithm exposes many opportunities for low- or no-cost optimizations, which maintain the readability of the codebase while outperforming a relatively naive implementation of the "faster" algorithm.

It should be noted that our Elkan implementation is not "naive" by choice. We put significant time into trying to further reduce the number of computations and floating point operations performed by that algorithm, but were unable to get it to outperform Lloyd's. It was far harder to develop, especially for GPU use, and would be far harder to maintain for its complexity. This makes it easy to understand why algorithms like Elkan, and to an even greater degree, Ding et al's Yinyang, have not seen the same level of penetration into production software as might be expected based on algorithmic performance alone.

Scalability: Though we did limited scalability testing for this investigation, our work yielded perhaps the most useful result from this investigation. Parallelizing Lloyd's algorithm over eight cores, which is possible on high-end consumer laptops today, can yield a five-fold increase in performance over a serial run of the same code.Users with access to a compute cluster will continue to benefit from increasing the number of cores, but parallel efficiency drops significantly as they do so, dramatically reducing the value of using more than 32 cores for computations of this nature, given the size and characteristics of this data. Extension of this study across additional data types, CPU counts, and systems, would produce more generalizable

results, but it's useful to know that a simple algorithm, and a user-friendly approach to parallelization, can yield significant end-user performance benefits.

Appendix 1: Software

Due to the relatively extensive software development required by this project, we have opted to submit code as a publicly available repository, in addition to the compressed file format required. All software products from this work, including monsoon-compiled binaries and select results are publicly available under MIT license at https://github.com/ChrisKeefe/599ParallelProject/

Due to filesize restrictions, the test data set is not publicly hosted.

Milestones

a. ~~Lloyd's Algorithm benchmark in R (10/18/20) - complete~~
b. ~~Lloyd's Algorithm in C (11/1/2020) - complete~~
c. ~~Multithread C Lloyd's using OpenMP (11/1/2020) - complete~~
d. ~~Implement Elkan in C (11/8/2020)~~
e. ~~Create a CUDA implementation of Lloyd's (11/15/2020)~~
f. ~~Create a CUDA implementation of Elkan (11/15/2020)~~
g. ~~Parallelize Elkan with OpenMP (11/22/2020)~~
h. ~~Benchmark and compare all above implementations (11/26/2020)~~

References

[1] D. Arthur and S. Vassilvitskii. "How slow is the *k*-means method?" in Proc. 22nd SoCG (SCG '06), Association for Computing Machinery, New York, NY, USA, 2006, pp. 144–153. DOI:https://doi.org/10.1145/1137856.1137880

[2] W. Kwedlo and P. J. Czochanski, "A Hybrid MPI/OpenMP Parallelization of K-Means Algorithms Accelerated Using the Triangle Inequality," in IEEE Access, vol. 7, pp. 42280-42297, 2019, doi: 10.1109/ACCESS.2019.2907885.

[3] Y. Ding et al, "Yinyang K-Means: A Drop-In Replacement of the Classic K-Means with Consistent Speedup," in Proc. 32nd Int. Conf. Mach. Learn. (ICML), vol. 37. Lille, France, 2015.

[4] A. Jain, "Data clustering: 50 years beyond K-means," in Pattern Recognition Letters, vol 31, no. 8, pp. 651–666, 2010.

[5] S. Lloyd, ‘‘Least squares quantization in PCM,’’ IEEE Trans. Inf. Theory,vol. IT-28, no. 2, pp. 129–137, Mar. 1982.

[6] E. W. Forgy, ‘‘Cluster analysis of multivariate data: Efficiency versus interpretability of classifications,’’ Biometrics, vol. 21, no. 3, pp. 768–769,1965.

[7] C. Elkan, ‘‘Using the triangle inequality to accelerate k-means,’’ in Proc. 20th Int. Conf. Mach. Learn. (ICML), vol. 3. Menlo Park, CA, USA: AAAI,2003, pp. 147–153.